Chair of Electrical Design Automation
TUM School of Computation, Information and Technology
Technical University of Munich

From Simulation to RVV Hardware: Evaluating the muRISCV-NN TinyML Inference Library on the CanMV K230 Platform

Research practice

Author:	Benedikt Witteler
Advisor:	Philipp van Kempen
Supervisor:	Prof. Dr.-Ing. Ulf Schlichtmann
Submission date:	15. October 2024

Abstract

The ratification of the RISC-V Vector Extension (RVV) is bringing new momentum to the RISC-V ISA in the field of embedded machine learning. muRISCV-NN is a library of efficient deep learning kernels that leverages RVV for inference on constrained edge devices. With the CanMV K230 development board - the first commercial hardware implementing RVV 1.0 - this study evaluates whether muRISCV-NN can perform as effectively on real hardware as in an instruction set simulator (ISS). Porting muRISCV-NN to the CanMV K230 involved addressing challenges like compiler issues and its integration within RTOS and Linux environments. The results of this project show that muRISCV-NN meets its performance expectations, achieving a 3.85x cycle count reduction during ResNet inference through vectorization. Layer-wise benchmarking with the MLonMCU framework revealed speedups of up to 6.68x in fully connected layers and up to 4.40x in Conv2D layers, with larger layers profiting more from vectorization than smaller ones. These findings confirm muRISCV-NN's effectiveness on physical hardware, supporting the broader adoption of RISC-V in embedded AI.

Contents

Contents **1**

1 Introduction **2**

2 State of the Art **4**
2.1 RISC-V Vector Extension 4
2.2 CanMV K230 Board 4
2.3 muRISCV-NN 5
2.4 MLonMCU 7

3 Tasks and Challenges **8**
3.1 Getting the K230 SDK Up and Running 8
3.2 Porting muRISCV-NN to the CanMV K230 9
3.2.1 Toolchain Issues 9
3.3 Alternative Setup: Running muRISCV-NN in a Linux Environment on the CanMV K230 10
3.4 Experimental Setup 10
3.4.1 RTOS 10
3.4.2 Linux 11

4 Experimental Results **12**
4.1 Setup 12
4.2 RTOS 12
4.3 Linux 14

5 Conclusion **16**

A Figures **17**

List of Figures **24**

List of Tables **25**

Bibliography **26**

Chapter 1

Introduction

In recent years, the increased need for efficient machine learning on the edge has driven innovations in the embedded domain. With its open-source nature and flexibility, the RISC-V instruction set architecture (ISA) has become an attractive choice for developing embedded machine learning solutions. The recent ratification of the RISC-V Vector Extension (RVV) 1.0 brings notable improvements in vector processing capabilities, which can significantly enhance the performance of machine learning inference on constrained platforms. Until now, however, there has been a lack of practical evaluations on actual hardware to verify these potential benefits.

Canaan Inc.'s CanMV K230 development board is the first commercial hardware to feature a processor with a vector processing unit (VPU) supporting the ratified RVV 1.0. The muRISCV-NN library offers a suite of efficient deep learning kernels optimized for RVV and provides a lightweight, platform-independent alternative to existing machine learning inference libraries. The primary goal of this research is to evaluate the effectiveness of muRISCV-NN on the CanMV K230 platform and to assess whether its performance matches the results observed in an instruction set simulator (ISS). The project aims to address the challenges of porting muRISCV-NN to the CanMV K230 and provides a detailed performance analysis of neural network inference utilizing vectorization on this platform. Specifically, the tasks involved configuring the K230 software development kit (SDK), overcoming toolchain compatibility issues, and setting up an experimental framework to benchmark various neural network models in both real-time operating system (RTOS) and Linux environments.

Through this project, the performance of muRISCV-NN was tested for anomaly detection, image classification, visual wake word, and keyword spotting neural network models, providing insights into how RVV vectorization impacts different neural network layers. The experimental results demonstrate significant cycle and instruction count reductions, supporting the broader adoption of RISC-V for embedded artificial intelligence (AI) applications.

The remainder of this report is structured as follows. In Chapter 2 the backgrounds of this work are explained by introducing RVV, the CanMV K230 development board, the muRISCV-NN library, and the MLonMCU benchmarking framework. Chapter 3 outlines how muRISCV-NN was ported to the CanMV K230 to obtain a hardware and software setup for benchmarking. The different performed tasks as well as the faced challenges are detailed in this section and the respective experimental results are presented in Chapter 4. Finally, this work is concluded in Chapter 5.

Chapter 2

State of the Art

2.1 RISC-V Vector Extension

The RISC-V ISA was introduced in 2010 by the computer science team at UC Berkley. Unlike proprietary ISAs such as *Intel's x86* or the ones of the *arm* family, RISC-V is offered under the *BSD license* and free to use. RISC-V follows the *reduced instruction set computer* (RISC) principle, making it very minimal in its base configuration. The ISA is very modular, however, and can easily be extended. Next to custom extensions designed by vendors for their RISC-V implementations, there exist numerous standard extensions ratified by *RISC-V International* [4].

The RISC-V vector extension proposed in [5] was ratified in November 2021 [4]. It extends the scalar RISC-V ISA by 32 new vector registers of length VLEN bits each and further introduces seven control and status registers (CSRs). One of RVVs strengths is that it is inherently vector length agnostic [13]. This means that the same compiled code can be executed on different implementations of RISC-V processors with different vector register lengths VLEN. Additionally, the size of vector elements as well as the number of elements in a vector can be changed at runtime. This is archived by introducing the selected element width (SEW) and length multiplier (LMUL) parameters that are stored in the *vtype* CSR. The SEW specifies the size of a vector element in bits and the LMUL is an integer value indicating how many vector registers are grouped to form a single vector, allowing it to span over multiple registers. If LMUL holds a fractional values (1/2, 1/4, 1/8, ...), multiple vectors are stored inside a single vector register.

