Zoozve: A Strip-Mining-Free RISC-V Vector Extension with Arbitrary Register Grouping Compilation Support (WIP)

Siyi Xu[*]
Limin Jiang[*]
Shanghai University
Shanghai, China
xusiyi@shu.edu.cn
jianglimin@shu.edu.cn

Yintao Liu
Shanghai University
Shanghai, China
berialest@shu.edu.cn

Yihao Shen
Shanghai University
Shanghai, China
shenyihao@shu.edu.cn

Yi Shi
Shanghai University
Shanghai, China
yishi1996@shu.edu.cn

Shan Cao
Shanghai University
Shanghai, China
cshan@shu.edu.cn

Zhiyuan Jiang
Shanghai University
Shanghai, China
jiangzhiyuan@shu.edu.cn

Abstract

Vector processing is crucial for boosting processor performance and efficiency, particularly with data-parallel tasks. The RISC-V "V" Vector Extension (RVV) enhances algorithm efficiency by supporting vector registers of dynamic sizes and their grouping. Nevertheless, for very long vectors, the static number of RVV vector registers and its power-of-two grouping can lead to performance restrictions. To counteract this limitation, this work introduces Zoozve, a RISC-V vector instruction extension that eliminates the need for strip-mining. Zoozve allows for flexible vector register length and count configurations to boost data computation parallelism. With a data-adaptive register allocation approach, Zoozve permits any register groupings and accurately aligns vector lengths, cutting down register overhead and alleviating performance declines from strip-mining. Additionally, the paper details Zoozve's compiler and hardware implementations using LLVM and SystemVerilog. Initial results indicate Zoozve yields a minimum $10.10\times$ reduction in dynamic instruction count for fast Fourier transform (FFT), with a mere 5.2% increase in overall silicon area.

Keywords: RISC-V, vector processing, LLVM, hardware implementation

1 Introduction

Extensive computational needs have spurred advancements in vector instruction set architectures (ISAs). To better serve an array of computational requirements, modern vector extension technologies have gradually moved towards variable-length registers, which allow vector lengths to be dynamically adjusted to suit varying workloads. New vector ISAs, such as the Scalable Vector Extension (SVE) [18] and the RISC-V "V" Vector Extension (RVV), have been introduced.

These designs allow vectors to be resized dynamically according to the demands of particular computational tasks, thus providing additional flexibility. The objectives in designing SVE and RVV include optimizing performance and enhancing resource optimization by adapting to workload variations [15]. In contrast to traditional fixed-length vector registers, these modern vector extensions not only eliminate compatibility conflicts between hardware and software but also markedly enhance computational efficiency. Numerous companies and universities have developed diverse vector extensions supporting various RVV release versions, catering to different areas such as high-performance computing (HPC) [2, 10, 13, 14, 16], neural networks [1], [9], the internet of things (IoT), and edge computing [3], [5]. These innovations illustrate the ongoing efforts to boost performance and efficiency across multiple sectors.

The RVV extension faces intrinsic obstacles in domain-specific computations, such as wireless communications and artificial intelligence, which often involve ultra-long vectors. When vector length multipliers (LMULs) are larger, they limit the number of registers available for allocation, which increases register pressure and results in more register spilling. Conversely, smaller LMULs require frequent strip-mining, negatively affecting performance [7]. Moreover, developers must possess an in-depth comprehension of RVV's varied functionalities and operations and need to optimize register usage and memory access to devise efficient vectorized code, adding to the complexity of programming. Therefore, enhancing performance requires meticulous kernel optimization specific to applications, balancing LMUL settings and register availability, and understanding architectural details.

This paper introduces *Zoozve*, a novel RISC-V vector extension that eliminates the need for strip-mining, aimed at

---

[*]Both authors contributed equally to this research.

Figure 1. (a) Valid LMUL values for RVV include 4 and 8 across different vector lengths. (b) Zoozve supports arbitrary register grouping values such as 3 and 5 for asymmetric instructions.

overcoming performance challenges when handling ultra-long vectors. This extension enhances both resource utilization and processing efficiency by incorporating innovative instruction formats that accommodate essential vector operations and offer increased access to physical registers, thereby minimizing memory usage. This work includes several contributions, outlined as follows:

Strip-Mining-Free Vector Extension: To address the fixed register count and power-of-two register group issues found in RVV, a flexible RISC-V vector instruction extension without strip-mining is proposed.

Arbitrary Register Grouping Strategy: A register allocation strategy that adapts dynamically to the circumstances is introduced, intelligently modulating the distribution of registers in accordance with real-time vector lengths and the current register availability.

Compilation Support: Intrinsic splitting and assembly coalescing passes have been developed, together with a comprehensive compilation mechanism for Zoozve using LLVM, allowing effortless conversion from high-level code to efficient machine instructions.

