RISC-V^2: A Scalable RISC-V Vector Processor

Kariofyllis Patsidis*, Chrysostomos Nicopoulos†, Georgios Ch. Sirakoulis*, Giorgos Dimitrakopoulos*
*Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece
†Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus

***Abstract*—Machine learning adoption has seen a widespread bloom in recent years, with neural network implementations being at the forefront. In light of these developments, vector processors are currently experiencing a resurgence of interest, due to their inherent amenability to accelerate data-parallel algorithms required in machine learning environments. In this paper, we propose a scalable and high-performance RISC-V vector processor core. The presented processor employs a triptych of novel mechanisms that work synergistically to achieve the desired goals. An enhanced vector-specific incarnation of register renaming is proposed to facilitate dynamic hardware loop unrolling and alleviate instruction dependencies. Moreover, a cost-efficient decoupled execution scheme splits instructions into execution and memory-access streams, while hardware support for reductions accelerates the execution of key instructions in the RISC-V ISA. Extensive performance evaluation and hardware synthesis analysis validate the efficiency of the new architecture.**

I. Introduction

The last few years have witnessed the widespread proliferation and massive adoption of machine learning as a fundamental thrust in a multitude of application domains. Increasingly, more aspects of everyday life are being disrupted by new capabilities enabled by machine learning. Neural Networks (NN) have emerged as the most popular approach to implementing machine learning, and they are considered state-of-the-art in such applications as pattern [1], image [2], and speech recognition. The rapid and vast increase in the use of NNs has accentuated the demand for hardware architectures that can accelerate the processing of various operations encountered in machine learning applications.

Traditional general-purpose processors have focused on Instruction-Level Parallelism (ILP) for decades. Consequently, they are not tuned to effectively handle the massively data-parallel workloads that machine learning algorithms and NNs have brought to the forefront [3]. While the addition of SIMD instructions to the ISA of general-purpose machines partially exploits Data-Level Parallelism (DLP), the obtained throughput is somewhat limited [4]. On the other hand, Graphics Processing Units (GPU) provide very high data parallelism, so they have been extensively used to accelerate NN workloads. Nevertheless, GPUs tend to be power-hungry and the energy efficiency they can achieve is not adequate for many implementations, e.g., those requiring computation on the edge, where battery life is of paramount importance [5] [6]. To address energy efficiency, researchers have turned to custom architectures targeting specific NN implementations [7], [8], [9]. Even though such application-specific designs are very efficient, they typically offer limited programmability and small flexibility in adapting to the evolving and emerging needs of NN workloads.

The search for high performance and energy efficiency in highly data-parallel workloads has brought vector processors – a concept heavily explored in the 1970s [10] – back into the spotlight. Vector architectures are almost unique in their ability to effectively combine high programmability attributes, high computational throughput, and high energy efficiency [11], [12]. The inception of modern vector processors was triggered by NN applications, which copiously rely on operations that can be readily vectorized [11]. The extensive proliferation of NNs in the last few years is precisely why vector processing is regaining notable traction in the community [13].

Building on this momentum, this paper presents a vector processor architecture that leverages the upcoming RISC-V [14], [15] vector extension [16], which allows RISC-V-based processors to be augmented with a vector processing core. While the proposed architecture is founded on the traditional tenets of vector processing [17], [18], [19], it introduces some novel techniques that reap high performance benefits in a very scalable and cost-effective implementation. Specifically, the new design is spearheaded by three mechanisms that collectively constitute the main contributions of this work:

- A new *register remapping* technique reimagines the notion of register renaming in a vector processing context. Coupled with a dynamically allocated register file, the new register remapping mechanism enables dynamic hardware-based loop unrolling and optimized instruction scheduling at run-time.
- The design's *decoupled execution* scheme employs resource acquire-and-release semantics to disambiguate between parallel computation and memory-access instruction streams, thereby allowing for independent execution/memory flow rates.
- A dynamically generated *hardware reduction tree* enables significant acceleration of reduction intstructions, which are prevalent in most NN and DSP algorithms.

