Accelerating Machine Learning with RISC-V Vector Extension and Auto-Vectorization Techniques

Willian Analdo Nunes and Fernando Gehm Moraes
School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazill
willian.nunes@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Convolutional neural networks (CNNs) have played a significant role in the recent evolution of machine learning (ML) due to their feature extraction and pattern recognition capabilities. CNNs involve numerous Multiply and Accumulate (MAC) operations, which are computationally expensive and often require hardware acceleration to achieve acceptable performance. The RISC-V Vector (RVV) Extension is a candidate for accelerating vector processing operations, such as those found in CNNs. Several studies have explored the application of the RVV extension in various areas of ML, where it is often implemented as a co-processor due to its complexity. This work presents an RVV implementation as a tightly coupled accelerator for a small RISC-V processor, RS5, which implements a subset of the RVV extension to maintain a non-prohibitive area overhead. The paper explores a case study of a 1-D CNN combined with the recently introduced auto-vectorization feature in GCC 14.1. Performance results show an average speedup of 1.88x compared to a scalar core, achieved without modifying the original code.**

***Index Terms*—RISC-V, Vector Extension, CNN, Machine Learning.**

I. Introduction and Related Work

The RISC-V Vector (RVV) Extension [1] enhances the RISC-V ISA [2] by adding vector processing capabilities. It aims to deliver efficient and scalable data parallelism, which is advantageous for large-scale data processing applications, including scientific computing, Machine Learning (ML) multimedia processing, and other high-performance computing tasks.

Table I compares RISC-V vector accelerators, highlighting integration type (co-processor or tightly coupled unit) and target applications. Our work differs by focusing on tightly coupled acceleration for embedded systems.

Hwacha [3], a precursor to RISC-V vector processing, uses a custom ISA extension for parallel data processing that inspired the RVV extension. As a **co-processor** for the Rocket Core, Hwacha supports multi-lane execution, maximizing vector processing efficiency.

ARA is a 64-bit vector processor serving as a **co-processor** for CVA-6 (Ariane), supporting RVV v0.5 [4] and v1.0 [5]. It offers multi-lane execution, targeting high efficiency. Yun [6] investigates ARA's performance in an SoC design using matrix-multiplication kernels.

Arrow [7], a dual-lane vector **co-processor**, implements a subset of the RVV v0.9 extension, designed mainly for edge ML inference. It features 256-bit registers (VLEN) and 64-bit words (ELEN).

Ali et al. [8] introduce a Vector Processing Unit (VPU) integrated with the CV32E40P core as a **co-processor** for embedded systems. The VPU supports parameterized VLEN and lane configurations, achieving a 2.3x reduction in cycles per instruction for MAC operations compared to the CV32E40P.

SiFive offers two commercial processors with the RVV extension, serving as a **co-processor**, aimed at machine learning and high speed: the SiFive Intelligence X280 [9] and the SiFive Performance P270 [10].

SPEED [11] is an RVV processor optimized for multi-precision inference in deep neural networks (DNNs), featuring customized instructions, hardware design, and dataflow mapping for efficiency.

Johns et al. [12] introduce a RISC-V vector processor for microcontrollers and embedded systems. This processor is an in-order, single-stage RV32E, modified with a tightly integrated Vector Processing Unit (VPU) that supports a subset of the vector extension.

While most works treat the RVV extension as a co-processor, our approach couples it with the processor to enhance resource sharing and minimize protocol communication latency and overhead. Unlike other works focusing on performance through complex processors (except [12]), we aim to boost performance while minimizing silicon area, making our solution suitable for embedded systems.

This paper presents a *parameterizable implementation of a circuit* for the RVV extension in a low-complexity RISC-V core, RS5 [13], an in-order execution core with a 4-stage pipeline and no floating-point unit. The proposed circuit, integrated with the auto-vectorization capabilities of GCC 14.1 [14], efficiently exploits data-level parallelism in ML applications, such as CNNs. By using auto-vectorization, the design accelerates the inference process of CNNs without requiring manual code modifications.

II. Introduction to the RISC-V Vector Extension

The RVV extension [1] adds 32 vector registers (v0-v31) and seven Control and Status Registers (CSRs). Each vector register has a fixed length, known as **VLEN**, which must be a power of 2. The Element Length (**ELEN**) parameter, also a power of 2, defines the maximum size of a vector element that any operation can produce or consume. The selected element width (**SEW**) is defined in runtime and ranges from 8-bit elements up to ELEN-bit elements. This feature enables the hardware to support various operations and precision levels.

