Accelerating Machine Learning using RISC-V Vector Extension in a Manycore Platform

Willian Analdo Nunes, Antônio Vinicius Corrêa dos Santos, César Marcon, and Fernando Gehm Moraes
School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
willian.nunes@edu.pucrs.br, antonio.s001@edu.pucrs.br, cesar.marcon@pucrs.br, fernando.moraes@pucrs.br

***Abstract*—This work addresses the acceleration of convolutional neural network (CNN) inference in manycore architectures using coarse and fine-grain parallelism. Prior approaches focus on dedicated accelerators or modified NoCs, limiting flexibility. This work proposes integrating a RISC-V processor extended with the vector extensions (RVV) as general-purpose processing elements in a NoC-based manycore. The implementation applies depthwise convolution mapped across PEs and uses auto-vectorization provided by the compiler. Experiments on a 4×4 manycore running the first AlexNet layer achieved up to 5.70x speedup and reduced execution cycles by 82.45% compared to a scalar single-core baseline.**

***Index Terms*—Manycore, RISC-V Vector Extension, Hardware Acceleration, CNN Inference Acceleration.**

I. Introduction

Hardware accelerators, specialized components optimized for specific tasks, have regained prominence due to the increasing computational demands of complex tasks, such as video processing and machine learning (ML) [1]. Hardware acceleration can be achieved either by coarse-grain (MIMD) or fine-grain (SIMD) parallelism.

Coarse-grain acceleration is typically implemented using manycore architectures, where multiple processing cores are integrated into a single chip to enable concurrent task execution. In ML applications, such architectures accelerate complex tasks during inference, leading to faster and scalable solutions.

Fine-grain acceleration can be achieved through vector processing. RISC-V [2], an open-source instruction set architecture (ISA), includes the Vector Extension (RVV) [3], which enhances the base ISA by incorporating vector processing capabilities. The RVV extension introduces 32 vector registers whose widths are implementation-defined via a parameter called **VLEN**. RVV supports runtime-configurable element widths, allowing applications to operate on data elements of various sizes, ranging from 8 bits up to ELEN bits, where ELEN denotes the maximum element width supported by the hardware. Additional runtime configurations, such as vector length (VL) and mask registers, control vector instruction execution, enabling complex and flexible operations.

The *objective* of this work is to investigate the speedup to execute the inference phase of convolutional neural networks (CNNs) using both coarse- and fine-grain parallelism. Coarse-grain acceleration is achieved through a method known as depthwise convolution, combined with a mapping strategy that optimizes traffic within the Network-on-Chip (NoC). Fine-grain acceleration is accomplished via vectorized convolution processing, using the auto-vectorization capabilities provided by the GCC compiler. The implementation of these hardware acceleration techniques, and the resulting speedup evaluation, is the *original contribution* of this work.

II. Related Work

The literature presents multiple works that employ NoCs for domain-specific acceleration (DSA). The reviewed works rely on NoCs to interconnect PEs specifically tailored for their target applications, often optimized for operations such as multiply-accumulate (MAC). Consequently, most NoC-based works focus on NoC architectures designed explicitly for DSA.

Ultra-NoC [4] employs a 13×13 mesh interconnect with 169 tiles to implement a unified routing scheme supporting unicast, multicast, and broadcast for diverse Deep Neural Network (DNN) data patterns. Each tile integrates a router, network interface, and a Processing Element (PE) comprising a multiplier, adder, and control logic. Ultra-NoC reduces memory access by 26–49% (AlexNet) and 30–56% (ResNet-50).

Eyeriss v2 [5] employs a hierarchical mesh NoC optimized for compact DNNs like MobileNet. Each PE handles 8-bit fixed-point data and includes local scratchpads, with global buffers (GLBs) forming a two-level memory hierarchy. Separate NoCs are dedicated to transferring inputs, weights, and partial sums. The architecture comprises 16 PE and 16 GLB clusters in an 8×2 array, each containing 12 PEs (3×4).

