Backporting RISC-V Vector assembly

Joseph K. L. Lee^{1[0000-0002-1648-2740]}, Maurice Jamieson^{1[0000-0003-1626-4871]}, and Nick Brown^{1[0000-0003-2925-7275]}

EPCC, University of Edinburgh, Bayes Centre, 47 Potterrow, Edinburgh, United Kingdom {j.lee,m.jamieson,n.brown}@epcc.ed.ac.uk

Abstract. Leveraging vectorisation, the ability for a CPU to apply operations to multiple elements of data concurrently, is critical for high performance workloads. However, at the time of writing, commercially available physical RISC-V hardware that provides the RISC-V vector extension (RVV) only supports version 0.7.1, which is incompatible with the latest ratified version 1.0. The challenge is that upstream compiler toolchains, such as Clang, only target the ratified v1.0 and do not support the older v0.7.1. Because v1.0 is not compatible with v0.7.1, the only way to program vectorised code is to use a vendor-provided, older compiler. In this paper we introduce the rvv-rollback tool which translates assembly code generated by the compiler using vector extension v1.0 instructions to v0.7.1. We utilise this tool to compare vectorisation performance of the vendor-provided GNU 8.4 compiler (supports v0.7.1) against LLVM 15.0 (supports only v1.0), where we found that the LLVM compiler is capable of auto-vectorising more computational kernels, and delivers greater performance than GNU in most, but not all, cases. We also tested LLVM vectorisation with vector length agnostic and specific settings, and observed cases with significant difference in performance.

Keywords: RISC-V vector extension · HPC · Clang · RVV Rollback.

1 Introduction

Whilst the first proposal of the RISC-V vector extension (RVV) was introduced in June 2015, this was only ratified in late 2021. The goal of the vector extension is to be efficient and scalable, and the result is a Cray-style, variable sized vector model. RVV can reconfigure element size and vector length at run time, and is flexible so that it works on different data types such as integer, fixed-point and floating-point, and microarchitectures such as in-order, out-of-order and decoupled. When combined with the base ISA the total instruction count is around 300 instructions which is far fewer than typical packed-SIMD alternative, and fits into a standard fixed 32-bit encoded space [9]. RVV also forms the foundation for other vector extensions, such as the vector cryptographic extension.

Prior to ratification at version 1.0, a draft version 0.7.1 was released in 2019. According to this release: version 0.7 is intended to be stable enough to begin developing toolchains, functional simulators, and initial implementations, 2 J. K. L. Lee et al.

though will continue to evolve with minor changes and updates [4]. With the warning that backwards-incompatible changes will be made prior to ratification, toolchains, simulators, and hardware implementations were developed.

The first, and currently only, mass-produced hardware implementation of the vector extension v0.7.1 is the T-Head XuanTie C906 [3], which contains 128-bit wide vector registers and supports up to 32-bit vector elements. This is used in the low-cost, widely available Allwinner D1 SoC, which reuses their existing Arm SoC peripheral IP. As of yet, no commercially available hardware cores implementing v1.0 have been announced, only IP cores are available for soft-core designs. Since v0.7.1 was not ratified, upstream compilers and software do not, and will not, target this RVV version.

The aim of this paper is to address the gap between v1.0, the target for current and future tool development, and v0.7.1, the version supported by available hardware. This paper is structured as follows, in Section 2 we describe the background to this work by exploring the differences between v1.0 and v0.7.1 of RVV before surveying support in different toolchains and highlighting related work. Our *rvv-rollback* tool is then presented in Section 3 where we describe both the design and how this is to be leveraged within the compiler flow. Section 4 then undertakes benchmarking comparisons between different compilers using our tool to better understand the performance properties of common toolchains and setting, before drawing conclusions and discussing further work in Section 5.

The key contributions of this paper are:

- 1. We review the main differences between the ratified RVV v1.0 and implemented v0.7.1 by currently available hardware
- 2. We present our *rvv-rollback* tool designed for translating RVV v1.0 assembly code into v0.7.1
- 3. We utilise our *rvv-rollback* tool to test the auto-vectorisation of available compilers using the RAJA Performance Suite [6] and explore the impact that settings and compilers have on the overarching performance obtained.

