A Performance Analysis of Autovectorization on RVV RISC-V Boards

Lorenzo Carpentieri
Department of Computer Science
University of Salerno, Italy

Mohammad VazirPanah
Department of Computer Science
University of Salerno, Italy

Biagio Cosenza
Department of Computer Science
University of Salerno, Italy

***Abstract*—The RISC-V instruction set architecture has become increasingly popular due to its open source and extensible design, making it a competitive choice in high-performance computing and embedded systems. The RISC-V Vector extension (RVV) empowers RISC-V processors with length-agnostic vectorization capabilities, a critical feature for efficiently handling parallel processing demands across different hardware. Compiler support for autovectorization allows to generate vector instructions automatically without requiring any effort to programmers. Given the limited yet evolving compiler support for RVV, this paper offers an in-depth examination of autovectorization capabilities in GCC and LLVM, for RVV version 0.7 and 1.0. We evaluated the autovectorization performance of LLVM, LLVM-EPI and GCC compilers across 151 loops from the Test Suite for Vectorizing Compilers (TSVC) ans seven real-world applications on the All-Winner D1 and BananaPi-F3 boards, representing RISC-V vector hardware. Our study focuses on quantifying and comparing the level of vectorization each compiler achieves across a diverse range of vectorization patterns and workloads, providing insight into their strengths and limitations with respect to RISC-V RVV. Our findings highlight that the LLVM-19 compiler outperforms GCC-14 in 76 out of 151 loops, and its performance is more sensitive to the selection of vector length. Additionally, tuning the vector Length Multiplier (LMUL) parameter can lead to performance improvements of up to 3x, and leveraging knowledge of the vector length can further enhance LMUL optimization in compilers.**

***Index Terms*—Autovectorization, RISC-V, RISC-V Vector Extension, RVV**

I. Introduction

RISC-V is an open-standard Instruction Set Architecture (ISA) [1] that has gained significant attention in recent years due to its modular, extensible, and open-source design. RISC-V is built on the principles of Reduced Instruction Set Computing (RISC), offering a streamlined and flexible base architecture that can be extended with a variety of optional features. This flexibility has made RISC-V a popular choice in academia, research, and industry, driving innovations across a wide spectrum of computing applications—from embedded systems to High-Performance Computing (HPC). One of the key extensions of the RISC-V ISA is the RISC-V Vector Extension (RVV), designed to enhance the architecture's capabilities in parallel data processing. RVV supports both Vector Length Agnostic (VLA) and Vector Length Specific (VLS) programming models, allowing developers to write code that adapts to varying vector lengths [2]. This flexibility enhances the potential for performance gains across a wide range of applications, especially in HPC. While vectorization has become an essential optimization technique, compilers still face limitations in fully leveraging these opportunities on RISC-V architectures [3]. This is largely because compilers struggle to capture key context from the code, which limits their ability to perform effective autovectorization. This challenge is particularly relevant in the case of RISC-V with its RVV, where effective autovectorization could enhance computational performance [4]. Understanding how `VLA` and `VLS` programming models influence autovectorization on RISC-V can provide valuable insights for maximizing the utility of the RVV.

This study examines the autovectorization capabilities of two widely-used compilers—GCC and LLVM—with a focus of RISC-V RVV versions 0.7 and 1.0. Although RVV 1.0-compatible hardware has only recently become available, much of the current RISC-V hardware still supports the earlier RVV 0.7 version [5]. However, the latest compilers offer limited support for RVV 0.7, which constrains optimization on older hardware. Additionally, most of the existing research in this area has relied on simulations rather than real hardware evaluations, which limits the practical applicability of their findings [3]. To address this gap, our study utilizes real RISC-V RVV boards for testing, providing an analysis of compiler support and performance across both RVV versions.

By investigating how GCC and LLVM exploit RVV features to generate autovectorized code, we identify each compiler's strengths and limitations with both RVV 0.7 and 1.0. Our findings reveal the current state of RISC-V compiler support for autovectorization. The insights gained from this study are intended to guide future efforts in optimizing compiler capabilities for RISC-V, facilitating better performance across a broader spectrum of applications. The key contributions of this paper are as follows:

- **Pattern-Based Analysis of TSVC Loops on RISC-V Vector Boards:** We explore the autovectorization capabilities of LLVM, LLVM-EPI and GCC compilers across various versions on real RISC-V RVV hardware. This analysis focuses on how these compilers automatically generate vectorized code compliant with RVV 1.0 and 0.7 specification.
- **Evaluation of Autovectorization in Real-World Applications:** We evaluate the autovectorization performance of LLVM, LLVM-EPI and GCC compilers across six

real-world applications, tested on the AllWinner D1 and BananaPi-F3 RISC-V RVV boards.