The *vector operands* or *results* may occupy one or more vector registers, configuring a register group. A register group is configured using a CSR field called "LMUL". Another CSR field is the "VL", which stands for Vector Length. It specifies the number of elements the instruction will process in the iteration and depends on the SEW and LMUL values. A configuration instruction (*vsetvl*) is provided to rapidly configure the vector CSRs to match application needs. It

TABLE I
COMPARISON BETWEEN RISC-V VECTOR ACCELERATORS.

Proposal	Scalar Core	Coupling	RVV Version	Pipelined	Target Application	Language
Arrow [7]	MicroBlaze v11.0 (not RV)	Co-Processor	v0.9 subset	✓	Edge ML	VHDL
Hwacha [3]	Rocket core	Co-Processor	Custom	✓	General Purpose	Chisel
Ara [4, 5]	CVA6 (ARIANE)	Co-Processor	v0.5 and v1.0	✓	General Purpose	SystemVerilog
Ali et al. [8]	cv32e40p	Co-Processor	v0.1	✓	Embedded systems	SystemVerilog
SiFive Intelligence X280 [9]	SiFive Intelligence X280	Co-Processor	v1.0 + custom	✓	AI/ML compute at the edge	N/S
SiFive Performance P270 [10]	SiFive Performance P270	Co-Processor	v1.0 + custom	✓	Maximum throughput	N/S
SPEED [11]	-	Coupled	v1.0 + custom	✓	Multi-Precision DNN Inference	N/S
Johns et al. [12]	Custom RV32E	Coupled	v0.9 subset	X	Embedded systems	N/S
This work	RS5 [13]	Coupled	v1.0 subset	✓	Embedded systems	SystemVerilog

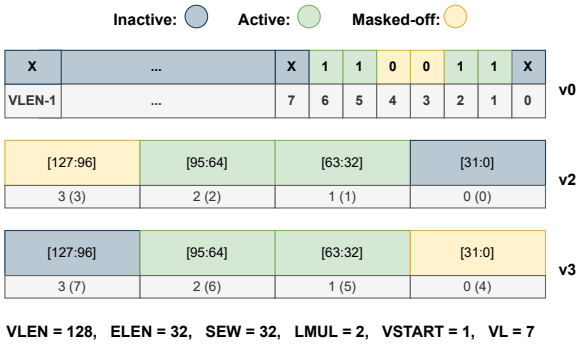

Fig. 1. Vector registers during operation. Position indexes are represented under the register with parentheses. The "X" values represent don't-cares.

changes the SEW and VL, among other configurations, such as mask policy.

Masking is supported on many vector instructions. The mask value is used to control the execution of a masked vector instruction and is always supplied by the vector register *v0*. Each bit of *v0* represents an element in the register group.

Figure 1 illustrates the use of two vector registers in a masked operation. The VLEN is 128 bits, and the vector registers *v2* and *v3* are configured with *LMUL=2*. The mask is provided by vector *v0*, and the Scalar Element Width (SEW) is 32 bits, allowing each register to hold four elements (VLEN/SEW). The *vstart=1* setting in the CSR specifies that processing begins at index 1, leaving the element at index 0 of *v2* untouched. With *VL=7*, the vector contains seven elements, but processing occurs from index 1 to 6. Index 7 is not processed, and elements at indices 3 and 4 are masked off, leaving the remaining elements as active for generating results.

Load and store instructions support three memory addressing modes: (*i*) *unit-stride*, where elements are accessed sequentially in memory starting from the base effective address; (*ii*) *strided*, where elements are accessed with a constant stride, beginning at the base address and incrementing by a byte offset specified in a scalar register; and (*iii*) *indexed*, where vector-indexed operations calculate each element's effective address by adding the contents of a vector register (containing offset values) to the base address.

III. VECTOR EXTENSION IMPLEMENTATION

The vector extension is implemented in the RS5 [13] processor as a **tightly coupled accelerator**. The vector instructions are decoded in the RS5 decoding unit, although it uses a different control signal for the instructions to maintain easy decoupling.

The RS5 processor includes multiple execution units, which are specialized modules for tasks like load-store and multiplication, integrated within the execute unit at the third pipeline stage. Some units require several cycles to complete an instruction; for example, multiplication takes 3 to 4 cycles, and division may take up to 33 cycles. During multi-cycle instruction execution, the pipeline stalls by asserting a *hold* signal until the operation is complete.

