Optimizing TinyEngine for the RISC-V Vector Extension

Allen Jason A. Tan*, Sherry Joy Alvionne S. Baquiran, Anastacia B. Alvarez
Electrical and Electronics Engineering Institute, University of the Philippines, Diliman
Email: allen.jason.tan@eee.upd.edu.ph, alvionne.baquiran@eee.upd.edu.ph, anastacia.alvarez@eee.upd.edu.ph

***Abstract*—Embedded edge devices face strict energy and size constraints for IoT and WSN applications. Because of this, advances have been made to enable state-of-the-art edge neural network models, such as MobileNetv2, to decrease their peak memory requirements to tens to hundreds of kilobytes using network architecture search tools and highly optimized libraries. However, optimization work in this space has been limited to a specific set of microcontroller cores including the recently developed embedded RISC-V processors that can take advantage of the highly parallel nature of machine learning workloads. This work explores the optimization of the TinyEngine inference library for the T908 vector processor as a reference for embedded vector processor designs. The vectorization achieved a 2.83x speedup relative to a scalar portable version of TinyEngine running on the same core, without a corresponding fall in average accuracy. In addition, replacing floating-point quantization with fixed-point operations is shown to have only a minor 3-4% reduction in Top 5 accuracy with negligible impact on performance, indicating that integer-only vector implementations can also be viable for embedded machine learning workloads.**

***Index Terms*—RISC-V, vector, TinyEngine, edge computing**

I. Introduction

Advances in embedded computing have led to the rise of Wireless Sensor Networks (WSN) and the Internet of Things (IoT). As these device applications scale into larger and more interconnected networks dealing with more complex data, it becomes more difficult to deal with the volume of data required. Tasks such as image recognition and visual wake words that are used to detect objects are now being offloaded onto individual nodes. This has led to the rise of the field of Tiny Machine Learning (TinyML) [1], which focuses on making these complex tasks work on hardware with strict performance and energy limits, which in turn limit their computational power and memory capacity.

Despite the complexity of state-of-the-art neural network (NN) models for applications such as image recognition, these tasks have been shown to be viable in microcontrollers [2]. Since TinyML generally works with lower quality data due to resource constraints, a viable approach to the problem has been to reduce the complexity of the problem by trading input size and model accuracy for a smaller and less compute-intensive model. This is further improved by using a Network Architecture Search (NAS) to find the optimal network architecture given a specific set of resource constraints [3]–[5]. With these trade-offs, it is now possible to fit an image recognition model targeting the ImageNet-1k dataset within 256kB of SRAM, placing it well within the memory budget of commercial microcontrollers [6].

With NN models now reduced in scope and scale, optimizing them for the target hardware is now increasingly important. For example, performance improves by compiling models directly for the hardware instead of relying on an interpreter layer. It is worth noting that, in MCUNet, most of the performance gains (ie. latency reduction) made against the baseline TF-Lite Micro framework can be explained by compiling with the Arm CMSIS-NN library [3].

Pushing the envelope even further in terms of both performance and energy efficiency means leveraging the massive increase in variety of specialized embedded processors. Hardware is informed by previous research on the characteristics of NN workloads, leading to novel embedded platforms with features ranging from multiple cores [7], [8], larger vector SIMD units [9], [10], and even dedicated accelerators [8]. These features lead to performance and efficiency gains above and beyond what is possible with the libraries optimized earlier [10] for more general-purpose microcontrollers.

In this work, we aim to add to this research by optimizing an existing optimized embedded machine learning library and determine what we can learn from it to inform future embedded vector processor design, to better match the demands of machine learning workloads.

II. Background and Related Work

A. Embedded Neural Network Libraries

Embedded NN libraries support less computationally complex NN layers to ensure that inference tasks fit in microcontroller-class hardware. For example, the open-source TensorFlow Lite Micro library supports commonly used operations such as convolutions and simpler merging, pooling, and activation layers [11]. On top of this, techniques such as quantization and patch-based inference reduce computational complexity and memory requirements while seeking to take advantage of the strengths of the hardware.