2 Background and Motivation

Vector Strip-Mining: A fundamental aspect of vector ISAs is strip-mining, which enables vector processors to handle data volumes exceeding the capacity of available registers. This method partitions sizable vectors into smaller *strips*, each handled separately within a loop, which can be implemented in either hardware or software [6]. For example, Advanced Vector eXtensions (AVX) [8] implements strip-mining through software without having specific hardware control registers. In contrast, SVE uses the `whilelt` predicative instruction for loop termination management. TSUBASA

[11] and RVV enhance strip-mining by integrating hardware registers that dictate appropriate vector lengths for each strip. Crafting an ISA that minimizes conditional overhead while optimizing data-level parallelism demands careful consideration.

Vector Register Grouping: Data structures for RVV and Zoozve are depicted in Fig. 1. The power-of-two register grouping (RG) approach in RVV has limitations in two principal areas: (i) For relatively short vector lengths (VL), even though an LMUL can fit all vector elements into a single RG, a VL falling between $(2^{n_{lmul}-1}+1) \cdot \frac{VLEN}{VEW}$ and $(2^{n_{lmul}}-1) \cdot \frac{VLEN}{VEW}$ may cause under-utilization of vector registers. Here, n_{lmul}, $VLEN$, and VEW correspond to the logarithm of LMUL, the bit width of an individual vector register, and the vector element bit width, respectively (illustrated in the RVV case where LMUL equals 8 in Fig. 1); (ii) In cases of longer VLs, RVV often faces challenges managing tail data during strip-mining, where small-sized tail data can occupy RGs in a higher LMUL setup, leading to diminished performance.

3 ISA Extensions and Methodology

3.1 Zoozve Extension Instructions

To eliminate strip-mining and address the aforementioned vector register grouping drawbacks, Zoozve is specifically designed for high-performance vector processing with the following key principles: **(a)** The quantity and dimensions of vector registers can be arranged with flexibility, not limited to set values. **(b)** Having an adequate quantity of vector registers enables more data to be kept in close proximity to the execution units.

Zoozve vector instructions are broadly categorized into three types: vector load/store instructions, vector arithmetic and logical instructions, and vector control instructions using RISC-V custom opcode regions (custom-0/1/2). Fig.2(a) presents the typical vector arithmetic and logical instruction format. To support the efficient access of a large number of vector register groups, the **v_head** field (e.g., **vd_head**, **vs2_head**, and **vs1_head**) is used. This register field allows for the access of up to 2^{13} vector registers. It can be further expanded by using `vsetcsr` instruction to write extra bits into control and status registers (CSRs). The field **v_head** stores the starting addresses of the vector registers involved in the operation. The scalar register **rs_avl** holds the target vector length. With the starting addresses provided by **v_head** and the vector length specified by **rs_avl**, it becomes possible to efficiently access large register groups. The **rs2** field is utilized to hold the scalar operand, enabling efficient vector-scalar computations. Fig. 2(b) compares Zoozve and RVV in a reduction add kernel, highlighting Zoozve's reduced instruction count due to the removal of strip-mining.

Asymmetric instructions are specifically designed for vector data processing, exemplified by scatter and gather instructions. Zoozve allows different VLs between source and

Figure 2. (a) Zoozve vector arithmetic and logical instructions format. (b) Comparison of reduction add assembly in Zoozve and RVV. (c) Compilation workflow for Zoozve in LLVM.

destination vectors by using register-level gather or register scatter instructions, shown on the right of Fig. 1. For lengthening vectors, a scatter instruction can be used: vd[vs2[i]] ← vs1[i]. Conversely, to shorten a vector, a gather instruction can be applied: vd[i] ← vs1[vs2[i]]. Unlike the vrgather instruction in RVV, which maintains the same VL for both source and target registers, Zoozve allows the target vector's length to match the length of the input indices, rather than the input vector's length, which minimizes register waste. These asymmetric instructions maximize register efficiency when there is a substantial VL difference between source and destination registers.

Leveraging the extensive encoding space described above, an arbitrary, data-adaptive register grouping allocation strategy is proposed to eliminate strip-mining at both the software and compiler level. Specifically, Zoozve allows for a variable number of vector registers (V_0 to V_n), which can be flexibly configured based on specific application requirements. The RGs in Zoozve can be dynamically adjusted based on the types of instruction values. The determination of RG depends on two factors: the starting register number, RG_{head}, allocated by register allocation (RA) algorithms of the compiler, and the vector data type, defined as $RG_{type} = L \cdot VEW$, where L is the programming vector length. Data types are declared within the high-level programming language. At compile time, RGs are allocated by calculating the range from RG_{head} to $RG_{tail} = RG_{head} + RG_{type}/VLEN$.