The vector unit is implemented as a multi-cycle execution unit within the execute module. As shown in Figure 2, the vector unit features a three-stage pipeline. The first stage includes control logic for vector CSRs, along with operand fetching and replication. The second stage handles execution, processing vectors using two units: the Vector Load and Store Unit (VLSU) and the Vector Arithmetic and Logic Unit (VALU). It also generates the write enable signal for the Vector Register File based on the mask and vector length (VL). In the final write-back stage, the pipeline selects between the VALU and VLSU results according to the instruction. It then performs the write-back to the Vector Register File (VRF) using the write enable and destination register signals.

The vector unit employs a Finite State Machine (FSM) to control its pipeline, operating in two states: *V_IDLE* and *V_EXEC*. The *V_IDLE* state is the initial state, where the unit remains when it is not operational (i.e., not executing a vector instruction). Upon detecting a vector instruction, it transitions to the *V_EXEC* state, raising the *hold_o* signal and propagating the operands to the second pipeline stage. In the *V_EXEC* state, computations are carried out. This state persists until all registers in the current group (LMUL of the instruction, up to 8 registers) are processed. The *hold_o* signal is de-asserted during the final execution cycle, allowing the scalar pipeline to resume operation. A data hazard detection mechanism is implemented to prevent pipeline stalls due to write-back conflicts. This mechanism is triggered when a vector instruction enters the first pipeline stage with a source register that matches the destination register in the last pipeline stage. In such cases, the pipeline is stalled for one cycle to ensure that results are properly written to the VRF.

The numbers in Figure 2 correspond to the elements that implement the RVV extension:

1) **Scalar And Immediate Replication.** Controls the replication of the scalar and immediate operands and the operands selection via multiplexers.
2) **Vector CSRs.** Implements the vector CSRs and associated logic.
3) **Control FSM.** Controls the vector unit.

   3.1) LMUL Control. Manages the register groups by

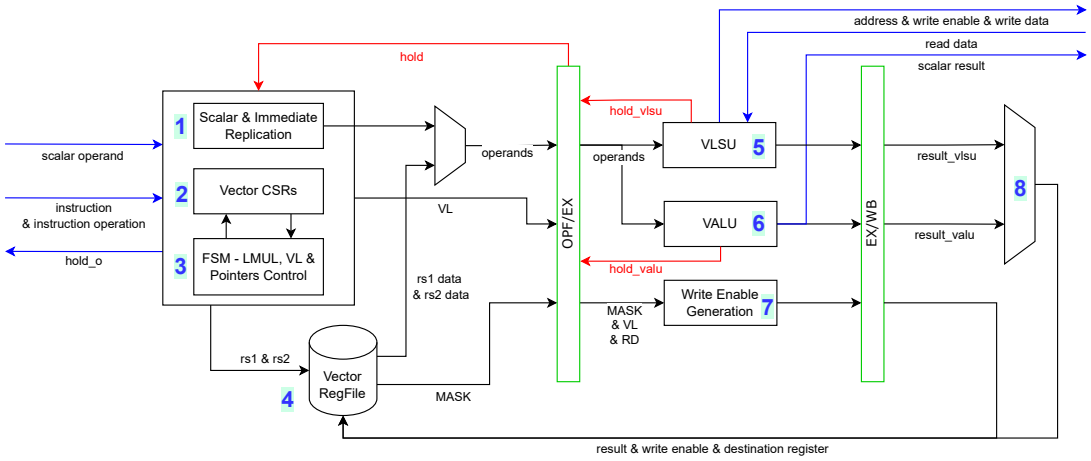

Fig. 2. Vector Unit Organization. Green components are temporal barriers. Signals that control the unit pipeline stall are marked in red. Blue signals are the communication between the scalar core.

tracking the number of registers that have been processed within a group. It is used to determine when the execution of an instruction is complete.

3.2) Register Pointers Control. Register groups are addressed by their smallest number, requiring adjustments to the operand read address. The base register address combines with cycle counters to obtain the correct address.

3.3) Vector Length Control. Manages the number of elements a register can hold and monitors how many have been processed during instruction execution.

4) Vector Register File (VRF). Implements the vector registers. The VRF has one write port and three read ports: first and second operands and mask (V0 register).

5) Vector Load-Store Unit (VLSU). It manages the vector memory access. Each vector access is multi-cyle, and the latency depends on the register length, address mode, SEW, and address alignment.