2.2 CanMV K230 Board

The CanMV K230 development board released in 2023 is the first commercial hardware featuring a processor that implements the ratified RISC-V vector extension version 1.0. The *Canaan Inc.* board comes with the *Kendryte K230 Dual-Core C908 64-bit* processor with two differently configured *XuanTie C908* cores. This 64-bit core was presented in 2022 by *T-Head Semiconductor*. It implements a nine-stage dual-issue in-order pipeline, a two-level cache system and supports branch prediction [14]. Manufactured under TSMC's 12 nm process, the core can be clocked with a maximum frequency of 2 GHz resulting in a dynamic power consumption of 52.8 mW/GHz per core. Additionally, the *XuanTie C908* includes an optional vector processing unit that is compatible with RVV 1.0. While its 2020 predecessor *XuanTie C906* featured VPU support as well, it was not compatible with RVV 1.0 but only with version 0.7.1 [13].

From the two cores inside the CanMV K230's CPU, only the more powerful one implements RVV 1.0 with a vector register length of 128 bits. Its maximum clock frequency is 1.6 GHz. The second core clocks with up to 800 MHz and does not have a VPU. With this configuration the *Kendryte K230 Dual-Core C908 64-bit* is not a multiprocessor in the common sense with two synchronized cores. Instead, the two cores work mostly independently of each other. In the boards standard development kit, the K230 SDK, the 800 MHz core runs a Linux system as a convenient way to access the board while the 1.6 GHz core runs the RTOS *RT-Thread*, inside which compute intensive applications such as machine learning inference can be executed. Following the naming convention in the board's documentation, the 1.6 GHz core implementing RVV will be called *big core* and the 800 MHz core not implementing RVV will be called *small core* in this report.

Next to the aforementioned CPU, the CanMV K230 is equipped with 512 MiB of LPDDR3 RAM and a micro SD card serving as permanent memory. It further comes with a built-in knowledge processing unit (KPU), dedicated graphics hardware acceleration units and a multimedia subsystem supporting 1080p HDMI output and an integrated camera [1]. Various peripherals such as USB 2.0, UART and I2C are also provided. Complete technical specifications of the CanMV K230 are detailed in [1] and an overview is illustrated in Figure 2.1. The focus of this work, however, is on the RVV 1.0 VPU of the *XuanTie C908* core and most other features are not further examined.

2.3 muRISCV-NN

muRISCV-NN [15] is a suite of optimized deep learning kernels designed specifically for embedded systems and microcontrollers. Created by the Chair of Electronic Design Automation at the Technical University of Munich, this library addresses the need for a lightweight, open-source, and platform-independent compute library that can fully utilize RVV 1.0 and the RISC-V packed "P" extension for machine learning loads. muRISCV-NN is based on *arm*'s CMSIS-NN inference library [11] but targets the RISC-V platform instead of *arm Neon* or *Helium* processors. It can act as a drop-in replacement for the latter since bit-accuracy to CMSIS-NN is ensured. As such, muRISCV-NN can be used with embedded machine learning frameworks like TensorFlow Lite for Microcontrollers (TFLM) [10] and microTVM [9]. The library is compatible with both the RISC-V GNU Compiler and LLVM [12]. While both compilers offer autovectorization, the resulting code does not take full advantage of RISC-V's length agnostic vector capabilities, as illustrated in [8]. Through its manually optimized operator implementations, muRISCV-NN achieves up to a 60% runtime improvement for convolutional models compared to LLVM's built-in

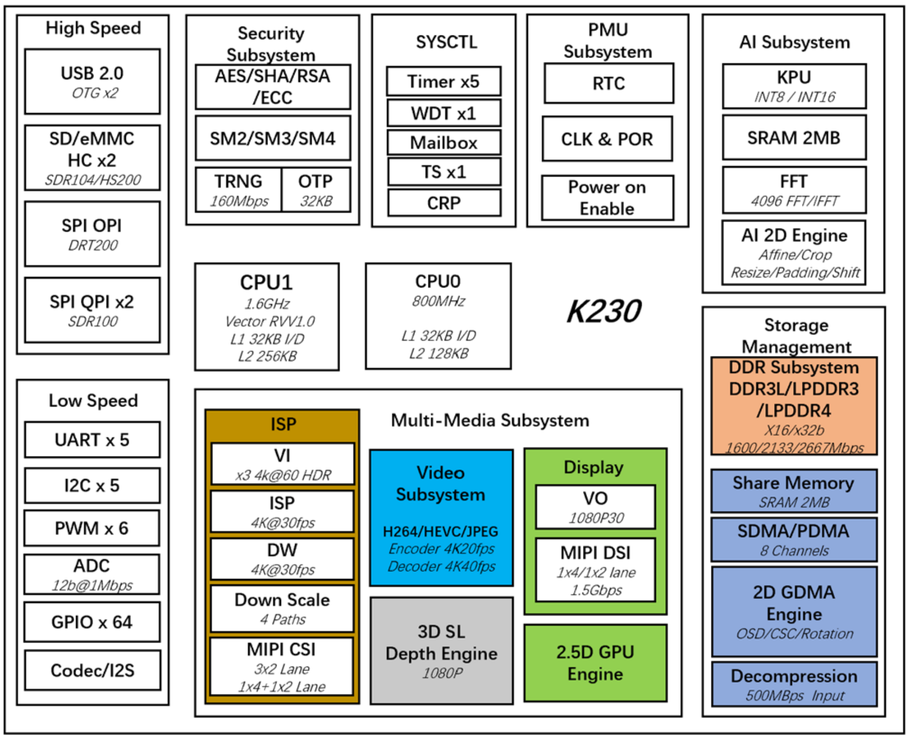

Figure 2.1: CanMV K230 block diagram from `https://github.com/kendryte/k230_docs/blob/main/en/CanMV_K230_Tutorial.md`.

autovectorization when tested on an instruction set simulator.