The efficacy and efficiency of the presented vector processor are corroborated through extensive performance simulations using real benchmark applications, and detailed hardware analysis of synthesized and placed-and-routed designs using commercial 45 nm standard-cell libraries.

II. The Proposed Vector Processor Architecture

The proposed processor design uses a superscalar core as the main control processor, with all the instructions being fetched and decoded in the superscalar pipeline, similar to [20], [21], [22]. A high-level overview of the micro-architecture is depicted in Figure 1. During the superscalar issue stage (sIS), the instructions are diverted to the correct path (i.e., scalar, or vector), based on their type. A vector instruction queue decouples the execution rates of the two datapaths. The vector processor core itself is implemented in a diversified pipelined organization, whereby the actual pipeline depth experienced by each instruction depends on the instruction type, as will be shortly explained. The vector pipeline includes the following

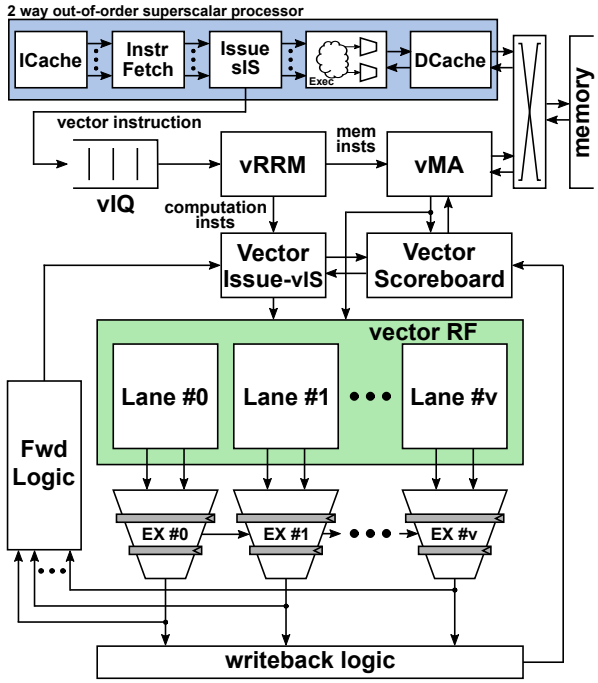

Fig. 1. A high-level overview of the micro-architecture of the proposed vector processor. All vector instructions are diverted to the vector execution path upon completion of the scalar Issue Stage (sIS).

stages: (a) Register Remap (vRRM), (b) Instruction Issue (vIS), (c) Execution (vEX), and (d) Memory Access (vMA). Computation instructions are decoupled from memory-access instructions, and the two instruction types follow different pipeline paths, as illustrated in Figure 1 and explained in Section II-B.

During the first vector pipeline stage (vRRM), the instruction operands are remapped to point to their new allocated locations. This process is facilitated by a dynamic register file allocation mechanism, as will be described in Section II-A. The remapped instructions then propagate to the issue stage (vIS), where they access the vector register file (RF), and/or the forwarding paths (as vector chaining [17]), to get their source data, before proceeding for execution. The vector RF is sliced into v lanes, with each one corresponding to a separate parallel execution lane. Vectors of arbitrary length are stripmined to the maximum number of lanes supported.

Hazarding is also implemented in this stage through the use of a scoreboard. If all the operands of an instruction are ready, they can be read directly from the register file. If any operand is pending (i.e., an earlier instruction that is currently in execution is the producer of the operand value), the instruction will be stalled – by the scoreboard – until the pending value appears on the forwarding path. Once all operands are available, the instruction can proceed to the execution stage (vEX). Since vector instructions operate on multiple elements (i.e., entire vectors), the vIS stage "transforms" vector instructions into multiple micro-operations (μops), with each μop operating on different register groups.Scheduling in the vIS stage is, therefore, performed at the granularity of individual μops.

The execution stage contains the pipelined parallel execution lanes. Similar to [17], each execution lane sees a portion of the vector RF. The duration, in cycles, of the vEX stage

TABLE I
THE EXECUTION LATENCIES OF THE VARIOUS INSTRUCTION TYPES.