1) Quantization: Quantization involves replacing floating-point weights and activations with integers, resulting in smaller model sizes and memory requirements. This also allows smaller microcontrollers that only have integer ALUs to run much more efficiently than if these had to rely on expensive floating-point software subroutines. In return, accuracy is reduced due to less precise integers being used in place of floating-point operations. Considering that smaller models are required to fit on microcontrollers regardless, this is an acceptable trade-off.

TABLE I
ADD TABLE CAPTION.

Library	FANN-on-MCU [12]	TinyEngine [3]	TinyTS [5]	PULP-NN [13]
Target	Armv7-M/RISC-V	Armv7-M	Armv7-M	RISC-V
Quantization	8-bit	4/8-bit	8-bit	1/2/4/8-bit
Patch-Based Inference	No	Yes	Yes	No
SIMD Optimizations	Yes	Yes	Yes	Yes

Although memory requirements and computational power are reduced further by smaller quantization levels [13], support is limited by the target processor. For most libraries that target various Arm Cortex-M processors, the lower limit is 8-bit quantization. While more aggressive (e.g. 4-bit) quantization still results in smaller model and activation sizes, the lack of support for 4-bit operations incurs processing overheads from converting 4-bit weights and inputs to supported integer datatypes and back. There is a resulting upper bound on the memory savings from quantization, which is typically from a 32-bit floating point format to 8-bit integers (int8), or 75%.

2) Patch-Based Inference: To reduce memory requirements without shrinking models even further, particularly in terms of peak activation memory consumption, the earlier layers of NN models can be split into smaller chunks and evaluated independently. These smaller chunks require a smaller activation buffer, and their outputs can be joined before later layers, after which the NN model can operate normally. For example, MCUNetv2 [14] simply divides earlier layers into "patches", which are independent portions of the input image, resulting in an 8x reduction in peak memory for the MobileNet-V2 model, at the cost of 20% additional inference latency relative to MCUNetv1. Since patch generation is done at compile time, more memory bottlenecks can be resolved by adding additional levels of patches targeting later layers.

However, this results in a great deal of duplicated multiply-accumulate operations, due to overlaps present between each patch. By dedicating an additional buffer in memory to hold reused calculations, these overlaps can be resolved, resulting in a 60-80% speedup over MCUNetv2.

Moreover, tensors within layers can also be treated as a packing problem and dynamically divided and rearranged such that inference throughput is maximized within a set memory budget, which is accomplished through a virtual memory management system.

Although all of these approaches work to reduce memory, dividing layers presents an opportunity cost in terms of parallelization. As divisions become more granular, it becomes more difficult to amortize the overhead presented by function calls, loops, and other necessary control flow structures.

3) Microarchitectural Optimizations: State-of-the-art embedded NN libraries typically target the Arm Cortex-M family of cores [3], [5], [12], [15], as these are designed for embedded applications and are commonly found on microcontrollers. This core family is supported by the Arm CMSIS-NN library [16], providing a reasonable starting point for any further optimization work. In particular, the Cortex-M4 and Cortex-M7 cores tend to be used for published results. Both cores feature support for packed SIMD, which is used extensively in convolution kernels and underlying matrix multiplications. This allows for supporting quantized int8 operations and leveraging parallelism in NN workloads.

At the time of this work, there is a lack of work dealing with NN libraries targeting Cortex-M cores that support the Armv8-M ISA. This is a missed opportunity for applications that can now benefit from the M-Profile Vector Extension, which features wider SIMD than in the Cortex-M4 and Cortex-M7 [17].

There has also been work on optimizing NN inference libraries for other processors, mostly implementing various flavors of the RISC-V Instruction Set Architecture (ISA). These include processors that implement techniques such as multicore processing [12], [13] , which leverage the massively parallel nature of NNs. These resulting in [set number] performance uplift for NN kernels, which even demonstrate improved energy efficiency compared to Arm Cortex-M based devices. Prior exploration in this space is still limited, even though suitable embedded processors [10], [18] demonstrate significant performance and efficiency gains for NN kernel workloads. This highlights the potential for more performance gains by taking advantage of parallelism through more specialized processor design.