Meta's [6] MTIA v1 employs an 8x8 mesh NoC that can be subdivided to run independent jobs. Each PE is equipped with two RISC-V processor cores (one with vector extension) and accelerators for domain-specific applications.

Intra-NoC acceleration modifies NoC components (e.g., routers) for specific Neural Network (NN) tasks. NOVA [7] introduces a NoC-based vector unit that approximates non-linear operations and broadcasts values via enhanced routers with lookup-driven data routing. Tiwari et al. [8] integrate In-Network Accumulation (INA) into routers, adding partial sum accumulation blocks that cut runtime by 1.22x and boost power efficiency by 2.16x. Gao et al. [9] propose a dual-layer NoC accelerator for lightweight CNNs, comprising a NoC array, PE clusters, a loop controller, and an ISA-based instruction decoder. Each PE performs a MAC operation, with the controller managing memory and data flow.

The effective mapping of DNNs onto NoC-based accelerators is key to minimizing communication overhead. Zhao et al. [10] group convolutions and map them to PEs using a genetic algorithm on a PE communication graph (PCG). By assigning pooling and convolution to the same PE, and allocating one PE per channel in early layers, their method achieves up to 50% power reduction.

Liang et al. [11] apply a modified Whale Optimization Algorithm for multi-objective CNN mapping, balancing energy, latency, and load. Despite its limitations for discrete spaces, it yields significant power and latency improvements.

Ye et al. [12] propose a task allocation scheme for resource-constrained NoCs, grouping CNN tasks under PE capacity constraints. A dynamic dense reverse mapping algorithm uses the PCG to reduce power by 31% and latency by 38%.

GAP-8 [13] uses a RISC-V controller allied with an 8-Core cluster with an additional dedicated accelerator for convolutions, the Hardware Convolution Engine. Although ARA2 [14] uses a core with an RVV accelerator in a multicore system, it lacks NoC interconnects, highlighting a gap in integrating RVV-based PEs within NoC architectures. Manycore designs often compensate for limited PE capability with fine-grain NoC router modifications, due to the use of MAC-centric PEs requiring NoC-level enhancements for performance.

Table I categorizes prior work. Unlike these, our approach uses general-purpose PEs with internal acceleration, enabling broader task flexibility. Our approach employs a NoC interconnecting RVV-enabled RISC-V cores, exploiting SIMD parallelism within vector cores and MIMD across the many-core fabric. The NoC facilitates efficient, concurrent inter-core communication without performance loss. We close a *gap* in CNN acceleration by integrating RVV-based processors into a NoC-based manycore, enabling support for diverse workloads with internally accelerated, general-purpose PEs.

TABLE I
RELATED WORK ON NN (DNN, CNN, ML) ACCELERATION.

	Architecture		**Target Application**
Ultra-NoC [4] (2024)	NoC Modification	Hybrid data transmission to leverage different data reuse methods	DNN
Eyeriss v2 [5] (2019)	Custom NoC Architecture	Separate NoC for different data types and clusters of PEs and GLBs	DNN on Mobile Devices
NOVA [7] (2024)	Intra-NoC Acceleration	Special router to support non-linear operations	CNN Attention Layers
Tiwari et al. [8] (2022)	Intra-NoC Acceleration	Routers support partial sum accumulation (In-Network Accumulation (INA))	DNN
Gao et al. [9] (2023)	Custom NoC architecture	Dual-layer NoC-based accelerator with dedicated custom ISA	CNN for Edge Computing
Zhao et al. [10] (2020)	Mapping method	PEs communication graph optimization algorithm (Genetic Algorithm)	CNN
Liang et al. [11] (2024)	Mapping method	PEs communication graph optimization algorithm (Whale Optimization Algorithm)	CNN
Ye et al. [12] (2023)	Mapping method	PEs communication graph optimization algorithm considering resource limitation	CNNs on resource-limited NoC
GAP-8 [13] (2018)	Manycore	RISC-V Controller + RISC-V 8-Core Cluster W/ Dedicated Accelerator	CNN on the edge
ARA2 [14] (2024)	Multicore	Multicore Cluster of RISC-V core W/ RVV Co-Processor	ML
This Work	General-Purpose PE with vector acceleration	Usage of a general purpose PE with a tightly coupled accelerator	ML (CNN)

III. CNN ACCELERATION METHOD

This section presents (i) the strategy for distributing a CNN through multiple PEs and (ii) the CNN mapping method.