- **Performance Optimization Through VLA, VLS, and with LMUL Tuning:** We highlight the performance impact of using `VLA` and `VLS` and optimizes vector Length Multiplier (`LMUL`) settings for various vectorization patterns. Furthermore, we show how the knowledge of vector length can affect the selection of `LMUL`.

II. RISC-V Architecture

RISC-V is an open-standard ISA developed to emphasize simplicity, flexibility, and modularity. One of the key strengths of RISC-V is optional extensions that can be added depending on the specific needs of the hardware or application. These extensions include support for single and double precision operations (F and D), atomic instructions (A), and, more recently, vector processing (V). Among these extensions, the RISC-V Vector Extension stands out as a critical advancement, designed to enhance data-parallel processing by allowing the execution of vectorized operations, as discussed in the following section.

A. RISC-V Vector Extension (RVV)

The RVV enables instructions to be applied to multiple data elements simultaneously, improving parallel processing capabilities. It defines a set of 32 vector registers that are each `VLEN` bits wide, where `VLEN` is a power-of-two greater than or equal to 128 (in the standard V extension). Vector registers can be interpreted as multiple 8/16/32/64 bit elements, and operated on accordingly as signed/unsigned integers or single/double precision floating point numbers depending on the hardware support. The RVV extension also includes control registers to configure the Standard Element Width (`SEW`) to determine the number of elements per vector, the Vector Length (`VL`), which defines the active vector length, and the `LMUL` to group registers for forming "longer vectors". A notable advantage of the RVV extension is the support for both Vector-Length Specific (`VLS`) and Vector-Length Agnostic (`VLA`), making it adaptable to a wide range of hardware configurations and efficiently utilizing available resources. The RVV specification has evolved through different versions, from version 0.7 to the ratified version 1.0, each enhancing its capabilities.

1) RVV 0.7: The 0.7 version of the RISC-V RVV was an early draft that introduced key concepts such as `VLA` execution, allowing instructions to operate on vectors of varying lengths based on hardware. However, many modern compilers no longer support RVV 0.7. Legacy compilers such as GCC 8.4, GCC 10.2, and LLVM-EPI are compatible with this version [5], while newer compilers focus on supporting the stable, ratified RVV 1.0.

2) RVV 1.0: This is the first official and stable release of the vector extension, building on the concepts introduced in version 0.7 with further refinements and optimizations. Version 1.0 introduces various changes in the vector instructions, resulting in binary incompatibility with the previous version. The major change relevant for performance is related to the support of fractional `LMUL` which reduces the number of bits used in a vector register. Fractional `LMUL` is used to increase the number of usable architectural registers when operating on mixed-width values by not requiring that larger-width vectors occupy multiple vector registers.

Modern compilers, including the latest versions of GCC and LLVM, generate code compliant with RVV 1.0 specification offering support for both manual vectorization through intrinsics and autovectorization.

III. Benchmark Methodology

To thoroughly assess the autovectorization capabilities of compilers, we adopted a dual approach that includes both synthetic benchmarks (TSVC) and real-world applications, ensuring a comprehensive evaluation across a wide range of vectorization patterns and practical workloads. In the following, we describe these approaches in detail.

Category	#Loops	Set 1	Set 2	Set 3	Set 4	Set 5
Control Flow	22	10	8	4	15	10
Control Loops	13	12	0	1	11	12
Crossing Thresholds	8	2	1	5	2	3
Equivalencing	5	3	1	1	5	5
Expansion	12	8	0	4	6	7
Global Data Flow	10	7	0	3	7	8
Indirect Addressing	7	5	0	2	4	7
Induction Variables	9	2	3	4	6	6
Linear Dependence	14	9	4	1	8	9
Loop Rerolling	4	0	4	0	3	2
Loop Restructuring	9	3	3	3	4	1
Node Splitting	6	2	0	4	1	2
Packing	3	1	0	2	0	0
Recurrences	3	1	0	2	0	0
Reductions	15	7	2	6	2	4
Searching	2	0	1	1	1	0
Statement Reordering	3	0	0	3	0	0
Symbolics	6	4	0	2	5	5
Sum	**151**	**76**	**27**	**48**	**80**	**81**

TABLE I: Number of loops in Set 1: (`GCC-14<LLVM-19`), Set 2: (`GCC-14>LLVM-19`), Set 3: (`GCC-14=LLVM-19`), Set 4 and 5: vectorized loops by GCC-14 and LLVM-19 compilers respectively.

A. Test Suite for Vectorizing Compilers (TSVC)

The Test Suite for Vectorizing Compilers (TSVC) builds on earlier work that used 100 Fortran loops to evaluate the effectiveness of autovectorizing compilers [6]. Additionally, TSVC2 includes 151 loops implemented in C/C++ categorized into 18 distinct groups [4] (Table I).