2 Background and related work

2.1 RVV version 1.0 vs version 0.7.1

The RISC-V vector extension (RVV) adds 32 vector registers which are specified by two implementation-defined parameters, the maximum size in bits of a vector element, $ELEN \geq 8$, and the number of bits in a single vector register, $VLEN \leq 2^{16}$. RVV v0.7.1 adds five unprivileged Control and Status Registers (CSRs) vstart, vxsat, vxrm, vl and vtype, whereas v1.0 extends this list with two additional registers, vcsr (a vector control and status register) and vlenb (vector register length in bytes). Among other information, these CSRs contain settings about the selected element width SEW, vector register group multiplier LMUL (the number of vector registers grouped together), and operational vector length vl (the number of elements to be updated from a vector instruction).

Other important differences between RVV v1.0 and v0.7.1 include:

- Configuration-setting instructions: To update the vector type settings, configuration-setting instructions vsetvl and vsetvli have to be used, where the application specifies the element type and total number of elements to be processed. The hardware then configures the vl and vtype CSRs to match what is required by the application. RVV v1.0 introduced an extra instruction vsetivli, where the application can provide an immediate value directly as the application vector length, enabling more compact code to be generated by the compiler.
- Fractional LMUL: RVV allows multiple vector registers to be grouped together so that a single vector instruction can operate on multiple vector registers concurrently. This allows double-width, or larger, elements to be operated on with the same vector length as single-width elements. It is also possible for instructions to accept source and destination vector operands with differing element widths but the same number of elements, thus increasing flexibility. Vector register grouping can also improve the execution efficiency for longer application vectors because the hardware is then flexible enough to enable these to run concurrently.

The grouping is defined by the vector length multiplier LMUL which represents the default number of vector registers that are combined together to form a vector register group. Implementations must support LMUL of integer values 1, 2, 4, and 8. For v1.0, LMUL can also accept the fractional values $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$, which reduces the number of bits used in a single vector register. This is particularly useful when operating on mixed-width values, enabling the compiler to effectively increase the number of usable vector register groups.

- Tail/mask agnostic policy: Tail elements are those which lie past the current vector length, vl, setting. By contrast, inactive elements are those within the current vector length but are disabled by the current mask because they do not receive new results during a vector operation. For v0.7.1, all regular vector instructions place zeros in the tail elements of the destination vector register group, and inactive elements are undisturbed. For v1.0, these elements can be independently marked either undisturbed or agnostic. The agnostic setting allows for the corresponding destination elements to either retain their values or be overwritten with 1s, the pattern of which is not required to be deterministic when the instruction is executed with the same inputs. The agnostic policy was added in RVV v1.0 to increase efficiency when the inactive or tail values are not required for subsequent calculations. For v1.0 all configuration-setting instructions, *vsetvl*, *vsetvli* and *vsetivli* must include the flags for whether it is following the tail and mask agnostic (*ta* and *ma*) or undisturbed policies (*tu* and *mu*).
- Other changes: RVV v1.0 simplifies the mask register layout by mapping the mask bit for element i to bit i of the mask register. Furthermore, v1.0 also introduces several new instructions, such as vl1r which is a whole register load instruction, and also renames some instructions for example the vfredsum.vs has become vfredusum.vs. It should also be noted that because

4 J. K. L. Lee et al.

instruction encodings are different between v1.0 and v0.7.1 they are not binary compatible.

2.2 Toolchain support

The current upstream RISC-V GNU compiler toolchain does not provide support for any version of the vector extension. Whilst the GNU repository does contain an *rvv-next* branch [8] which aims to support v1.0, at the time of writing this is not actively maintained. There is also a previous, and now deleted, *rvv-0.7.1* branch which targeted v0.7.1. Because of this lack of GCC support, T-Head, who are the chip division of Alibaba, provides their own modified GNU compiler toolchain (XuanTie GCC) which has been optimised for their C906 processor. This bespoke compiler supports both RVV v0.7.1 and also their own custom extensions. Several versions of this compiler have been provided, and through experimentation we found that GCC8.4 as found in their 20210618 release (a mirror is available at the EPCC RISC-V testbed website [1]) provides the best auto-vectorisation capability and-so is used for benchmark comparisons in Section 4 because this version generates code specifically targeting 128-bit vector lengths.