3.2 Compilation Workflow

To provide compilation support for Zoozve, we design the compilation workflow shown in Fig. 2(c). Several LLVM

passes are implemented to remove the strip-mining assemblies. Below is a detailed step-by-step explanation of the compilation process.

Step 1: The implementation of the built-in functions for Zoozve in Clang provides programmers with an intuitive interface to utilize Zoozve operations efficiently. These built-in functions allow developers to explicitly specify the vector value type, which is leveraged in subsequent optimizations.

Step 2: Clang then transforms these built-in functions into intrinsics via intrinsic library mapping. Through static single assignment (SSA), variables in the high-level language are converted into virtual registers.

Step 3: To resolve the aforementioned compilation issue, the Zoozve intrinsic splitting pass is introduced. In this pass, the original intrinsic intermediate representation (IR) is transformed into a split form with a specific value type, ensuring that the range of virtual registers is clearly defined and effectively managed. The total number of split instances is determined by the split count. delimiter intrinsics are inserted before and after each split to guide the RA phase. These delimiters indicate which registers must be allocated consecutively to form a register group.

Step 4: Once the transformed IR is generated, it is utilized in the RA stage, where registers are assigned based on the lifespan of virtual registers as determined by live interval modification. The process operates between the lifetime analysis and RA, ensuring that virtual registers grouped by delimiter intrinsics are assigned consecutively. First, eligible registers are scanned, tracing back to locate the delimiter intrinsic and determine the *LMUL*. For each subsequent *LMUL* split intrinsic, it enforces the lifespan of the registers to that

Figure 3. Speedup of dynamic instruction counts and strip-mining iterations for RVV and Zoozve across different kernels: Top: `fft` processing of $f32$ elements from 32 to 2048 points. Middle: `dotproduct` of $f32$ elements from 512 to 16,384. Bottom: `axpy` operation on $f32$ elements from 512 to 16,384.

of the first intrinsic. Finally, all split virtual registers are enqueued for RA in the designated queue. The RA operates by performing a lifetime analysis of virtual registers, allowing virtual registers grouped by delimiter intrinsics to be assigned in a manner that preserves their spatial continuity.

Step 5: Following RA, the IR_{split} is translated into corresponding Zoozve assembly instructions (`Z_asms`) through the Zoozve assembly coalescing pass. During this pass, consecutive `Z_asms` are detected, where vector registers are consecutive and other parameters remain the same, and merged into a single, consolidated instruction, corresponding to the original built-in C function. This merging step optimizes instruction flow and minimizes redundancy, effectively ensuring that the final assembly sequence mirrors the efficiency of the original function.

4 Experiment and Evaluation

4.1 Experimental Setup

We conduct experiments to compare the performance of kernels executed using RVV and Zoozve, utilizing LLVM 15.6.0, the Spike simulator [17], and our customized Zoozve simulator. The benchmarks include `dotproduct` and `axpy` from OpenBLAS [12], as well as manually implemented `fft`

of various sizes, all of which are fundamental to scientific computing, signal processing, and machine learning. Our experiments analyze the impact of instruction count and scalable LMUL in Zoozve, with configurations featuring up to 2048 registers and LMUL values up to 1024. Furthermore, we implement a hardware proof-of-concept to validate the feasibility of the proposed ISA.

4.2 Kernel Comparison

Fig. 3 shows the speedup of Zoozve relative to RVV in dynamic instruction counts, along with the corresponding strip-mining iterations for three different kernels: `fft`, `dotproduct`, and `axpy`. `fft` demonstrates the significant advantages of Zoozve (LMUL=64) over RVV (LMUL=4) in terms of dynamic instruction count speedup and the number of strip-mining operations when processing FFT computations for data sizes ranging from 32 to 2048 points. Regardless of the data size, Zoozve consistently outperforms RVV in terms of dynamic instruction count. Even for smaller data sizes like 32 points, Zoozve achieves a 10.1× speedup, which increases to a 344.44× speedup when handling the larger 2048-point data size. Unlike the traditional divide-and-conquer method [4], Zoozve overcomes the LMUL constraints, enabling data calculation and permutation with minimal register spilling.

Benefits are also evident when executing operations such as `dotproduct` and `axpy` for $f32$ elements spanning from 512 to 16,384. Both kernels exhibit linear computational complexity of $O(n)$, typically involving only multiplications and additions without nested loops. In RVV, the dynamic instruction counts and strip-mining instances increase as the vector size grows due to limited register resources. Specifically, for `dotproduct`, the instruction count rises from 52 to 1292 and the number of strip-mining increases from 8 to 256 as the vector size increases. In contrast, Zoozve's larger registers and adjustable LMUL parameters result in a constant dynamic instruction count of 17, eliminating the need for strip-mining. For large data size, such as 16,384 $f32$ elements, the speedup achieved by Zoozve can reach up to 76×. A similar trend is observed in `axpy` where RVV's instruction count grows from 25 to 707 and strip-mining count rises from 2 to 64. Meanwhile, Zoozve maintains a constant dynamic instruction count of 12 in `axpy`, achieving a speedup of up to 58.92× when processing the same large data size.