2.4 MLonMCU

MLonMCU [16] is a tool designed to benchmark TinyML frameworks on microcontrollers, streamlining the process of comparing various frameworks and configurations. It allows users to perform comprehensive evaluations efficiently, with minimal manual intervention, by automating the benchmarking flow. By supporting a range of platforms and the leading TinyML frameworks TFLM and TVM, MLonMCU provides valuable insights into analyzing and optimizing machine learning applications for constrained edge devices.

Chapter 3

Tasks and Challenges

3.1 Getting the K230 SDK Up and Running

To enable custom software execution on the CanMV K230, the first step of the project was to get the vendor's standard development kit up and running on the board. In this so-called K230 SDK, the small 800 MHz core of the CPU runs a Linux system providing convenience functionalities, for example, for interfacing with the board. The big 1.6 GHz RVV core runs an RTOS inside which the benchmarks of this project have been measured.

While a lot of documentation is provided for the board and its firmware, working with it was not easy. Firstly, large amounts of the materials are in Chinese and the English translations are often not very good. More problematic, however, is the way the documentation is structured. The repository [7] is quite messy and not intuitive to work with. While some tutorials are provided, it is very hard to find specific pieces of information in the poorly structured list of Markdown pages.

Following the instructions in [2], the K230 SDK could be built using the cumbersome *Docker* build provided by the vendor. The produced image file was written to a micro SD card to get the CanMV K230 running without having to make changes to the firmware. To control the two individual cores of the board, both the Linux system on the small core as well as the RTOS on the big core offer a UART interface. Basic operations such as the program launch for a benchmark can be performed in separate command line interfaces.

The initial way to transfer a cross-compiled application to the CanMV K230 is by physically removing its micro SD card, copying the application from the host PC to a reserved partition on the card, re-inserting it into the CanMV K230, and relaunching the board. This procedure is very time consuming and prohibits any automated benchmarking. To improve this, a LAN connection and an SSH server were established on the Linux system of the CanMV K230. This allows sending application binaries to the small core via the *secure copy protocol* (SCP). By storing them in a memory location that is also accessible to the big core, the benchmark applications of this project could be executed on the RVV hardware. The functionality of the setup was verified by running a demo image scaling application on the big core [3]. In this demo, enabling RVV already showed a 1.45x execution time reduction compared to pure scalar execution.

3.2 Porting muRISCV-NN to the CanMV K230

Porting the muRISCV-NN inference library to the CanMV K230 platform involved several key steps. The process required modifications to the library's build setup, addressing toolchain compatibility issues, and coming up with a two-step build to successfully run the muRISCV-NN benchmark code on physical hardware rather than an ISS.

The first task was to adapt the muRISCV-NN *CMake* build system to the CanMV K230 specifics. Following the example of different build configurations of muRISCV-NN for ISS, a suitable configuration for the CanMV K230 hardware was implemented. It was important to make sure that compiler and linker flags for the library build were identical to the ones used in K230 SDK demo projects. Compiler flags: *-march=rv64gcv -mabi=lp64d -mcmodel=medany*. Linker flags: *–static -T link.lds* (with *link.lds* the vendor linker script).

3.2.1 Toolchain Issues

The first major toolchain issue was encountered when using the vendor-provided toolchain, which utilizes GCC version 12.0.1, to build a muRISCV-NN benchmark with vectorization enabled. GCC version 12.0.1 does not fully support the explicit vector instructions required by muRISCV-NN, as vector intrinsics have undergone changes in GCC 13. The muRISCV-NN library relies on these updated intrinsics for leveraging RVV. As a result, attempting to compile the library with the vendor's GCC resulted in compilation errors due to the absence of the necessary intrinsic functions.

To circumvent this problem, compiling with standard GCC 13 was considered, which successfully built the vectorized code due to its support for the updated C intrinsics. However, this led to a second problem: the executables built with GCC 13 could not be run on the CanMV K230. This incompatibility was traced back to specific system calls used by the RTOS running on the CanMV K230. These system calls are built into the vendor toolchain but their source code is not available for integration. As a result, using GCC 13 directly to compile the entire application meant that the necessary RTOS-specific system calls were missing, which rendered the resulting binaries unusable on the K230 hardware.

To resolve these issues, a two-step build process was developed. First, muRISCV-NN was compiled using standard GCC 13 and stored as a static library *libmuriscvnn.a*. This allowed using the updated vector intrinsics and enabled full utilization of RVV capabilities during the library's build process. The second step involved building the final executable, containing benchmarking code, for example, using the vendor's GCC 12.0.1 toolchain, which linked the previously compiled static *libmuriscvnn.a*. This approach leveraged the specific system calls and runtime environment embedded within the vendor's toolchain, thus ensuring the final executable was compatible with the K230's RTOS. By linking the library built with GCC 13 to the main application compiled with the vendor toolchain, it was possible to achieve the best of both worlds.

3.3 Alternative Setup: Running muRISCV-NN in a Linux Environment on the CanMV K230

While the two-step build process provides a functional way to run muRISCV-NN benchmarks within an RTOS on the big core of the CanMV K230, it has certain limitations. A key downside is the inability to leverage the potential of MLonMCU for deployment and extensive benchmarking of TinyML applications on the CanMV K230. MLonMCU is a tool for benchmarking, offering automated deployment and analysis with minimal manual effort. However, it does not support applications that require multiple toolchains during the build process, as was necessary in the RTOS setup. To address this challenge, an alternative firmware setup was implemented. Instead of relying on the vendor's K230 SDK, a minimal Debian Linux environment was installed on the big core, allowing the benchmarks to run directly within a Linux environment. This setup provides a consistent toolchain environment, simplifying the benchmarking process and making it fully compatible with MLonMCU.