By comparison, Clang 15, provided as part of LLVM supports RVV v1.0. Furthermore, programmers are able to target RVV assembly which is vector length agnostic via the flag *scalable-vectorization=on* or vector length specific via the flag *riscv-v-vector-bits-min=N* (where N is the fixed vector width in bits). Therefore it can be stated that, at the time of writing, Clang provides greater support for RVV than vanilla GCC.

In this paper, we use the RAJA Performance Suite [6] to test the autovectorisation performance across compilers for a range of loop-based computational kernels. We observed that T-Head's XuanTie GCC 8.4 is capable of vectorising fewer kernels than Clang 15 with either vector length agnostic (VLA) or vector length specific (VLS) settings. This can be seen in Table 3, which lists the kernels which are able to be auto-vectorised by the different compilers at different settings. This demonstrates the benefit of being able to leverage Clang on existing RISC-V hardware, and furthermore as further developments to main branch versions of these compilers will only support RVV v1.0, in future vectorisation will only be usable on existing vector hardware if the code are translated to RVV v0.7.1. This is the aim of the *rvv-rollback* tool we have developed and describe in this paper.

2.3 Related work

This paper aims to bridge the gap between compilers that are targeting RVV v1.0 and mass-produced physical hardware available for consumer purchase which only supports v0.7.1. [11] presents an upgrade of Ara, a vector co-processor design, and reviewed the differences between RVV v0.5 and v1.0 to study the design changes required for updating to v1.0. [7] studied the auto-vectorisation capability of Clang 15 for RVV via dynamic instruction counting, and identified

areas of improvements including the requirement to undertake improved shuffle pattern analysis and outer-loop vectorisation.

The RISC-V vector landscape and software ecosystem was surveyed in [10], exploring results from benchmarks running on T-Head's C906 using the XuanTie GCC 8.4 compiler. However, the authors were unable to include Clang in their benchmarking because of the RVV version issues that we are looking to address in this paper.

Vehave [5] developed by BSC is a runtime library that enables the execution of vector instructions conforming to RVV v0.7.1 on RISC-V CPUs which do not support the vector extension, effectively trapping the unknown instructions and emulating them in software. Whilst this approach provides the ability to simulate vector instructions, the trapping and execution in software is far slower than execution over hardware. By contrast, in our approach we directly modify the assembly code generated by the compiler to *roll back* the vectorisation to the v0.7.1 standard, thus enabling the code to run on the hardware directly and not requiring any runtime support.

3 The RVV rollback tool

We have developed *rvv-rollback*, a Python based tool that backports RVV v1.0 assembly code to v0.7.1 assembly, and this is available at [2]. Whilst this tool is capable of translating most v1.0 instructions into version v0.7.1, some lesser used features of v1.0 such as fractional LMUL are not yet supported.

RVV v1.0 introduced instructions which are immediate value versions of those found in v0.7.1. This is where the RVV instruction being issued contains part of the data being operated upon, such as a constant, rather than loading this from a register. Examples are the configuration set instruction *vsetivli*, and the whole register load/store vl1r and vs1r. By default, our tool will convert these instructions to first store the current vector configuration in memory, then reconfigure the vector settings, followed by performing the instruction itself, and finally restoring the setting from memory. However, this process adds some overhead and furthermore is often unnecessary because a temporary register can be used instead or the reconfigurations being issued by the compiler are simply redundant. In verbose mode, the tool will print out these instances and recommend alternative optimised configuration options, with the user then able to manually determine the appropriate translation.

It should be noted that this tool is aimed primarily to aid benchmarking applications, and whilst we have tested it extensively we make no guarantee as to bit reproducibility between the Clang generated RVV v1.0 assembly and our translated v0.7.1 code.

3.1 RVV rollback compiler workflow

In order to use our tool and generate RVV v0.7 executables using Clang, the user follows the following steps:

- 6 J. K. L. Lee et al.
- 1. Compile with Clang to obtain RVV v1.0 assembly code with the appropriate vector flags, for instance *-march=rv64gcv -O3 -mllvm -riscv-v-vector-bits-min=128* for VLS or *-scalable-vectorization=on* for VLA. The *-no-integrate-as* flag is also necessary as it directs the compiler to generate assembly which can be assembled by the GNU assembler in the third step. ¹
- 2. Translate the assembly code to RVV v0.7.1 using our *rvv-rollback.py* Python tool
- 3. Assemble the generated assembly code using T-Head's XuanTie GCC assembler, provided as part of v2.6.1 of the the Xuantie-900-gcc-linux toolchain (also available at [1]). This is required because a RVV v0.7.1 conforming compiler is needed to translate the v0.7.1 assembly into machine code.