6) Vector Arithmetic and Logic Unit (VALU). Handles vector arithmetic and logic operations, processing an entire VLEN register in parallel for any element width up to ELEN. The number of cycles required varies based on SEW, with wider operations taking more cycles. For instance, additions complete in one cycle per register, while multiplications can take up to 4 cycles.

7) Write Enable Generation. Generation of the register file's write enable signal. It is a byte-enable signal that considers SEW, VL, and mask.

8) Result Control and Demultiplexing. Demultiplex the result of the VALU and VLSU and send to write back in the VRF or to the scalar register file

The VLEN is a design parameter. It can be changed at the module instantiation and is flexible in powers of 2. When VLEN is changed, the entire vector unit adapts to handle VLEN-length data.

The vector unit implements a subset of the RVV extension, selected specifically for machine learning applications. These instructions are categorized into classes, as outlined in Table II, which also indicates the processing cycles required for each Scalar Element Width (SEW) and instruction class. The final column, **optional**, identifies classes that can be disabled via design-time code parameters to further minimize the implemented instruction subset.

TABLE II
INSTRUCTIONS OF THE RS5 VECTOR UNIT AND THE CYCLES PER REGISTER FOR DIFFERENT SEW CONFIGURATIONS.

Instruction Class	Instruction Names	SEW=8 Cycles/Reg	SEW=16 Cycles/Reg	SEW=32 Cycles/Reg	Optional
Arithmetic	VADD, VSUB, VRSUB	1	1	1	N
Logic	VAND, VOR, VXOR	1	1	1	Y
Shifts	VSLL, VSRL, VSRA	1	1	1	N
Mask Compares	VMSEQ, VMSNE, VMSLTU, VMSLT, VMSLEU, VMSLE, VMSGTU, VMSGT	1	1	1	N
Min/Max	VMIN, VMINU, VMAX, VMAXU	1	1	1	Y
Multiplication	VMUL, VMULH, VMULHU, VMULHSU	1	1	3-4	N
Widening Multiplication	VWMUL, VWMULU, VWMULSU	2	2	4-5	N
Multiply and Accumulate	VMACC, VNMSAC, VMADD, VNMSUB	2	2	4-5	N
Division	VDIV, VDIVU, VREM, VREMU	1-8	1-16	1-32	Y
Sum Reduction	VREDSUM	1	1	1	Y
Min/Max Reduction	VREDMIN, VREDMINU, VREDMAX, VREDMAXU	1	1	1	Y
Logic Reduction	VREDAND, VREDOR, VREDXOR	1	1	1	Y
Register Moves	VMV, VMVR, VMVSX, VMVXS	1	1	1	N
Unit-Strided Load/Store	VLE, VSE	∼VLEN/32	∼VLEN/32	∼VLEN/32	N
Strided Load/Store	VSLE,VSSE	VLEN/8	VLEN/16	VLEN/32	N
Index load/Store	VLUXEI, VLOXEI, VSUXEI, VSOXEI	VLEN/8	VLEN/16	VLEN/32	N

Configuration instructions take one cycle to execute. They are used to set vector configurations such as VL, SEW, and LMUL.

IV. RESULTS

The validation application is a 1-dimensional CNN [15], converted from a PyTorch model to a C model using integer representation. It consists of three convolutional layers and two fully connected layers, with rectified linear activation (ReLU) applied after each convolution. The work in [16] extended this CNN by adding temporal awareness while maintaining low complexity. The model receives a unidimensional vector corresponding to the raw data extracted from the dataset as input. This raw data input is 1 × 120, *KERNEL_SIZE=5*, and the number of filters equals 64. In total, for a single inference, it is necessary to execute 5,428,096 MAC operations.

A. Auto-vectorization

The application was compiled for RISC-V using GCC 14.1 [14] and executed on the RS5 processor to generate baseline results. This compiler version introduced **automatic vectorization**, while the previous version, GCC 13, added support for RVV extension instructions through inline assembly. The compilation flags used for the RVV extension were *"rv32im_zve32x_zvl64b"*, where *zvl64b* specifies the minimum VLEN required to run the compiled code, ensuring compatibility across RVV platforms with different VLENs. To optimize performance, the code should be compiled for the target VLEN by adjusting the *zvl* flag and using the additional flag *-mrvv-vector-bits=zvl*.