The official vendor SDK for the CanMV K230 includes a Linux system only for the small core. Therefore, getting Linux to run on the big core required following the instructions provided in [6]. The main steps involved flashing a custom system image to the micro SD card and using *Debootstrap* to bootstrap Debian on the system. The resulting system consists of a bootloader and a Debian Linux kernel running exclusively on the big core. This configuration makes all 512 MiB of RAM and the entire micro SD card storage available to the big core, with the small core disabled and no operating system running on it. While WiFi is not present in this setup, a LAN interface is available for communication.

3.4 Experimental Setup

For the experiments of this study, the neural network inference benchmarks were executed either within an RTOS or within a minimal Linux environment. While the results in an RTOS come closer to true bare metal performance, more extensive benchmarking is possible in the Linux environment thanks to MLonMCU. The experimental setup differs slightly in both cases.

3.4.1 RTOS

For benchmarks executed within the RTOS, first the source code of muRISCV-NN and then the source code of the benchmark itself are compiled in the previously discussed two-step build process. The resulting ELF file is then transferred to the CanMV K230 via SCP. Control of the big core is achieved through a UART interface from the host PC, which allows the execution of the benchmark remotely. Once it completes, the results are transmitted back to the host for analysis. All steps are automated using a script.

3.4.2 Linux

For the Linux environment benchmarks, MLonMCU is used to streamline the entire process, from compilation to execution. In this case, both muRISCV-NN and the benchmark application are built consecutively in a single build process. The resulting ELF file is transferred to the big core of the CanMV K230, where it is executed. Both transfer and execution are managed automatically via SCP and SSH, respectively. This is not possible in the RTOS configuration, in which only the small core is connected to a network via LAN, while the big core is only accessible via UART.

Chapter 4

Experimental Results

4.1 Setup

For this project, the same MLPerfTiny neural network models were used for benchmarking as those used in [15]. All models have been quantized to *int8* format. The models have different use cases and varying quantized sizes (values as reported in [15]):

- *toycar*: anomaly detection, 270 kB
- *resnet*: image classification, 96.2 kB
- *vww*: visual wake words, 325 kB
- *aww*: keyword spotting, 58.3 kB

For all experiments, GCC's compiler auto-vectorization feature was deactivated. All vector operations in the benchmarks go back to the explicit RVV instruction used in muRISCV-NN.

4.2 RTOS

Using compiler optimization level -*Os*, instruction counts for benchmarks run on CanMV K230 hardware and the ISS were nearly identical, with only minor differences attributable to different compiler versions.

Vectorization led to notable reductions in both cycle count and instruction count as can be seen in Figure 4.1. The *toycar* benchmark achieved the highest cycle reduction (5.43x), while the *ic* benchmark had the largest instruction count reduction (5.63x). These improvements highlight the effectiveness of vector execution with RVV 1.0 in reducing both computation complexity and runtime on real hardware. The Clock Cycles Per Instruction (CPI) was generally higher for vector execution compared to scalar execution (for *toycar* exception see next section) as illustrated in the table in Table 4.1. This is logical since

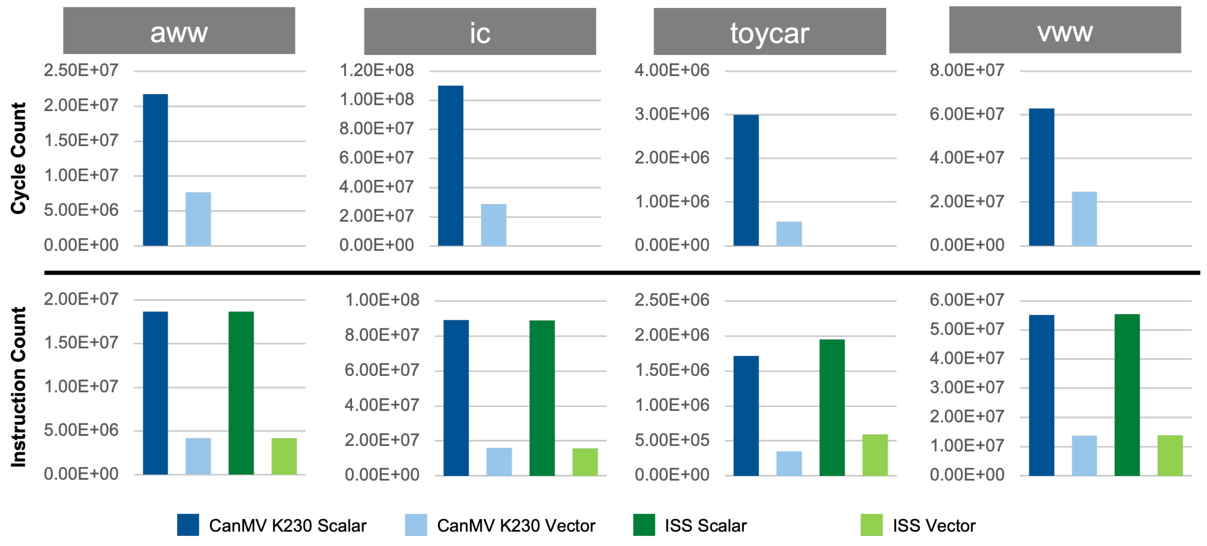

Figure 4.1: Cycle and instruction counts for *aww, ic, toycar,* and *vww* benchmarks with vectorization enabled and disables. The instruction count on the CanMV K230 hardware platform is also compared to the results on the ISS.