B. SIMD/Vector Computing in RISC-V

As a free, open, and modular ISA, RISC-V has been used to develop embedded processors specifically to leverage the parallel nature of NN workloads. SIMD development on embedded RISC-V processors can be broadly split into two: those implementing custom SIMD extensions, and RISC-V Vector Extension (RVV) implementations.

1) SIMD on RISC-V: Unlike Arm v6-M and later versions of the Arm M-Profile ISA, there is no ratified packed SIMD extension, although the P extension specifications have been in development since 2019 [19]. As such, packed SIMD efforts take the form of custom extensions [7] which have also been targets for RISC-V NN libraries [12], [13].

It is worth noting that, because of the freedom afforded by the RISC-V ISA, various SIMD architecture designs have been subjects of exploration, which include packed SIMD, but also include multicore processing managed by a controller core [7], [8], automatic control flow management [7], [8], and streaming co-processors [18].

All of these additions serve to leverage neural network parallelism while attempting to free up memory bandwidth by reducing the instructions to fetch, but two bottlenecks exist: the width of scalar registers, limiting how many elements can be processed per instruction, and the 32-bit instruction width, which limits how complex operations can be. This leads quite naturally to the evolution of embedded multicore clusters closer to vector processor cores, such as by forcing cores to execute the same instructions at the same time to minimize instruction fetches [20].

2) The RISC-V Vector Extension: The RISC-V Vector Extension (RVV) is a family of vector extensions that add vector SIMD and architectural vector registers to the RISC-V ISA. It features a configurable vector programming model, where the number and datatype of vector elements can be controlled independently of the vector instructions themselves. RVV also allows up to 8 vector registers to be grouped together, expanding vectors past vector register width alone [19].

The minimal subset of RVV is Zve32x, which only supports integer operations with an element width of up to 32 bits. The minimum vector register length and ALU length can both match the scalar register width (32 bits), but vector register groups still allow for up to 8x less instructions to be executed than using the scalar ISA, allowing for truly minimal vector implementations [21] to demonstrate significant speedup compared to purely scalar cores, even when hardware reuse is maximized.

Among other existing RVV implementations, Spatz [10] is explicitly aimed at embedded applications. Vector chaining between its three vector units (load/store, arithmetic, and permutation) is used to achieve close to optimal performance on NN kernels through interleaved instructions.

A later version increased its throughput by adding 256-bit vector registers and four floating-point units per core [22]. Each floating-point unit is nearly as large as the vector register file itself, on top of floating-point operations being more expensive than memory accesses. The cost of floating-point support highlights an opportunity for smaller, more efficient vector cores if floating-point requirements can be removed entirely.

III. TinyEngine Vectorization

For this work, three reference MCUNet models (MBV2-w0.35, MCUNet-10fps, MCUNet-512kB) [6] are used to profile scalar and vectorized library kernel functions. These were chosen to cover every layer used in inference across all pretrained MCUNet models. Patch-based inference is not evaluated for the reasons outlined in the previous section.

A. Target Platform

The CanMV-K230 board is chosen as the target platform, which has two onboard T-Head C908 CPU cores. In particular, this work targets Core 2, which supports the full RISC-V Vector Extension v1.0 spec. While this is not a microcontroller class processor, it has the same vector register width (128 bits) Arm MVE implementations [17]. More notably, because of its split 64-bit vector execution units which can chain two dependent operations together [23], its vector processing is actually in line with dual-beat MVE implementations.

B. Scalar Reference Implementation

Since the provided GCC version in the K230 SDK does not support the API, Clang-19 is instead chosen as the compiler. All scalar-only and vectorized versions of the port are built with the -O3 flag. Scalar-only versions are built without the vector extension in the target architecture.