The selected CNN mapping strategy exploits input channel independence to maximize parallelism via **depthwise convolutions** [15]. This technique assigns different input channels to separate PEs, enabling intra-layer parallelism. As each input channel and filter pair produces one output channel, output channel computations are also independent. Thus, a 1st PE layer performs depthwise convolutions, passing each output channel's feature map to a 2nd PE layer, aggregating the results. While the 2nd layer processes one set of outputs, the 1st layer can advance to the next, enhancing pipeline efficiency.

Equation (1) describes the standard convolution.

$$\mathbf{O}[c_o][x][y] = \mathbf{B}[c_o] + \sum_{k=0}^{C_i-1} \sum_{i=0}^{\text{Width}-1} \sum_{j=0}^{\text{Height}-1} (\mathbf{I}[k][Sx+i][Sy+j] * \mathbf{W}[c_o][i][j]) \quad (1)$$

where: c_o is the current output channel; x and y are the horizontal and the vertical positions; C_i is the total number of input and filter channels; Width and Height correspond to the filter size; S is the stride; O is the output; I is the input; W is the filter tensors; and B is the bias vector.

Equation 2 describes the convolution operation performed by the 1st PE layer, using the depthwise convolution method. It differs from the standard convolution by not having the outer summation, which sums the input channels.

$$\mathbf{O_p}[c_o][x][y] = \sum_{i=0}^{\text{Width}-1} \sum_{j=0}^{\text{Height}-1} (\mathbf{I}[K][Sx+i][Sy+j] * \mathbf{W}[c_o][i][j]) \quad (2)$$

where: O_p: partial output; K: input channel (fixed parameter in PEs that performs depthwise convolution).

To compensate for the removal of the outer summation, the 1st PE layer must include c_i processing elements (PEs), corresponding to the number of output channels. Each PE is assigned a fixed *K* parameter, which designates the specific input channel it is responsible for processing, and performs the computation defined in Equation (2).

The operation output O_p is partial and needs to be summed with the partial outputs from other input channels to generate the output feature map. The operation in Equation (2) is executed for every output channel in the 1st PE layer.

Equation (3) defines the convolution in the 2nd PE layer, finishing the operation started in Equation (2). It aggregates all partial outputs from the 1st PE layer per output channel, adds the bias **B**, and generates the final output feature map. This layer may also apply ReLU and Max-Pooling to balance the computational load. The resulting feature maps are then passed to a 3rd PE layer, initiating the next convolution via depthwise convolution.

$$\mathbf{O}[c_o] = \mathbf{B}[c_o] + \sum_{k=0}^{C_i-1} O_p[c_o][C_i] \quad (3)$$

The number of PEs varies with layer depth and manycore size. For example, the 1st convolutional layer of AlexNet has three input channels, so the 1st PE layer consists of three PEs, each handling one channel. Each PE performs 96 iterations of depthwise convolution, matching the output depth of the layer. After each iteration, results are sent to the 2nd PE layer.

The 2nd PE layer comprises one PE and is responsible for receiving the data sent by the 1st PE layer after the processing of each output channel. It then sums the partial feature maps, adds the bias, and performs the ReLU and Max-Pool operations. The output feature map is then sent to the 3rd PE layer, initiating the next convolutional layer.