4.3 Hardware Proof-of-Concept

Fig. 4 illustrates a possible hardware implementation for Zoozve. Building upon [2], two key components are introduced to support the Zoozve extension. In the control path, additional logic is incorporated to accommodate the flexibility of the RG and detect hazards between instructions. Comparators (CMPs) determine whether register indices fall within the range of RG_{head} and RG_{tail}, with their outputs OR'ed to generate a hazard signal. In the data path, a shuffle engine – comprising a crossbar and multiple processing

Figure 4. A hardware architecture supporting the Zoozve extension, with the additional components required for Zoozve shaded.

elements (PEs) – is implemented to handle inter-lane asymmetric operations, while lanes execute symmetric operations. Our design is synthesized using the SMIC 40nm process (400 MHz), yielding a 7.2 mm^2 synthesis area and 11.9 mm^2 layout area for a 64-lane, 1024-register configuration, with a negligible 5.2% area overhead.

5 Conclusion

This work presents Zoozve, a strip-mining-free RISC-V vector extension that tackles performance bottlenecks with arbitrary register grouping in ultra-long vector computation. Compiler optimizations, including intrinsic splitting and assembly coalescing, further enhance performance, achieving a 10.10× FFT instruction reduction with just a 5.2% area increase.

References

[1] Renzo Andri, Tomas Henriksson, and Luca Benini. 2020. Extending the RISC-V ISA for efficient RNN-based 5G radio resource management. In *57th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, Piscataway, NJ, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218496

[2] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. 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* 28, 2 (2019), 530–543. https://doi.org/10.1109/TVLSI.2019.2950087

[3] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, Piscataway, NJ, USA, 52–64. https://doi.org/10.1109/ISCA45697.2020.00016

[4] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301. https://doi.org/10.2307/2003354

[5] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE transactions on very large scale integration (VLSI) systems* 25, 10 (2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[6] John L Hennessy and David A Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, USA. https://dl.acm.org/doi/10.5555/1999263

[7] Hung-Ming Lai and Jenq-Kuen Lee. 2022. Efficient support of the scan vector model for RISC-V vector extension. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3547276.3548518

[8] Chris Lomont. 2011. Introduction to Intel Advanced Vector Extensions. *Intel white paper* 23 (2011), 1–21. https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf

[9] Marcia Sahaya Louis, Zahra Azad, Leila Delshadtehrani, Suyog Gupta, Pete Warden, Vijay Janapa Reddi, and Ajay Joshi. 2019. Towards deep learning using TensorFlow Lite on RISC-V. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, Vol. 1. 6. https://people.bu.edu/joshi/files/tflowlite-carrv-2019.pdf

[10] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, César Hernandez, Joan Cabre, Lina Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. 2023. Vitruvius+: An area-efficient RISC-V decoupled vector coprocessor for high performance computing applications. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–25. https://doi.org/10.1145/3575861

[11] NEC Corporation. 2018. SX-Aurora TSUBASA Architecture Guide Revision 1.1. [Online]. Available: https://sxauroratsubasa.sakura.ne.jp/documents/guide/pdfs/Aurora_ISA_guide.pdf.

[12] OpenMathLib. [n. d.]. OpenBLAS. GitHub. [Online]. Available: https://github.com/OpenMathLib/OpenBLAS.

[13] Matteo Perotti, Matheus Cavalcante, Renzo Andri, Lukas Cavigelli, and Luca Benini. 2024. Ara2: Exploring Single-and Multi-Core Vector Processing with an Efficient RVV 1.0 Compliant Open-Source Processor. *IEEE Trans. Comput.* 73, 7 (2024), 1822–1836. https://doi.org/10.1109/TC.2024.3388896

[14] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. 2022. A 'New Ara' for vector computing: An open source highly efficient RISC-V V 1.0 vector processor design. In *IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE Press, Piscataway, NJ, USA, 43–51. https://doi.org/10.1109/ASAP54787.2022.00017

[15] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. 2019. A performance analysis of vector length agnostic code. In *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE Press, Piscataway, NJ, USA, 159–164. https://doi.org/10.1109/HPCS48598.2019.9188238

[16] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2020. Stream semantic registers: A lightweight RISC-V ISA extension achieving full compute utilization in single-issue cores. *IEEE Trans. Comput.* 70, 2 (2020), 212–227. https://doi.org/10.1109/TC.2020.2987314

[17] Spike. [n. d.]. Spike: RISC-V ISA Simulator. GitHub. [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[18] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35