	Reduction Cycles	Reduction Instructions	CPI Scalar	CPI Vector
aww	2.83	4.47	1.16	1.83
ic	3.85	5.63	1.24	1.81
toycar	5.43	4.95	1.75	1.6
vww	2.53	4.03	1.14	1.81

Table 4.1: Cycle and instruction count reductions for the four different benchmarks as well as the CPIs comparing vector to scalar execution.

vector instructions combine multiple scalar operations into one, resulting in fewer overall instructions that often execute for more than one clock cycle.

The same benchmarks have been executed with compiler optimization level *-O3*. In this configuration as well, instruction counts could be reduced through vectorization. Cycle counts, however, increased when both aggressive level *-O3* optimizations and explicit vectorization in muRISCV-NN were activated. It can be concluded that both features interfere with each other.

4.3 Linux

With the help of MLonMCU, layer-wise benchmarking was possible within the Linux environment. This revealed further insights. All benchmarks in this section have been recorded with compiler optimization level *-Os*. With the Linux kernel running in the background during all benchmarks, utilizing more resources than the RTOS in the previous section, there are higher inaccuracies in all measurements. Fortunately, no particular errors occurred that could be traced back to extensive Linux kernel loads.

The *toycar* model only consists of ten dense layers of different sizes as depicted in Figure A.2. Vectorization yielded significant speedups, with an overall cycle count reduction of 5.44x and an instruction count reduction of 5.73x. The chart in Figure A.1 reveals that the extent of speedup for the individual layers varied depending on their sizes. Larger layers demonstrated strong performance improvements of up to 6.68x (cycle count) and 6.31x (instruction count), while smaller layers showed only modest gains. Layer 5, for example, configured with 128x8 weights, experienced no to minimal speedups - 0.96x for cycle count and 1.77x for instruction count - due to its small size. This limited performance boost is attributed to the short input feature vectors (8 elements), which are insufficient to fully leverage RVV vector execution. Further optimization efforts for layer 5 would be ineffective, however, as it constitutes only about 3% of the overall runtime. To some extent, the layer-wise benchmark re-exhibits the aforementioned CPI reduction with vectorization enabled that only occurred in the RTOS *toycar* benchmark. While the overall CPI with vectorization enabled is higher in this benchmark (1.98 compared to 1.86), the generally low CPI stems from the relatively low CPI for dense layers with vectorization enabled. At this point, it is unclear why dense layers achieve particularly low CPIs while using vectorization.

As depicted in Figure A.4, the *aww* model comprises a sequence of alternating pointwise Conv2D and DepthwiseConv2D layers (pairs also known as *Pointwise Conv2D*), along with one AveragePool2D, one Reshape, one dense, and a final softmax layer in the end. Figure A.3 highlights that both types of convolutional layers benefit from vectorization, with regular Conv2D layers achieving maximum speedups of 2.74x for cycle count and 4.21x for instruction count. The speedups are even more pronounced in DepthwiseConv2D layers, which realize reductions of up to 4.05x and 7.38x for cycle and instruction counts, respectively, indicating a greater potential for vectorization in this layer type. In contrast, the single AveragePool2D layer showed limitations when vectorization was enabled. Further optimization of this layer was not prioritized, as it contributes to only about 3% of the overall runtime.

For the most part, the *resnet* model is constituted of Conv2D layers that are sometimes traversed by parallel data streams, as illustrated in Figure A.6. Figure A.5 demonstrates that the speedups for those layers are similar to the ones exhibited for *aww*, reaching up to 4.40x and 6.24x for cycle count and instruction count, respectively for the particularly large layer 9. The three Add layers in this model can be vectorized particularly efficiently, reaching reductions of up to 5.59x and 23.3x for cycle and instruction counts, respectively.

Since the *vww* model has the same structure as *aww*, only with more repetitions of Conv2D and DepthwiseConv2D layers, its layer-wise analysis does not provide new insights and is therefore ommitted in this report.

Chapter 5

Conclusion

This project has successfully demonstrated the performance of the muRISCV-NN library on the CanMV K230 development board, validating its efficient deep learning inference capabilities using RVV not only on an ISS but also on physical hardware. Detailed benchmarking revealed substantial performance improvements. Through vectorization, cycle counts as well as instruction counts could be reduced for four different neural network model benchmarks. These results support the wider adoption of RISC-V for embedded AI applications.

Porting the muRISCV-NN inference library to the CanMV K230 presented several challenges, such as toolchain compatibility issues and insufficient vendor software support. However, these challenges were mitigated by implementing a two-step build process to enable RVV vectorization and running the benchmarks not only in an RTOS but also in a Linux environment.[1]

In future work, this research can be further expanded. Firstly, the recently introduced *Banana PI BPi-F3 SpacemiT K1* presents a hardware platform implementing RVV 1.0 that is far more suitable for machine learning applications as it comes with eight vector cores as opposed to a single one in the CanMV K230 as well as up to 16 GB of RAM. Porting muRISCV-NN to a multicore platform promises even larger improvements in runtime than presented in this work. Secondly, building a real-world application based on muRISCV-NN would provide deeper insights into its practical utility and allow for testing under real-world constraints. Finally, continued optimization of specific neural network layers based on hardware benchmark results could yield even greater performance gains.

[1]Code changes are not upstreamed yet. A pull request will follow.