The scalar reference was initially developed using an STM32F746G-based microcontroller board. The internal implementation of NN layer functions was kept as close to one-to-one as possible in an effort to match model functionality and accuracy and keep them comparable to the original MCUNet results. However, model accuracy was observed to be affected mostly by quantization and rounding behavior, which freed other portable code up to being optimized, as shown in the 1x1 convolution function example below, with an input size of 12x12x64.

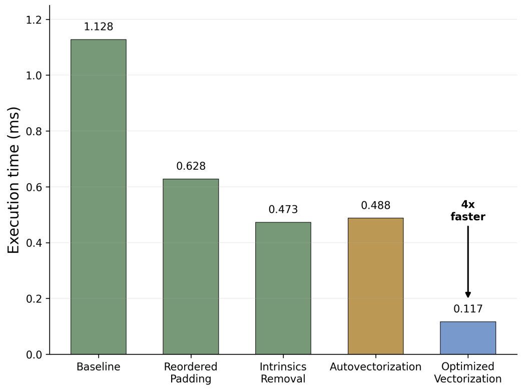

Fig. 1. Pointwise Convolution Function Speedup

In the original TinyEngine implementation, sign-extensions were interlaced due to the use of the SXTB16 instruction, resulting in interleaved padded inputs. These are first replaced with non-interleaved padding, and then followed up by replacing all intrinsics with portable C code, resulting in an around 2.4x speedup, which is then used as the new scalar baseline.

Vector support was then enabled to leverage autovectorization, to evaluate whether scalar code is accelerated. However, the scalar reference is not particularly amenable to brute-force vectorization due to the limited number of channels per loop iteration (4). As a result, the autovectorized function actually fares slightly worse, as it cannot yet fully take advantage of both software and hardware: the underlying organization of activations in memory and of the configurability of RVV.

C. Kernel Vectorization

Kernels used in the MCUNet model repository broadly use four categories of layers: pointwise convolution, depthwise convolution, average pooling, and add layers to accomplish skip connections. Of these, both types of convolution layers make the up the most of the layers and most of the execution time in the scalar references. The internal memory organization for TinyEngine uses an HWC (height-width-channel) format for both activations and kernels. This makes pointwise convolutions, especially those dealing with 1x1 kernels, relatively trivial to vectorize.

Each loop iteration processes the lesser between the maximum vector length or the number of remaining channels. This way, only one loop structure is needed to process all channels. Only one scalar step is needed: reducing the vector elements down to a scalar sum.

By the end of the loop, *v_sum* contains partial sums across all of its elements, so a sum reduction is needed to get the final sum. Because of how reduction operations work, it is impractical to vectorize the steps afterwards without using an additional buffer to hold all the sums.

This process is the common baseline for all vectorized kernels. More bespoke optimization is done for other kernels.

1) Segmented Load-Store Operations: On RVV, segmented loads allow a single load operation to split its results over a group of 2-8 registers. This enables vectorization of 2x2 and larger kernels, provided that the address offset is low enough, such as for RGB pixel inputs for all model inputs.

2) Slide Operations: For depthwise convolutions, using vector slides to effectively move the convolution window allows for great operand reuse. When combined with segmented loads, this allows for continuous multiply-accumulate operations on the limited kernel strides (1 or 2) that TinyEngine kernels do work with. In practice, however, operand reuse for depthwise layers is limited by the size of the kernel without setting aside more buffers for partial sums.

3) Full Integer-Only Quantization: TinyEngine has one layer used in all models that requires floating-point support, even when the floating-point requantization build option is turned off. The add layer (add_fpreq) uses FP32 scaling and offsets, and does not have a corresponding integer-only counterpart in the library.

RVV has vectorized fixed-point integer operations to enable quantization, the output of which saturates rather than overflows. Converting the add layer to use these fixed-point operations would remove the need for a floating-point unit, thus allowing smaller vector implementations to run TinyEngine.