It should be highlighted that those RAJA kernels which failed to automatically vectorise with T-Head's XuanTie GCC compiler detailed in Table 3 are due to limitations in the front-end of the GCC compiler, where automatic vectorisation opportunities are identified and applied, rather than the assembler. Consequently, whilst we leverage the GNU assembler as our third step it does not reduce opportunities for automatic vectorisation that have been identified by Clang higher up in the compilation process.

4 Benchmarking and Comparison

To demonstrate the use of our RVV rollback tool and the compilation workflow described in Section 3, we utilise the RAJA Performance Suite compiled using Clang 15 (generating RVV v1.0 assembly and backported to v0.7.1 using *rvv-rollback*) and XuanTie GCC 8.4 which natively generates an RVV v0.7.1 executable. The suite is compiled with single-precision floating point numbers (some double precision constants found within the code were manually converted to single precision). The compiler and relevant flags are listed in Table 1.

Name	Compiler	RVV Version	Compiler flags		
GCC8.4-scalar	XuanTie GCC 8.4	N/A	-03 -march=rv64gc -ffast-math		
GCC8.4-vector	XuanTie GCC 8.4	0.7	-03 -march=rv64gcv0p7 -ffast-math		
Clang15-scalar	Clang 15.0	N/A	march=rv64gc -O3 -ffast-math		
Clang15-vector-vls	Clang 15.0	1.0	-march=rv64gcv -O3 -mllvm		
			riscv-v-vector-bits-min=128		
			-ffast-math		
Clang15-vector-vla	Clang 15.0	1.0	-march=rv64gcv -O3 -mllvm		
			-scalable-vectorization=on -ffast-math		

Table 1. Compiler specifications

In this section we compare vectorisation performance of these benchmarks across the compilers on the Allwinnner D1. For reference, we also include result

¹ the **-save-temps** flag can be useful for saving all intermediate assembly files if this is desired by the programmer.

from the popular StarFive VisionFive V2 board (VF2), which contains a non-vectorised StarFive JH7110 processor (quad core SiFive U74). For these results benchmarks are run on a single core to provide a like-for-like comparison. The details of the systems we use in our experiments are reported in Table 2.

	Allwinner D1	StarFive JH7110 (VF2)	
Processor	XuanTie C906	SiFive U74	
Processor clock speed	$1.0 \mathrm{GHz}$	$1.5 \mathrm{GHz}$	
Cores	1	4	
Cache	32 KB I-cache + 32 KB D-cache	32KB I-cache + 32 KB D-cache + 2MB L2	
Memory	512MB DDR3	8GB DDR4	
ISA	RV64GC+V0.7	RV64GC	
Vector width	128bit	N/A	

 Table 2. Compute system specifications

4.1 Performance results

Table 3 lists the kernels which are able to be auto-vectorised by the different compilers at different settings. As mentioned, T-Head's XuanTie GCC 8.4 is capable of vectorising fewer kernels than Clang 15 with either vector length agnostic (VLA) or vector length specific (VLS) settings. Out of the kernels listed in Table 3, 22 were translated using the compiler workflow described in Section 3. For reporting performance comparisons in Figures 1a, 1b and 1c, we group kernels into three separate categories:

- 1. Those kernels not vectorised by T-Head's XuanTie GCC 8.4 compiler
- 2. Kernels vectorised by XuanTie GCC 8.4, but the kernel executed scalar code instead of vectorised code
- 3. Kernels vectorised by XuanTie GCC 8.4, and the vectorised code was executed

Figures 1a, 1b and 1c report the runtime for each kernel compiled using GCC 8.4 and Clang 15.0 with scalar and vector for the Allwinner D1, and scalar for VF2. All runtimes are averaged across three runs and normalised against scalar code compiled with GCC8.4 on the Allwinner D1.

There are a number of noticeable features and behaviours that can be highlighted in these figures. Firstly it can be seen that Clang is capable of vectorising more kernels than GCC, especially for the LCALS routines, and this provides a significant speedup as seen when comparing Clang and GCC results in Figures 1a and 1b. However, for some kernels such as INIT3, TRIDIAG_ELIM, and GESUMMV Clang's vectorised code is slower than its scalar counterpart.