Appendix A

Figures

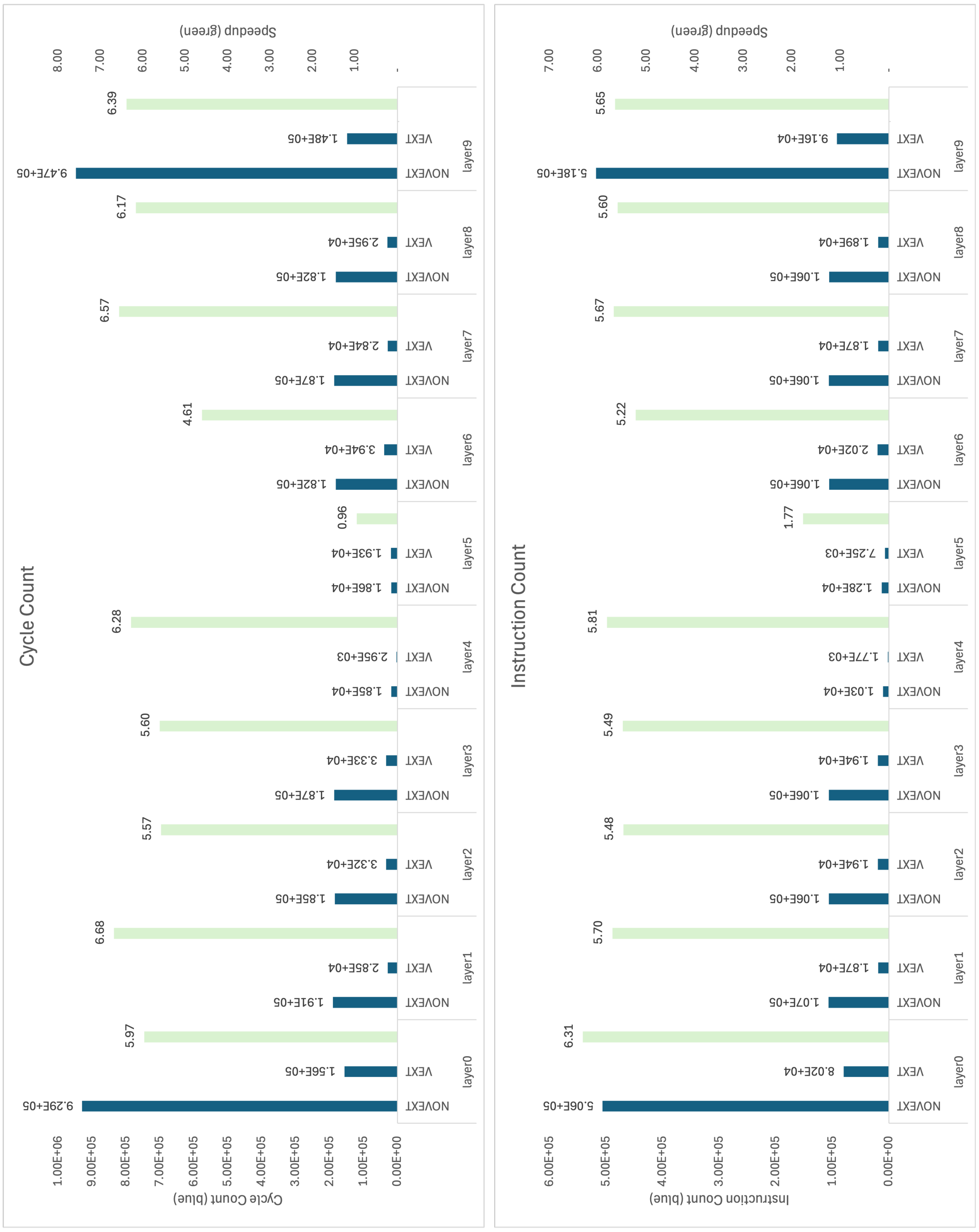

Figure A.1: Layer-wise benchmark results of the *toycar* model.

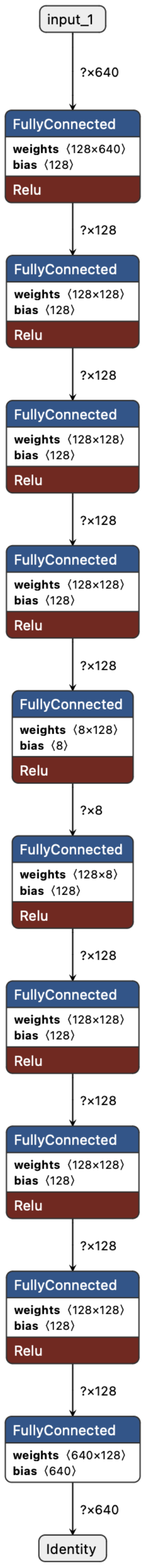

Figure A.2: *toycar* model architecture.