IV. Vectorized Model Performance

Individual model inference latency and accuracy are shown in Table II. Note that the baseline results do not have comparable inference runtime figures since these are taken from runs using STM32F7 microcontrollers.

The models all received significant performance gains from optimized vector code, with a geometric mean of 2.83x speedup relative to their respective scalar reference versions without significant changes in model accuracy. Meanwhile, relying on compiler autovectorization did not result in any major changes in performance, despite the generation and use of vector instructions within NN kernel functions' inner loops.

The replacement of the last floating-point scaling in TinyEngine with equivalent 32-bit fixed-point operations resulted in a slight drop in accuracy, which is expected since this is essentially carrying out quantization that the NN models were not trained for. There is also an average speedup of 1%, but the small scale of this performance boost is partly due to add operations comprising a relatively small part of each inference.

In theory, since each vector unit in the C908 processor is 64 bits wide, the theoretical performance gain should be somewhere in the range of 4x, since that is the ratio of 16-bit elements that the vector processor can handle per cycle than a scalar RISC-V counterpart. However, this theoretical speedup was not met, which can be partly explained by the characteristics of the models selected and of the convolution layers.

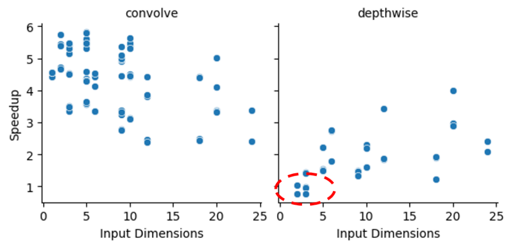

Fig. 2. Pointwise (left) and Depthwise (right) Speedup vs Input Dimensions

When looking at layer performance with respect to the input size (in pixels), there is clear performance degradation that happens to depthwise layers as the input size drops below 5 by 5. This is because of the internal CHW (channel-height-width) organization for depthwise kernels, which result in vector registers that are not filled up. Unfortunately, the C908 core does not reduce its vector latency when this happens [23], so the overhead of using the vector extension outweighs whatever additional SIMD processing can be done. For this reason, a rewrite of depthwise convolutions away from CHW for smaller input sizes is recommended. Alternatively, since this implementation relies heavily on slow vector slide operations for relatively small vector lengths, this could also be taken as a recommendation to focus on smaller, faster permutations in vector processor designs.

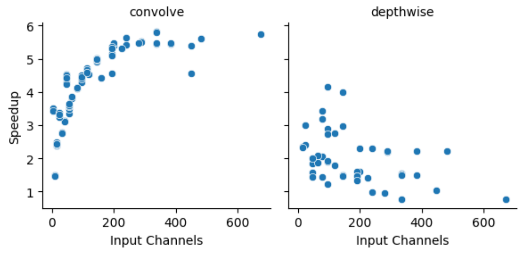

Fig. 3. Pointwise (left) and Depthwise (right) Speedup vs Input Channels

Pointwise convolution performance increases with respect to the number of input channels, but performance saturates after a certain point, which can be attributed to the kernels being starved for memory. Conversely, this also means that the first layer, which has to take on 3-channel RGB inputs, is also the most difficult to vectorize.

TABLE II
SCALAR AND VECTOR INFERENCE RESULTS

Model (input size)	MBv2-w0.35 (144px)	MCUNet-10fps (48px)	MCUNet-512kB (160px)
Target	Top5 Acc (%) / Time (ms)	Top5 Acc (%) / Time (ms)	Top5 Acc (%) / Time (ms)
Baseline [6]	73.8% / –	66.3% / –	88.4% / –
Scalar Reference	71.0% / 103.5	58.0% / 26.52	82.8 % / 283.7
Autovectorized Scalar Reference	71.7% / 103.5	58.3% / 26.24	82.3% / 284.4
Vectorized	71.4% / 42.12	60.3% / 9.146	80.9% / 88.38
Vectorized Fixed-Point	68.4% / 41.90	58.4% / 9.135	76.1% / 88.06