Table 3. List of RAJA Performance Suite kernels auto-vectorised by XuanTie GCC 8.4 and Clang 15.0 compilers. * denotes kernels vectorised by GCC 8.4 but only scalar code was executed during runtime, and [†] denotes kernels vectorised by Clang (VLS and VLA) but only scalar code was executed during runtime.

Kornols	XuanTie	Clang15	Clang15
Kerneis	GCC8.4 vector	vector VLA	vector VLS
Algorithm: MEMCPY, MEMSET,			
REDUCE_SUM			
Apps: ENERGY, FIR, PRESSURE			
Basic: SAXPY, SAXPY_ATOMIC,	X	Х	X
REDUCE3_INT			
Lcals: FIRST_DIFF*, FIRST_SUM*,			
GEN_LIN_RECUR, HYDRO_1D*, HYDRO_2D*,			
TRIDIAG_ELIM*			
Polybench: $2MM^{\dagger}$, $3MM^{\dagger}$, ATAX, FDTD_2D,			
GEMM [†] , GEMVER, GESUMMV, JACOBI_1D*,			
JACOBI_2D*, MVT			
Stream: ADD, COPY, DOT, MUL, TRIAD			
Total: 30			
Apps: LTIMES, LTIMES_NOVIEW, VOL3D			
Basic: IF_QUAD, INDEXLIST_3LOOP,			
INIT_INIT_VIEW1D, INIT_VIEW1D_OFFSET,		Х	X
INIT3, MAT_MAT_SHARED, MULADDSUB,			
NESTED_INIT, PI_ATOMIC, PI_REDUCE,			
REDUCE_STRUCT, TRAP_INT			
Lcals: DIFF_PREDICT, EOS, INT_PREDICT			
Polybench: FLOYD_WARSHALL, HEAT_3D			
Total: 21			
Algorithm: SORT			
Apps CONVECTION3DPA, DEL_DOT_VEC_2D,			v
DIFFUSION3DPA, HALOEXCHANGE_FUSED,			A
MASS3DPA, NODAL_ACCUMULATION_3D			
Lcals: PLANCKIAN			
Total: 8			
Algorithm: SCAN			
Apps: HALOEXCHANGE			
Basic: INDEXLIST			
Lcals: FIRST_MIN			
Polybench: ADI			
Total: 5			



Fig. 1. Runtime for RAJA Performance Suite kernels normalised against Allwinner D1 with GCC8.4 scalar



(b) Kernels vectorised by GCC8.4-vector but only scalar code executed



(c) Kernels vectorised by GCC8.4-vector and vector code executed

For the 2MM, 3MM and GEMM matrix multiplication kernels whose performance is reported in Figure 1c it can be seen that Clang's vectorised performance exactly matches that of its scalar performance. This is because, whilst Clang was able to auto-vectorise the routines, the scalar code was executed, whereas by contrast for these benchmarks GCC executed its vectorised code and produces significantly faster runtimes. It can be seen therefore that whilst the auto-vectorisation reported in Table 3 demonstrates that on the whole Clang is able to vectorise more kernels than GCC, there are some exceptions to this rule.

Across most of the benchmark kernels Clang VLA (vector length agnostic) and VLS (vector length specific) settings provide very similar performance, except for specific kernels such as ATAX, FDTD_2D and GEMVER. This demonstrates that it is important to experiment with these different compiler settings as it can make a difference in some situations to the achieved performance.

When comparing the performance of non-vectorised, scalar, code execution, it can be seen that for almost all kernels Clang 15 and GCC8.4 provide very similar performance, often within around 10% of each other. However several kernels are an exception to this rule, for instance GCC is 52% faster for the EOS kernel,

29% faster for FIRST_DIFF and 15% faster for FIRST_SUM. When comparing scalar performance on the Allwinner D1 against a single core of the U74 which is in the VisionFive V2, it can be observed that the V2 is significantly faster for high arithmetic intensity kernels, such as GEMM, compared to the Allwinner D1 running either vector or scalar code. However, for most kernels the vectorised kernels running on the Allwinner D1 is comparable with, if not faster than, the U74. This is especially impressive considering that the Allwinner D1 is considerably cheaper than the VisionFive V2, although it should be highlighted that we are comparing single-core performance here and unlike the D1 the U74 contains four compute cores so would likely deliver greater performance in practice.