Instruction Type	Latency (cycles)
Simple arithmetic & logical	1
Multiplication	3
Division	4
Reductions	Variable: $\log_2(vector\_length)$
Load/Store	Variable

is variable, and it depends on the operation being executed. Table I lists the latencies for the various classes of instructions. When a result is generated, it becomes available to the issue stage through the forwarding paths. Since the execution latency is variable, the orchestration of instruction progress is performed by the scoreboard, which notifies stalled instructions in the issue stage whenever their pending operand values are ready. The stalled instructions "wake up" and proceed to the next pipeline stage. During execution, vector μops may trigger the same operation in multiple execution lanes, based on the vector length.

Memory instructions do not access the execution lanes; instead, they are routed after the vRRM pipeline stage directly to the memory unit, as depicted in Figure 1. The memory unit features two parallel engines that allow for the simultaneous processing and disambiguating of one load and one store instruction. All instructions in the vMA and vEX stages are always issued and retired in order, writing their results directly into the register file upon retirement.

A. Register remapping and dynamic register file allocation

The first key micro-architectural novelty of the proposed processor design is a brand new approach to register renaming within the context of vector processing. The mechanism is aptly called *register remapping*, and it operates within the vRRM pipeline stage shown in Figure 1. The register remapping mechanism enables *vector loops to be unrolled dynamically in hardware*, thereby (a) minimizing the overhead of control instructions executed in the superscalar pipeline, and (b) maximizing the utilization of the available fetch bandwidth. The operation of the register remap scheme comprises three distinct phases, as abstractly depicted in Figure 2.

In the first phase, the mechanism *generates groups* of vector registers, based on the number of logical registers requested by the software. In the RISC-V ISA vector extension, the software communicates to the processor – through specialized system registers – the desired amount of logical registers for the upcoming computations. This information is leveraged to generate the desired register-group numbers and sizes, as shown in Figure 2.

Upon completion of group generation, the proposed mechanism proceeds to the second phase of its operation; it uses a remapping table (similar to a register alias table) to remap the logical registers to the corresponding base address of their assigned register group. Since these assignments are static for the duration of each computational kernel, the remap table is only written once per logical register, during the first time each new destination operand is encountered in the instruction stream. Contrary to traditional register renaming [23], the presented register remapping process does not perform one-to-one register mappings; it performs *one-to-group* register mappings, whereby a *single* logical register is mapped to a

group of registers to enable loop unrolling. Subsequently, the remapping table dynamically allocates the generated register groups into the register file, as illustrated in Figure 2.

Finally, in phase three of the scheme, the remapped instructions are "expanded"' to operate on the full size of their groups. In the vIS pipeline stage, each instruction generates and dispatches multiple micro-operations (μops) to the execution stage (vEX). In the example of Figure 2, the original instruction generates two μops. Each dispatched μop executes the parent instruction's operation, but with adjusted operands, so that the computation is applied to a different set of inputs within the assigned group space. Once the μops have covered the full group size, the parent instruction is retired. This expansion scheme also exists inside the memory unit, since memory instructions also need to undergo the same transformation.

In summary, the new register remapping mechanism facilitates dynamic loop unrolling in hardware. The unrolling mitigates the stalls incurred by data dependencies, since the direct consumer of a result is now separated from its producer by multiple μops. Consequently, resource utilization increases substantially.

B. Decoupled execution: computation and memory accesses

To further increase the utilization of the vector pipeline, we also introduce a novel memory decoupling scheme that effectively hides the latency of memory accesses. The computation (execution) and memory instructions are separated into two independent streams, which are appropriately diverted into different execution paths after the vRRM pipeline stage.

Traditionally, synchronization in decoupled processor schemes is achieved by employing so called *synchronization queues* and special move operations [24]. However, such schemes are not amenable to vector processors, where hundreds of elements have to be moved from/to the memory. The synchronization queues incur a significant hardware overhead, while the inserted move instructions block the computation stream until all the data has been transfered.