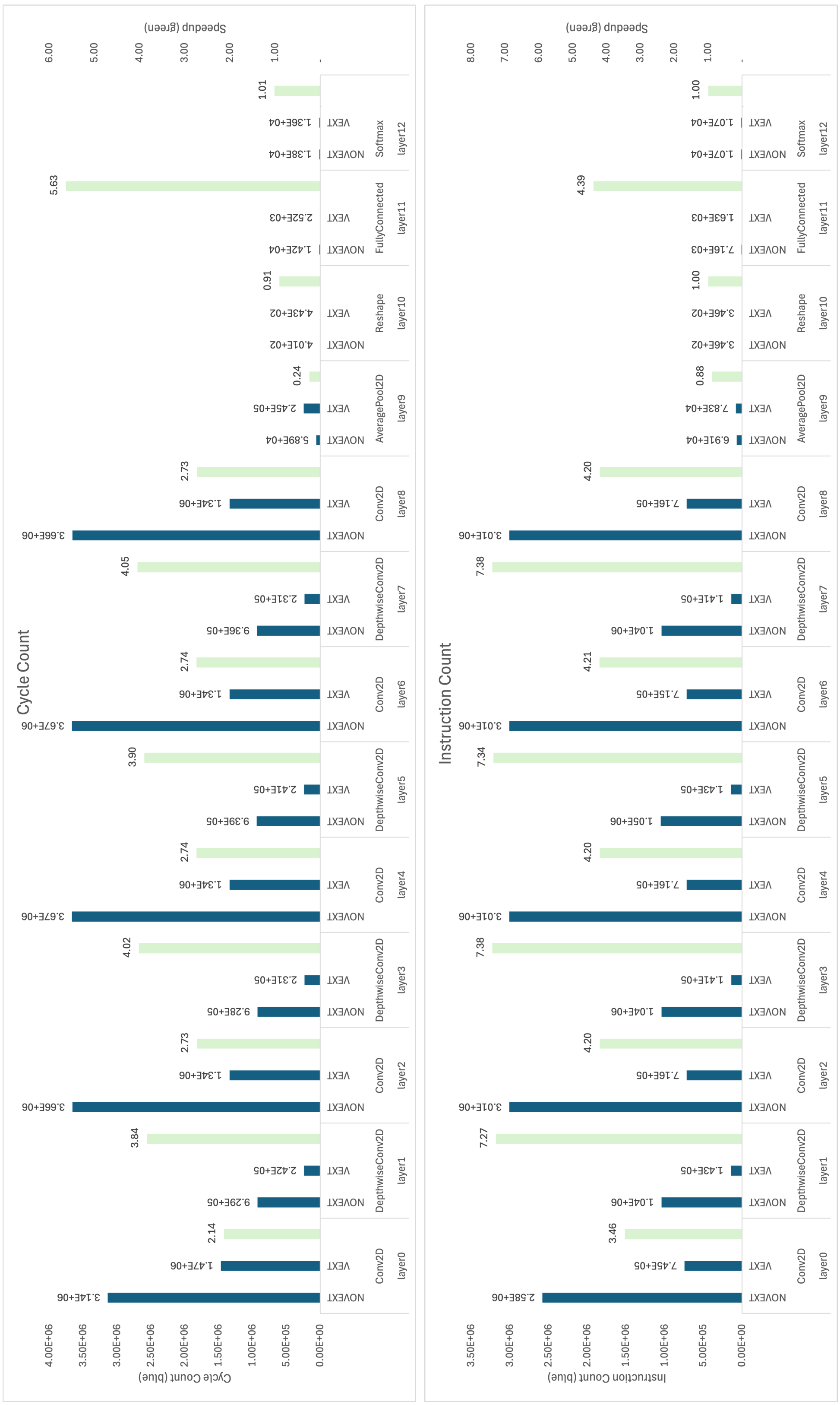

Figure A.3: Layer-wise benchmark results of the *aww* model.

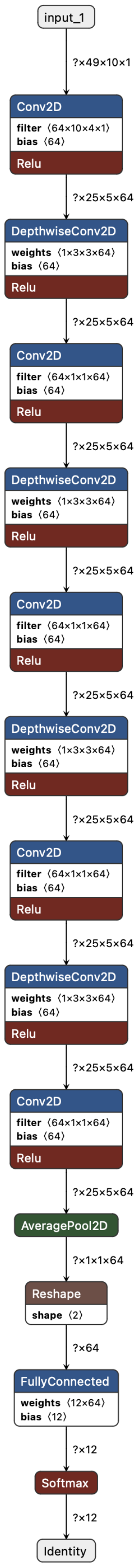

Figure A.4: *aww* model architecture.