A more general observation across our benchmark kernels was that we found when it comes to the compiler determining whether to generate vectorised or scalar instructions for execution depends heavily on loop ranges, which both compilers tend to be very sensitive to. For example, for some kernels the vectorised code is run only when the loop range is divisible by 8, and this demonstrates that it is therefore crucial that users manually check whether vectorised code is being emitted by the compiler, and executed, after compilation in order to obtain best performance.

5 Conclusions, recommendations and future work

In this paper we have explored compiler toolchains that enable vectorisation on mass-produced, commodity available RISC-V physical hardware. Whilst there is no main branch version of GCC that supports RISC-V vectorisation, a bespoke version by T-Head based on GCC 8.4 does support v0.7.1. However, as illustrated in Table 3, it is less capable of automatic vectorisation compared to Clang 15. The challenge with Clang is that this only supports RVV v1.0 and-so we have introduced our tool, *rvv-rollback*, to backport the generated v1.0 assembly to v0.7.1

We have demonstrated that our tool runs across a wide set of benchmark codes, and the gathered performance numbers have illustrated that, in the main, vectorisation via Clang 15 is beneficial compared to T-Head's GCC 8.4 although there are always exceptions to this rule. Furthermore, we have demonstrated that whilst for most of our benchmark kernels the performance difference when compiling using VLA or VLS via Clang is narrow, for some codes it can make a more significant difference and-so it is important for programmers to experiment with these compiler flags.

One of the surprising aspects for us was that whilst the compiler will report that it has auto-vectorised code, it can sometimes revert to executing scalar-only code without the programmer knowing. Therefore it is crucial that programmers are aware of this and manually check what has been generated. One of our recommendations is that Clang should be clearer on this and also improve range checking to reduce the sensitivity around whether it picks one path or the other. Furthermore, effort should be invested into investigating why Clang is currently unable to execute auto-vectorised matrix multiplication operations. 12 J. K. L. Lee et al.

In terms of future work, at the time of writing Clang 16 was released just a couple of days ago. Whilst we do not anticipate that this will have any impact on our *rvv-rollback* tool, it will be interesting to explore whether the performance insights reported in Section 4 have changed at all due to this latest version.

6 Acknowledgement

The authors would like to thank the ExCALIBUR H&ES RISC-V testbed for access to compute resource and for funding this work.

References

- 1. ExCALIBUR H&ES RISC-V testbed, http://riscv.epcc.ed.ac.uk/
- 2. RISCVtestbed/rvv-rollback: Translate RISC-V Vector Assembly from v1.0 to v0.7, https://github.com/RISCVtestbed/rvv-rollback
- 3. T-Head C906, https://www.t-head.cn/product/c906?lang=en
- 4. RISC-V "V" Vector Extension 0.7.1 (2019), https://github.com/riscv/ riscv-v-spec/releases/tag/0.7.1
- 5. Vehave User Guide · Wiki · EPI-public / RISC-V Vector Environment · GitLab (Nov 2021), https://repo.hca.bsc.es/gitlab/epi-public/ risc-v-vector-simulation-environment/-/wikis/Vehave-User-Guide
- 6. Raja performance suite (Feb 2023), https://github.com/LLNL/RAJAPerf
- Adit, N., Sampson, A.: Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V. IEEE Micro 42(5), 41–48 (Sep 2022). https://doi.org/10.1109/MM.2022.3184867, conference Name: IEEE Micro
- 8. GNU, International, R.V.: Risc-v gnu compiler toolchain (rvv-next branch), https://github.com/riscv-collab/riscv-gnu-toolchain/tree/rvv-next
- 9. International, R.V.: Risc-v "v" extension 1.0, https://github.com/riscv/ riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf
- 10. Lee, J.K.L., Jamieson, M., Brown, N.: Test-driving RISC-V Vector hardware for HPC. Proceedings for the First International workshop on RISC-V for HPC (Mar 2023), under peer review
- Perotti, M., Cavalcante, M., Wistoff, N., Andri, R., Cavigelli, L., Benini, L.: A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. In: 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 43–51 (Jul 2022). https://doi.org/10.1109/ASAP54787.2022.00017, iSSN: 2160-052X