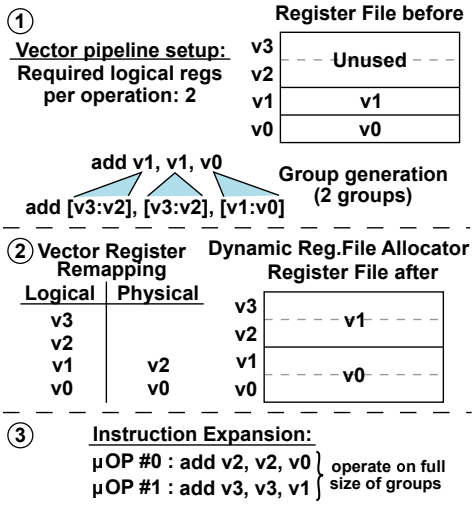

Fig. 2. The three-phase process of remapping the registers and dynamically allocating the register file. The software initially requests a number of logical registers, which dictates the number of groups the vector register file is going to be split into. Each instruction dispatches multiple μops to cover the full size of the allocated group.

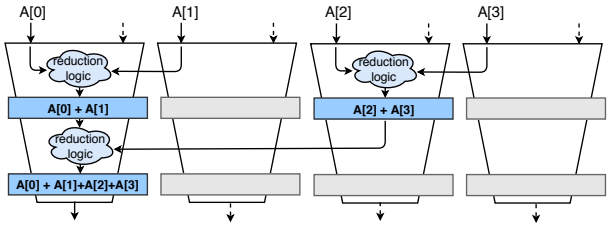

Fig. 3. The deployment of the dynamically generated hardware reduction tree in a setup with 4 execution lanes. Each cycle the length of the vector is reduced in half by computing the neighboring partial results, until the final result is ready.

To alleviate these shortcomings of the traditional approach, the proposed scheme uses a resource-locking mechanism to effectively safeguard the correct program execution flow without hindering the flow of the computation stream. During the vRRM pipe stage, the memory instructions are diverted to the memory unit, while a *ghost* copy of the instruction is dispatched into the vIS stage. The *ghost instruction* only updates the scoreboard, by locking the source operands of the memory instructions. It then disappears from the pipeline without triggering any computation. This way, data to be stored, or address offsets, are safeguarded against tampering from future computation instructions. At the same time, the computational flow remains completely unblocked to continue dispatching instructions, as long as no instruction tries to modify the locked registers. When the memory instruction finally retires, the data is written directly into the register file, and the corresponding registers are unlocked, thereby allowing for their subsequent reuse.

C. Hardware support for reduction operations

Reduction operations have historically been handled using specialized hardware that iteratively shifts and computes on pairs of elements. However, such approaches tend to have long execution latencies and are unfit for contemporary NN (and convolutional NN) applications that rely heavily on such computational patterns. To effectively accelerate reduction operations, we employ a *scalable tree* scheme, which calculates multiple partial results in parallel, in order to achieve significant speedups.

The reduction tree is automatically generated and distributed, based on the design's number of configured vector execution lanes. The generated tree includes all necessary interconnections between the execution lanes. During each pipeline stage, the tree operates on pairs of neighbors, reducing the input vector's dimensionality in half. The partial results are then registered and used in the next stage's computations. The organization of the reduction tree is depicted in Figure 3 for four execution lanes. The vector length being operated on at any given time determines the required reduction depth, which, in turn, triggers the reduction tree control logic to dynamically activate the appropriate interconnects of the tree. Since the tree is automatically generated and the interconnects dynamically activated at runtime, the scalability of the overall design is maintained, without requiring any manual effort in adjusting the design's RTL code. The proposed reduction scheme yields significant latency improvements; the execution latency of the unit is calculated as $\log_2(vector\_length)$.