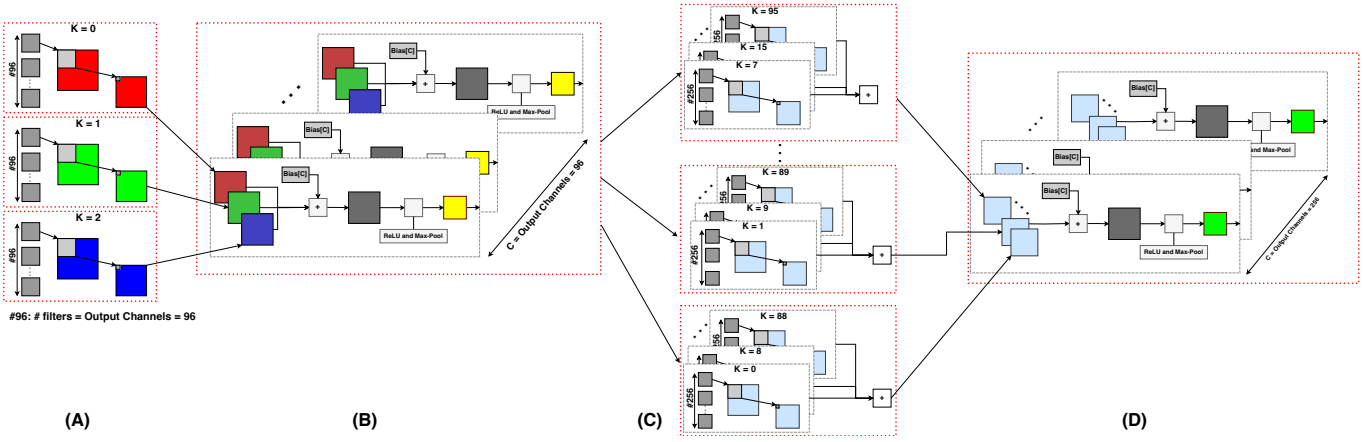

Fig. 1. 1st and 2nd layers of the depthwise convolution. Red dashed lines indicate the computation boundaries for each process. **(A)** P1 processes, each handling one input channel. **(B)** P2 process, which receives partially convolved channels from all P1 processes, accumulates them to produce the 1st convolution output, and applies ReLU and Max-Pool to generate the output feature map (in yellow). **(C)** P3 processes, each handling multiple input channels. **(D)** P4 process, which receives partially convolved data from P3, accumulates it, and applies ReLU and Max-Pool to produce the 2nd output feature map (in green).

In the 3rd PE layer, the input depth is 96 and the output depth is 256 (in the AlexNet example). Due to the high input depth, assigning a PE per input channel is impractical, as it would require an excessive number of PEs. Instead, each PE processes multiple input channels, with the number per PE determined by $\frac{96}{PEs_3}$, where PEs_3 is the number of allocated PEs. Each PE accumulates partial outputs from its assigned channels before passing them to the 4th layer.

The 4th PE layer is similar to the 2nd one. It is responsible for gathering the partial outputs, summing them, performing the ReLU and Pooling operations if required, and then sending the result to the 5th PE layer, which has a structure similar to the 3rd layer.

Four C-programs were developed using this partitioning approach, each named with a leading "P" followed by the corresponding PE layer number. For clarity, only four PE layers were implemented, covering the first two convolutional layers of AlexNet.

- *P1* - Entry-point program implementing the 1st PE layer. It processes a single input channel and forwards the result to the second program. Three instances are required, each running with a different K parameter (input channel).
- *P2* – Implements the 2nd PE layer. It receives partial outputs from P1 processes, sums them to form the output feature map, applies ReLU and Max-Pool, and forwards each output channel to the next PE layer.
- *P3* – Implements the 3rd PE layer. It processes multiple input channels (based on PE count and convolution depth), receives data from P2, performs depthwise convolution, and sends results to the next layer.
- *P4* – Implements the 4th PE layer. It sums partial outputs from P3 processes to produce the output feature map, applies ReLU and Max-Pool, and passes each output channel to the following PE layer.

Figure 1(A-B) presents *P1* (3 instances) and *P2* (with a single instance) programs handling the 1st convolutional layer of AlexNet. Figure 1(C-D) presents *P3* (with 8 instances) and *P4* programs handling the 2nd convolutional layer of AlexNet.