We demonstrate auto-vectorization with the ReLU operation due to its simplicity, which effectively shows how the vector extension functions. Although the convolution and fully connected layers also benefit from auto-vectorization, explaining their code is beyond the paper's space limitations. The ReLU operation outputs the input if positive; otherwise, it outputs zero and is applied after each convolution layer on its results. In C, this involves a loop iterating over all vector outputs (convolution results), replacing negative values with zeros. This loop translates to the assembly code shown in Listing 1 when using the RVV extension (256-bit VLEN *-zvl256b*).

```
1  la  a4, <vector>      # Vector Pointer
2  li  a5, <vector_size> # Amount of elements
3
4  vsetivli zero,8,e32,m1,ta,ma # SEW=32b, VL=8, LMUL=1
5  vmv.v.i  v1,0  # Fill v1 register with zeros
6
7  vle32.v  v0,(a4) # Load v0 register with elements
8  addi a5,a5,-8  # Decrement the amount of remaining
       elements to be processed
9
10 vmsle.vi v0,v0,0 # Create a mask with negative
       elements set
11 vse32.v  v1,(a4),v0.t # Store zeros on elements with
        mask==1
12
13 bnez a5, <line 7>
```

Listing 1. Commented Assembly instructions generated for ReLU.

This section shows that GCC 14.1 effectively vectorizes the source code without requiring modifications and handles complex loops like convolution. While it has limitations, such as redundant configuration instructions and not fully utilizing register groups (e.g., LMUL=1), it significantly improves performance compared to scalar code.

B. Performance evaluation

Table III presents the performance results for executing the target application with different VLEN values and speedups. The baseline is the scalar RS5 processor, with the RV32IM ISA. A VLEN of 256 offered the best tradeoff between register size and performance improvement, achieving a speedup of 1.819, making it the optimal choice for the target application with 32-bit elements. VLEN 64 showed minimal performance gains and is therefore not recommended. VLENs larger than 256 exhibited a performance gain plateau.

The performance evaluation was conducted using 32-bit data. We also evaluated the same application with 16-bit data and observed at the 128-bit VLEN an acceleration similar to the 256-bit VLEN using 32-bit data. This result demonstrates a direct correlation between data width and VLEN. Thus, substantial performance gains can be obtained using CNNs trained with smaller bit-width integers, such as 8 or 16 bits.

TABLE III
PERFORMANCE RESULTS RUNNING THE TARGET APPLICATION FOR RS5 CORE WITH VECTOR UNIT.

VLEN	Number of Cycles	Cycles Reduction (%)	Speedup
-	759,217,940	-	-
64	748,079,170	1.467	1.015
128	560,439,300	26.182	1.354
256	417,339,740	45.030	**1.819**
512	408,593,050	46.182	1.857
1024	404,004,080	46.786	1.879
2048	401,730,790	47.086	1.889

C. Area Evaluation

Table IV shows synthesis results for different VLEN and subsets (28nm@500 MHz, Cadence Genus 22.15-s086_1, typical corner). The **minimum** subset includes mandatory instructions (Table II), while the minimum subset with *RedSum* instructions is required for CNN applications. The **maximum** subset contains all optional instructions.

TABLE IV
AREA RESULTS WITH THE VECTOR UNIT – 28NM@500MHZ.

Instruction Set	VLEN	Total # Gates	Total Area (μm^2)	Slack (ps) - Nominal
Baseline (No Vector)	-	8,804	12,997	397
Minimum - Plus RedSum	64	28,381	37,349	411
Maximum - No Div	64	31,714	40,916	376
Maximum	64	37,446	47,952	401
Minimum - Plus RedSum	128	44,305	57,651	434
Maximum - No Div	128	50,943	64,254	404
Maximum	128	62,990	79,365	410
Minimum - Plus RedSum	**256**	79,014	102,021	411
Maximum - No Div	256	92,177	115,263	405
Maximum	256	115,406	145,613	406

Results show that in the minimum subset, the area overhead varies from 2.87x (VLEN=64) to 7.85x (VLEN=256). Including division can be costly, adding from ≈7,000 (VLEN=64) to ≈30,000 (VLEN=256) μm^2 to the total area. All VLENs reached timing closure for all corners (*worst*, *typical*, *fastest*). The 1.819 speedup was obtained with the "minimum - plus RedSum" subset, with an area overhead of 7.85x.