V. CONCLUSION

This work demonstrates that there are significant performance gains that can be unlocked through vectorization for embedded machine learning workloads, with a geometric mean of 2.83x relative to a scalar port to the same C908 core on the CanMV-K230 board. In particular, the RISC-V Vector Extension is shown to work well for this purpose, as features such as its dynamic configurability, segmented load operations and other features that enable greater operand reuse and efficiency. All this comes in spite of bottlenecks that come in some layers that would otherwise require an overhaul to get around. However, as shown by the relatively poor results from compiler autovectorization, vectorization alone is not a solution that will work independently of an understanding of the underlying models and kernels.

An important finding of this work is that floating-point requantization and scaling can be replaced with equivalent RVV fixed-point operations for inference without major degradation in model performance or accuracy. This shows that smaller, more efficient integer-only RISC-V Vector processors can be a viable option.

REFERENCES

[1] Tinyml foundation. [Online]. Available: https://www.tinyml.org/

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[3] J. Lin, W.-M. Chen, Y. Lin, C. Gan, and S. Han, "MCUNet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[4] F. Paissan, A. Ancilotto, and E. Farella, "Phinets: A scalable backbone for low-power ai at the edge," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 5, 2022. [Online]. Available: https://doi.org/10.1145/3510832

[5] Y.-Y. Liu, H.-S. Zheng, Y. Fang Hu, C.-F. Hsu, and T. T. Yeh, "Tinyts: Memory-efficient tinyml model compiler framework on microcontrollers," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 848–860.

[6] MIT HAN Lab, "MCUNet: Tiny Deep Learning on IoT Devices." [Online]. Available: https://github.com/mit-han-lab/mcunet

[7] A. Pullini, D. Rossi, I. Loi, A. Di Mauro, and L. Benini, "Mr. wolf: A 1 gflop/s energy-proportional parallel ultra low power soc for iot edge processing," in *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, 2018, pp. 274–277.

[8] D. Rossi, F. Conti, M. Eggiman, A. D. Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, 2022.

[9] Arm Corporation, "Arm Cortex-M55 Processor Technical Reference Manual." [Online]. Available: https://developer.arm.com/documentation/101051/0101

[10] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, "Spatz: A compact vector processing unit for high-performance and energy-efficient shared-l1 clusters," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.

[11] TensorFlow, "TensorFlow Lite for Microcontrollers." [Online]. Available: https://github.com/tensorflow/tflite-micro

[12] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4403–4417, 2020.

[13] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: A computing library for quantized neural network inference at the edge on risc-v based parallel ultra low power clusters," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 33–36.

[14] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 2346–2358.

[15] H.-S. Zheng, Y.-Y. Liu, C.-F. Hsu, and T. T. Yeh, "Streamnet: Memory-efficient streaming tiny deep learning inference on the microcontroller," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 37 160–37 172.

[16] Arm Corporation, "CMSIS NN." [Online]. Available: https://github.com/ARM-software/CMSIS-NN

[17] ——, "Armv8-M Architecture Reference Manual." [Online]. Available: http://developer.arm.com/documentation/ddi0553/latest/

[18] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1845–1860, 2021.

[19] RISC-V Foundation, "The RISC-V Instruction Set Manual Volume I." [Online]. Available: https://github.com/riscv/riscv-isa-manual

[20] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, A. D. Mauro, L. Benini, and D. Rossi, "Dustin: A 16-cores parallel ultra-low-power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 6, pp. 2450–2463, 2023.

[21] M. Johns and T. J. Kazmierski, "A minimal risc-v vector processor for embedded systems," in *2020 Forum for Specification and Design Languages (FDL)*, 2020, pp. 1–4.

[22] M. Perotti, S. Riedel, M. Cavalcante, and L. Benini, "Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 7, pp. 2488–2502, 2025.

[23] Olaf Bernstein, "RVV benchmark XuanTie C908." [Online]. Available: https://camel-cdr.github.io/rvv-bench-results/canmv$_k230/index.html$