Figure 2 illustrates the **mapping** of tasks (programs) onto the manycore, based on the number of PEs assigned to each PE layer: 3 for the 1st, 1 for the 2nd, 8 for the 3rd, and 1 for the 4th. For clarity, the figure is divided into three parts. The 13 tasks are mapped onto the manycore using 12 out of 16 available PEs. Only the PE located at position 1x2 executes more than two tasks. This mapping was chosen to minimize the hop count between tasks.

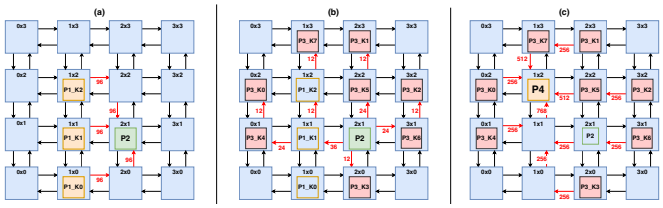

Fig. 2. Mapping in a 4x4 manycore. Red arrows represent communications, and the number associated represents the number of channels/packets sent. (a) three P1 instances communicating with the P2 process. (b) P2 process sending the channels to be processed by the eight P3 processes. (c) eight P3 processes sending data to the P4 process.

Figure 2(a) presents the three P1 instances and the P2 process. The P2 process is centralized to receive data from the maximum number of ports, avoiding packets using the same router. With the P1 and P2 mapping positions, the P2 process receives data from the South (S), West (W), and North (N) ports, while the East (E) is unused. Figure 2(b) presents the P2 process sending the first convolution layer output channels to the P3 processes (8x P3 instances). Each P3 process receives 12 channels to process. They are mapped to PEs surrounding the centralized PE to divide the packets between all the ports. The routing is not optimal given that the routing algorithm used by the NoC is the XY, which first routes the channel in the X-axis. Finally, Figure 2 (c) presents the P3 processes sending data to the P4 process. The P4 process is centralized and receives the packets in the following order: W, N, E, S, S, E, S, N. Although some ports are used sequentially, the NoC is not a bottleneck and can handle the data transmission without reducing the application's performance.

IV. Results

This work adopts the Memphis-V manycore [16]–[19] publicly available at https://github.com/gaph-pucrs/Memphis-V, providing coarse-grain acceleration. Memphis-V is modeled at the RTL level using SystemVerilog. A subset of the RISC-V Vector Extension (RVV) [20], aimed at accelerating Convolutional Neural Networks (CNNs), has been integrated into the processor. This extension features a configurable vector length (VLEN). This processor is responsible for the fine-grain acceleration through the RVV extension. The CNN model is compiled using GCC 14.1 [21], with support to **automatic vectorization**. The compiler's auto-vectorization showed limitations by only vectorizing portions of the code, but still provided good results without the need to modify the source code. A solution is adopt manual vectorization.

This Section presents the AlexNet [22] 2-D CNN results. Experiments adopted a 4x4 manycore, which mapping can be seen at Figure 2. Simulations were performed using a scalar PE and 3 accelerated versions. The 3 accelerated versions include the RVV extension with VLENs of 64, 128, and 256 bits.

Table II presents the results of the P1 tasks on the manycore. Each process only computes one of the three RGB input channels and generates the 96 (partial) output channels of the 1st convolutional layer for that given input channel.

TABLE II
Results on running RGB channels computation (P1) on MEMPHIS-V with vector extension.

	Computation			Communication	
VLEN	**Avg cycles per channel**	**Speed-up**	**Total cycles (x96 channels)**	**Avg cycles per channel**	**Total cycles (x96)**
-	4.74e+6	-	4.55e+8	1.19e+5	1.14e+7
64	4.75e+6	1.00x	4.56e+8	1.19e+5	1.14e+7
128	2.65e+6	1.79x	2.54e+8	1.34e+5	1.28e+7
256	2.37e+6	2.00x	2.27e+8	1.39e+5	1.34e+7

The average number of cycles required to compute each output channel is presented in the 2nd column of Table II. The scalar and 64-bit VLEN configurations exhibit comparable cycle counts. Implementations with larger VLENs achieve computational speedups, reducing the number of cycles required per output channel by up to **2x**. Communication time is lower by approximately one order of magnitude, as the computational load of the application significantly outweighs the volume of data transmitted. Overall, the 1st convolutional layer of AlexNet involves 105,415,200 multiplication operations. The P2 process performs fewer computations during execution, approximately one order of magnitude less than P1, and spends most of its time waiting for data from the P1 processes.