III. Evaluation Results

A. Performance evaluation

In this sub-section, we perform a detailed performance evaluation of the proposed vector design and its key features. A total of 10 benchmark applications are employed, consisting of 7 well-known linear algebra kernels and basic DSP algorithms, and 3 NNs of varying complexity: a simple perceptron, a 4-stage convolutional NN, and an 8-stage deep convolutional NN. The examined NNs execute inference tasks on digit recognition using the MNIST database [25]. The compared designs were implemented in fully-functional and synthesizable RTL code that will be open-sourced on GitHub [26]. All benchmarks were cycle-accurately executed at the RTL level, with various statistics retrieved from hardware counters and speciliazed trackers facilitating processor profiling.

We first examine the impact of the novel *register remapping scheme* discussed in Section II-A. We compare the proposed design with a simpler baseline vector processor [22] that does not have the register remapping mechanism and operates with a shorter pipeline (i.e., one without the vRRM stage). Figure 4 depicts the results, normalized to the throughput of the baseline design. The average throughput – calculated as Elements Per Cycle (EPC), the ratio of total elements over the execution time – increases by 2.1×. This significant improvement is primarily attributed to the enhanced instruction scheduling resulting from the synergistic effect of register remapping, instruction expansion, and the dynamically allocated register file.

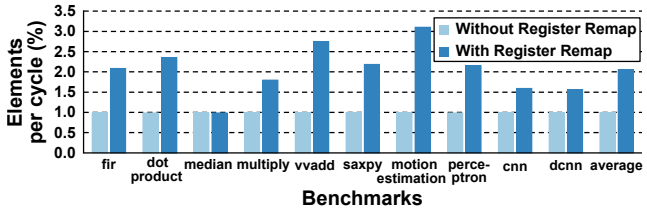

Fig. 4. The performance improvement obtained when using the novel register remapping mechanism and the dynamic allocation of the register file. Both vector cores feature 8 execution lanes.

The next feature we evaluate is the hardware support for reduction operations, as presented in Section II-C. The presence of a reconfigurable reduction tree improves the performance of NN algorithms. Consequently, our experiment focuses on the three NN benchmarks, since they make heavy use of reduction operations. Table II shows the obtained throughput results, normalized to the throughput of a design with no reduction tree. As can be seen, the hardware acceleration of reduction operations yields massive throughput improvements in the inference operations of the NN benchmarks.

TABLE II
Impact of the hardware-based reduction tree on the throughput (EPC) of NN algorithms in various configurations.

Design	Perceptron	CNN	Deep CNN	Average
No Red. Tree	1	1	1	1
With Red. Tree	2.57	1.89	1.87	2.11

Finally, we evaluate the scalability of the overall vector processor design. We compare three different vector configurations using 4, 8, and 16 execution lanes and a baseline dual-issue superscalar processor. All three vector cores have all the features presented in Section II. Figure 5 shows the obtained throughput results. An almost linear scaling (with the number of lanes) is achieved in the 7 linear algebra and DSP algorithms, but smaller gains are observed in the 3 NN algorithms. This is due to the complex memory access patterns that NN kernels exhibit (primarily using indexed accesses), leading to limited scaling.

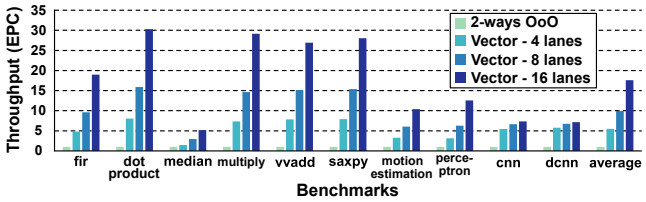

Fig. 5. Performance scaling for 3 different vector configurations, as compared to a baseline dual-issue superscalar core.

B. Hardware cost analysis

The proposed vector processor is also assessed in terms of its hardware cost and power efficiency. The RTL code of the design was synthesized using a commercial 45 nm standard-cell library under worst-case conditions (0.8 V, 125 °C), using the Cadence digital implementation flow. All designs under investigation were optimized for 1 GHz operation. The derived area/power results are summarized in Table III. Similar to Section III-A, the results also include the baseline dual-issue superscalar core. Recall that the proposed vector processor uses said superscalar core as the main control processor.