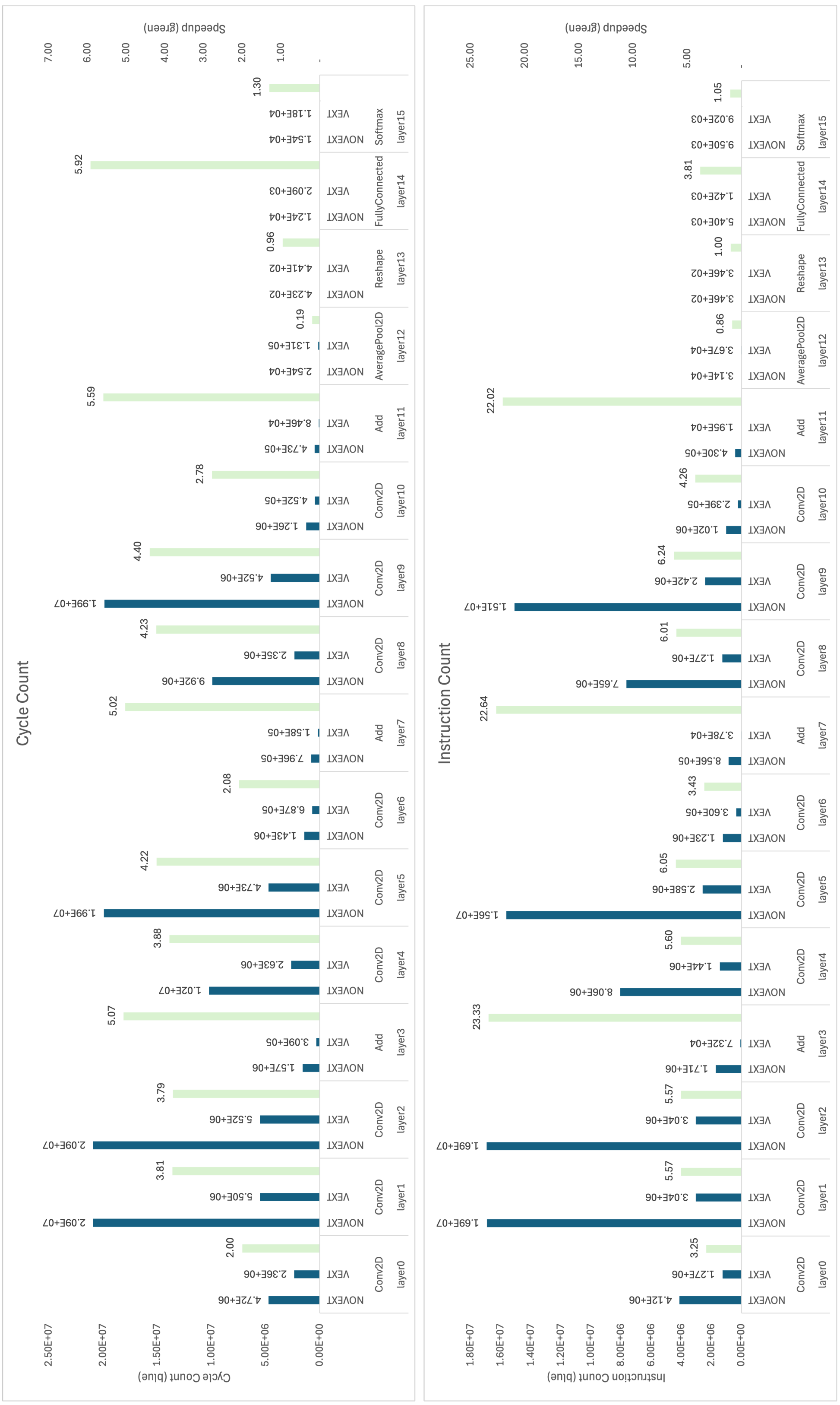

Figure A.5: Layer-wise benchmark results of the *resnet* model.

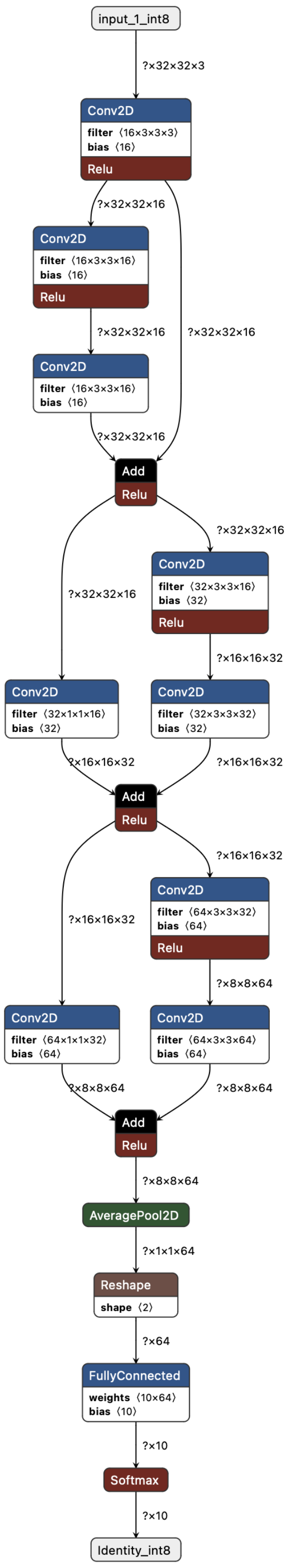

Figure A.6: *resnet* model architecture.

List of Figures

2.1 CanMV K230 block diagram from `https://github.com/kendryte/k230_docs/blob/main/en/CanMV_K230_Tutorial.md`. 6

4.1 Cycle and instruction counts for *aww, ic, toycar,* and *vww* benchmarks with vectorization enabled and disables. The instruction count on the CanMV K230 hardware platform is also compared to the results on the ISS. 13

A.1 Layer-wise benchmark results of the *toycar* model. 18
A.2 *toycar* model architecture. 19
A.3 Layer-wise benchmark results of the *aww* model. 20
A.4 *aww* model architecture. 21
A.5 Layer-wise benchmark results of the *resnet* model. 22
A.6 *resnet* model architecture. 23

List of Tables

4.1 Cycle and instruction count reductions for the four different benchmarks as well as the CPIs comparing vector to scalar execution. 13

Bibliography

[1] K230_docs/en/00_hardware/K230_datasheet.md at main · kendryte/k230_docs. https://github.com/kendryte/k230_docs/blob/main/en/00_hardware/K230_datasheet.md.

[2] K230_docs/en/01_software/board/K230_SDK_User_Manual.md at main · kendryte/k230_docs. https://github.com/kendryte/k230_docs/blob/main/en/01_software/board/K230_SDK_User_Manual.md.

[3] K230_docs/en/02_applications/tutorials/K230_RVV_In_Action.md at main · kendryte/k230_docs. https://github.com/kendryte/k230_docs/blob/main/en/02_applications/tutorials/K230_RVV_In_Action.md.

[4] Ratified Extensions - Home - RISC-V Tech Hub. https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154732/Ratified+Extensions.

[5] Release Vector Extension 1.0, frozen for public review · riscv/riscv-v-spec. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[6] Remlab: Installing Debian on the K230-CanMV. https://www.remlab.net/op/k230-canmv-debian.shtml.

[7] Kendryte/k230_docs. Kendryte, September 2024.

[8] Neil Adit and Adrian Sampson. Performance left on the table: An evaluation of compiler autovectorization for risc-v. *IEEE Micro*, 42(5):41–48, September 2022.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[10] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. Tensorflow lite micro: Embedded machine learning for tinyml systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[11] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR*, abs/1801.06601, 2018.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[13] Joseph K. L. Lee, Maurice Jamieson, Nick Brown, and Ricardo Jesus. Test-driving risc-v vector hardware for hpc. In Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse, editors, *High Performance Computing*, pages 419–432, Cham, 2023. Springer Nature Switzerland.

[14] RISC-V. Community News. XuanTie C908: High-performance RISC-V Processor Catered to AIoT Industry | Chang Liu, Alibaba Cloud – RISC-V International.

[15] Philipp van Kempen, Jefferson Parker Jones, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 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*, CF '24 Companion, page 75–78, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Philipp van Kempen, Rafael Stahl, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Mlonmcu: Tinyml benchmarking with fast retargeting. In *Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI*, CODAI '23, page 32–36, New York, NY, USA, 2024. Association for Computing Machinery.