Table III compares the single and manycore versions of the 1st layer of the 2-D CNN benchmark, exploring the scalar and vectorized versions for multiple VLENs. In the single-core version, the speedup of using the vector extension reached 1.65x. Applying the manycore mapping technique, the speedup reached ≈**3x** using scalar PEs.

The best trade-off was the manycore version with a 128-bit VLEN, offering a speedup of **5.15x**, meaning a reduction of around 80% of the cycles compared to a scalar single-core and 1.72x of the scalar manycore.

TABLE III
Performance on the AlexNet first layer, comparing single and manycore architectures.

Single/-Manycore	**VLEN**	**Cycles**	**Speed-up**	**Cycle reduc. (%)**
Single	-	1,418,604,031	-	-
Single	64	1,418,604,052	1.00x	0.00
Single	128	1,012,741,796	1.40x	28.61
Single	256	858,535,157	**1.65x**	39.48
Many	-	474,913,995	**2.99x**	66.52
Many	64	475,713,468	2.98x	66.47
Many	128	275,530,467	**5.15**x	80.58
Many	256	248,906,823	5.70x	82.45

Results for the 2nd AlexNet layer are omitted due to memory constraints on the manycore. Partial simulations validated the partitioning method's correctness, but speedup could not be measured. A potential solution is to store application parameters in external memory, minimizing local storage use by dynamically loading weights and biases as needed.

Direct comparison with related work is difficult due to different parameters and goals. Works that involve NoC architecture have very different PEs, as our work focuses on having a general purpose, while theirs focuses on having dedicated accelerators. Manycore works are more similar due to using RISC-V cores and the RVV extension, but they lack NoC interconnection.

The Authors of GAP-8 [13] report speedups ranging from 5.3x to 8.4x when using eight cores in the cluster. These results are comparable to those presented in this work in acceleration regarding the 1st layer of the CNN using a distributed approach, where with only four PEs, we achieved a speedup of up to 5.7x. For full CNNs, the performance improvement of GAP-8 ranged from 9.3x to 11.7x using 8-PEs. ARA2 [14] concluded that a multicore design with smaller Ara2 instances behaves better than a single-core larger Ara2. This work achieved similar results, as having multiple scalar cores achieved a better speedup (almost double) than the biggest single core with VLEN of 256 bits.

V. Conclusions and Future Work

This work demonstrated that integrating RVV-enabled RISC-V cores into a NoC-based manycore enables CNN inference acceleration through combined SIMD and MIMD parallelism. The proposed method achieved a 5.70x speedup and an 82.45% reduction in execution cycles compared to a scalar single-core baseline. These results indicate that combining vectorized processing and manycore mapping strategies enhances the efficiency of CNN inference. Distributing a CNN layer to multiple PEs also helps keep memory usage low by reducing the number of parameters each PE must hold.

Future work includes: (i) support for additional CNN layers (and full network); (ii) integrate external memory to address local storage limitations; (iii) evaluate scalability on larger manycore configurations; (iv) explore the suitability of manycores and the vector extension for other machine learning structures beyond CNN.

ACKNOWLEDGMENTS

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Finance Code 001; Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), grants 308182/2023-5 and 305621/2024-6; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), 23/2551-0002200-1.

REFERENCES

[1] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *Journal of Systems Architecture*, vol. 129, pp. 1–51, 2022, https://doi.org/10.1016/j.sysarc.2022.102561.

[2] RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20240411," 2024, https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link, January 2025.