TABLE III
Hardware implementation results of four investigated designs (caches are excluded) at 45 nm / 0.8 V at 1 GHz.

Design	Area (mm^2)	Avg. Power (mW)	Power Efficiency
Superscalar	0.24	13.1	1
Vector - 4-Lane	0.61	43.9	1.54
Vector - 8-Lane	0.97	75.3	1.62
Vector - 16-Lane	1.67	124	1.70

As expected, the area increases significantly when augmenting the superscalar processor with a vector core. The area overhead of the vector core scales almost linearly with the increase in the number of execution lanes. The same trends are also followed by the power consumption. Nevertheless, modern systems (and especially resource-constrained ones) demand increasingly higher computational power implemented in a cost-effective manner. Therefore, a key metric is that of *power efficiency* (EPC/Watt). Clearly, the proposed architecture achieves a markedly better overall power efficiency that scales well with bigger vector configurations.

IV. Conclusion

This work presented a RISC-V-based high-performance and power-efficient vector processor architecture. The new design employs three novel mechanisms that collectively yield impressive performance gains: (a) a register remapping scheme facilitated by dynamic register file allocation; (b) a decoupled execution scheme that separates execution and memory-access instructions; and (c) hardware support for vector reduction operations. A detailed evaluation of the new architecture highlighted both its performance prowess and its power efficiency.

REFERENCES

[1] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8609–8613.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proc. of IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[3] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[4] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 129–140.

[5] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, "Neural network stream processing core (nnsp) for embedded systems," in *IEEE International Symposium on Circuits and Systems*, May 2006, pp. pp.–2776.

[6] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with cots hpc systems," in *Proc. of International Conference on International Conference on Machine Learning (ICML)*, 2013, pp. III–1337–III–1345.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2012, pp. 449–460.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[9] Y. Chen, T. 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, June 2019.

[10] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[11] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan, "Spert-ii: a vector microprocessor system," *Computer*, vol. 29, no. 3, pp. 79–86, March 1996.

[12] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: a vector extension to the alpha architecture," in *Proc. International Symposium on Computer Architecture (ISCA)*, May 2002, pp. 281–292.

[13] S. Hurkat and J. F. Martínez, "VIP: A versatile inference processor," in *IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2019, pp. 345–358.

[14] "RISC-V Foundation," http://www.riscv.org, accessed: 17-10-2019.

[15] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision," *Journal Signal Processing Systems*, vol. 84, p. 339–354, 2016.

[16] "Working draft of the proposed RISC-V V vector extension," https://github.com/riscv/riscv-v-spec, accessed: 17-10-2019.

[17] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[18] C. E. Kozyrakis and D. A. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov 2003.

[19] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core cpu accelerator," in *Proc. of ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, 2008, pp. 222–232.

[20] C. Celio, D. A. Patterson, and K. Asanovic, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep., Technical Report UCB/EECS-2015-167 2015.

[21] C. Celio, P. F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, "BOOM v2: An Open-Source Out-of-Order RISC-V Core," EECS Department, University of California, Berkeley, Tech. Rep., Technical Report UCB/EECS-2017-157 2017.

[22] K. Patsidis, D. Konstantinou, C. Nicopoulos, and G. Dimitrakopoulos, "A low-cost synthesizable risc-v dual-issue processor core leveraging the compressed instruction set extension," in *Microprocessors and Microsystems*, vol. 61, 2018, pp. 1–10.

[23] R. Espasa, M. Valero, and J. E. Smith, "Out-of-order vector architectures," in *Proc. of ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1997, pp. 160–170.

[24] R. Espasa and M. Valero, "Decoupled vector architectures," in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 1996, pp. 281–290.

[25] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, Nov 2012.

[26] IC-Lab-DUTH Repository. (2020) RISC-V-Vector processor. [Online]. Available: https://github.com/ic-lab-duth/RISC-V-Vector