Wang et al. [11] synthesized Ara and SPEED using Synopsys Design Compiler 2014.09 on the TSMC 28 nm process with four lanes and a vector length of 4,096 bits. At a frequency of 500 MHz and an RV64GCV1.0 ISA, they reported areas of 0.44 mm^2 for Ara and 1.10 mm^2 for SPEED. The largest RS5 RVV version requires 0.145 mm^2 under the same frequency and technology, corresponding to one-third the area of Ara and ≈7.5 times smaller than SPEED, despite being single-lane and having a smaller vector length.

V. CONCLUSION AND FUTURE WORK

This work presented an RVV extension implementation in an embeddable processor, adding better capabilities to handle ML tasks as proved by the test-case application. Speedups of up to 1.89x showed a significant improvement without any code modification, brought by the GCC 14.1 auto-vectorization, Future work comprises power evaluation of the vector unit, clock gating analysis, and the addition of CNNs with more complexity to the test-case applications.

ACKNOWLEDGMENTS

This work was financed in part by by CAPES (Finance Code 001), CNPq (grant 309605/2020), and FAPERGS (grants 21/2551-0002047-4 and 23/2551-0002200-1).

REFERENCES

[1] K. Asanović, "RISC-V "V" Vector Extension - Version 1.0," 2024, https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[2] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," Berkeley, Tech. Rep., 2024, https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link.

[3] C. Schmidt, A. Ou, and K. Asanović, "Hwacha v4: Decoupled data parallel custom extension," in *SUMMIT*, 2018, pp. 1–40, https://riscv.org/wp-content/uploads/2018/12/Hwacha-A-Data-Parallel-RISC-V-Extension-and-Implementation-Schmidt-Ou-.pdf.

[4] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019, https://doi.org/10.1109/TVLSI.2019.2950087.

[5] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design," in *ASAP*, 2022, pp. 43–51, https://doi.org/10.1109/ASAP54787.2022.00017.

[6] M. Perotti, M. Cavalcante, A. Ottaviano, J. Liu, and L. Benini, "Yun: An Open-Source, 64-Bit RISC-V-Based Vector Processor With Multi-Precision Integer and Floating-Point Support in 65-nm CMOS," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 10, pp. 3732–3736, 2023, https://doi.org/10.1109/TCSII.2023.3292579.

[7] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, and M. A. Saghir, "Arrow: A RISC-V vector accelerator for machine learning inference," *arXiv preprint arXiv:2107.07169*, 2021, https://arxiv.org/abs/2107.07169.

[8] M. Ali, M. von Ameln, and D. Goehringer, "Vector processing unit: A RISC-v based SIMD co-processor for Embedded Processing," in *DSD*, 2021, pp. 30–34, https://doi.org/10.1109/DSD53832.2021.00014.

[9] SiFive, Inc, "SiFive Intelligence X280 Data Sheet," 2023, https://sifive.cdn.prismic.io/sifive/9405d3d0-35a1-4680-a259-7a5598d1ecb2_sifive-intelligence-x200-datasheet.pdf.

[10] ——, "SiFive Performance P270 Data Sheet," 2022, https://sifive.cdn.prismic.io/sifive/859c28c0-8bd5-4fc4-9113-a25a2a89bf9c_P270+Data+Sheet.pdf.

[11] C. Wang, C. Fang, X. Wu, Z. Wang, and J. Lin, "A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference," in *ISCAS*, 2024, pp. 1–5, https://doi.org/10.1109/ISCAS58744.2024.10558028.

[12] M. Johns and T. J. Kazmierski, "A minimal RISC-V vector processor for embedded systems," in *FSL*, 2020, pp. 1–4, https://doi.org/10.1109/FDL50818.2020.9232940.

[13] W. A. Nunes, A. E. Dal Zotto, C. d. S. Borges, and F. G. Moraes, "RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems," in *LASCAS*, 2024, pp. 1–5, https://doi.org/10.1109/LASCAS60203.2024.10506171.

[14] GCC 14, "GCC 14 Release Series Changes, New Features, and Fixes," 2024, https://gcc.gnu.org/gcc-14/changes.html.

[15] R. S. Reusch, L. R. Juracy, and F. G. Moraes, "Deploying Machine Learning in Resource-Constrained Devices for Human Activity Recognition," in *SBESC*, 2023, pp. 1–6, https://doi.org/10.1109/SBESC60926.2023.10324073.

[16] ——, "Assessment and Optimization of 1D CNN Model for Human Activity Recognition," in *SBESC*, 2022, pp. 1–7, https://doi.org/10.1109/SBESC56799.2022.9964520.