[3] ——, "RISC-V "V" Vector Extension," 2021, https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[4] K.-C. J. Chen, H.-H. Peng, and P.-C. Shen, "Ultra-NoC: Unified Low-Transmission Routing Assisted NoC for High-flexible DNN Accelerator," in *IEEE International System-on-Chip Conference (SOCC)*, 2024, pp. 1–5, https://doi.org/10.1109/SOCC62300.2024.10737754.

[5] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019, https://doi.org/10.1109/JETCAS.2019.2910232.

[6] Meta AI, "MTIA v1: Meta's first-generation AI inference accelerator," 2023, https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/.

[7] M. Upadhyay, R. Juneja, W.-F. Wong, and L.-S. Peh, "NOVA: NoC-based Vector Unit for Mapping Attention Layers on a CNN Accelerator," in *ACM/IEEE Design, Automation Test in Europe Conference (DATE)*, 2024, pp. 1–6, https://doi.org/10.23919/DATE58400.2024.10546727.

[8] B. Tiwari, M. Yang, X. Wang, and Y. Jiang, "In-Network Accumulation: Extending the Role of NoC for DNN Acceleration," in *IEEE International System-on-Chip Conference (SOCC)*, 2022, pp. 1–6, https://doi.org/10.1109/SOCC56010.2022.9908106.

[9] J. Gao, Q. Shao, F. Deng, Q. Wang, N. Jing, and J. Jiang, "An NoC-based CNN Accelerator for Edge Computing," in *International Conference on ASIC (ASICON)*, 2023, pp. 1–4, https://doi.org/10.1109/ASICON58565.2023.10396346.

[10] Y. Zhao, F. Ge, C. Cui, F. Zhou, and N. Wu, "A Mapping Method for Convolutional Neural Networks on Network-on-Chip," in *International Conference on Communication Technology (ICCT)*, 2020, pp. 916–920, https://doi.org/10.1109/ICCT50939.2020.9295883.

[11] Z. Liang, F. Hu, B. Xu, and C. Wei, "Multi objective non dominated sorting whale optimization genetic algorithm for convolutional neural network-based on-chip networks," in *International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, 2024, pp. 653–656, https://doi.org/10.1109/AINIT61980.2024.10581699.

[12] J. Ye, F. Ge, and F. Zhou, "A Method of Mapping Convolutional Neural Networks on Resource-limited NoC Platform," in *International Conference on ASIC (ASICON)*, 2023, pp. 1–4, https://doi.org/10.1109/ASICON58565.2023.10396382.

[13] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–4, https://doi.org/10.1109/ASAP.2018.8445101.

[14] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini, "Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1822–1836, 2024, https://doi.org/10.1109/TC.2024.3388896.

[15] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: efficient convolutional neural networks for mobile vision applications (2017)," *CoRR*, vol. abs/1704.04861, pp. 1–9, 2017, http://arxiv.org/abs/1704.04861.

[16] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, "Memphis: a framework for heterogeneous many-core SoCs generation and validation," *Springer Design Automation for Embedded Systems*, vol. 23, no. 3-4, pp. 103–122, 2019, https://doi.org/10.1007/s10617-019-09223-4.

[17] M. Ruaro, F. B. Lazzarotto, C. A. Marcon, and F. G. Moraes, "DMNI: A Specialized Network Interface for NoC-based MPSoCs," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1202–1205, https://doi.org/10.1109/ISCAS.2016.7527462.

[18] E. Wachter, L. L. Caimi, V. Fochi, D. Munhoz, and F. G. Moraes, "BrNoC: A broadcast NoC for control messages in many-core systems," *Microelectronics Journal*, vol. 68, pp. 69–77, 2017, https://doi.org/10.1016/j.mejo.2017.08.010.

[19] 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 *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, 2024, pp. 1–5, https://doi.org/10.1109/LASCAS60203.2024.10506171.

[20] W. A. Nunes and F. G. Moraes, "Accelerating Machine Learning with RISC-V Vector Extension and Auto-Vectorization Techniques," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1–5, https://doi.org/10.1109/ISCAS56072.2025.11043225.

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

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017, https://doi.org/10.1145/3065386.