B. Real Applications

In addition to the synthetic benchmarks categorized in the TSVC loops, we evaluated seven real-world applications across diverse domains, as summarized in Table II. These benchmarks were selected for their relevance in different computational fields and their ability to highlight vectorization patterns. Each application belongs to a domain and demonstrates specific computational models and data-level parallelism (DLP) patterns, providing a comprehensive overview

TABLE II: Overview of applications and their characteristics

Application	Application Domain	Algorithmical Model	DLP Pattern	Size of Benchmark
Blackscholes	Financial Analysis	Dense Linear Algebra	Regular	16K entries
Jacobi-2D	Engineering	Dense Linear Algebra	Regular	256 x 256 grid, 100 iterations
Needleman-Wunsch	Bioinformatics/Genomics	Dynamic Programming	Irregular	Sequence length 2^{20}
Particlefilter	Medical Imaging	Structured Grids	Mix	128 x 128 x 16 grid, 4096 particles
Pathfinder	Grid Traversal	Dynamic Programming	Regular	2048 width, 256 iterations
Streamcluster	Data Mining	Dense Linear Algebra	Mix	8192 points, 128 dimensions
Axpy	Vector Operations	Dense Linear Algebra	Regular	Vector size 2^{20}

of vectorization challenges and opportunities on RISC-V systems.

In addition, these applications cover a range of algorithmic models, such as dynamic programming, structured grids, and linear algebra, offering a diverse set of workloads for performance evaluation. The selected benchmarks also vary in their DLP patterns, including regular, irregular, and mixed patterns most of which are derived from RISCV-Benchmark-Suite [7]. Table II provides an overview of the selected applications, highlighting their domains, algorithmic models, DLP patterns, and size of benchmark. These characteristics illustrate the diversity of workloads used to evaluate vectorization potential on RISC-V RVV systems.

IV. Experimental Evaluation

This section presents a performance analysis of autovectorization on RVV RISC-V boards for the GCC and LLVM compilers.

TABLE III: Compute System Specifications

Allwinner D1	
Processor	XuanTie C906, Single-core 64-bit, 1.0 GHz
Cache	32 KB I-cache, 32 KB D-cache
Memory	1 GB DDR3
ISA	RV64GCV (RVV 0.7)
Vector Width	128 bits
Banana Pi F3 (BPI-F3)	
Processor	SpacemiT K1, 8-core 64-bit RISC-V AI CPU **Cluster-0:** 4-core, 2.0 TOPS AI, 32 KB L1/core, 512 KB L2, 512 KB TCM **Cluster-1:** 4-core, 32 KB L1/core, 512 KB L2
Memory	4 GB LPDDR4
ISA	RV64GCVB (RVV 1.0)
Vector Width	256/128-bits x2

A. Experimental setup

1) Hardware: Table III provides the specifications of the compute systems used in this study, focusing on RISC-V boards with support for vector extension. We evaluated two different setups: the *Allwinner D1* and the *BananaPi-F3*. The *Allwinner D1* supports RVV 0.7 and features a single-core XuanTie C906 processor, making it a lower-power option suitable for examining basic vectorization capabilities in minimal setups. Additionally, the *BananaPi-F3* includes the SpacemiT K1 processor with RVV 1.0 support. These setups enable the analysis of autovectorization on two RISC-V boards with different RVV versions.

TABLE IV: Compiler Flags for Vectorization Enabled, Categorized by RVV Version

RVV Version	Compiler	Vectorization Enabled
RVV 0.7	GCC-8.4 / GCC-10.2	`-O3 -ftree-vectorize -march=rv64gcv0p7`
	LLVM-EPI	`-O3 -fvectorize -mepi -menable-experimental-extensions -march=rv64gcv0p7`
RVV 1.0	GCC-13 / GCC-14	`-O3 -ftree-vectorize -march=rv64gcv1p0`
	LLVM-16 / LLVM-19	`-O3 -fvectorize -march=rv64gcv1p0`
	LLVM-EPI	`-O3 -fvectorize -mepi -menable-experimental-extensions -march=rv64gcv1p0`
	GCC-14-VLS	`-O3 -ftree-vectorize -mrvv-vector-bits=zvl -march=rv64gcv_zvl256b`
	LLVM-19-VLS	`-O3 -fvectorize -mrvv-vector-bits=zvl -march=rv64gcv_zvl256b`

2) Compilers: Table IV provides a comprehensive overview of the compilers and compiler flags utilized in our experiments. The RVV 0.7 specification is not supported by the mainstream GCC and LLVM compilers. However, on the *Allwinner D1* board, vector instructions compatible with RVV 0.7 can be generated using the XuanTie GCC [8] or the LLVM-EPI compilers [9]. XuanTie GCC, a customized fork of the GNU compiler created by T-Head, is specifically designed for T-Head processors and supports the RVV 0.7 specification. The LLVM-EPI compiler also enables the generation of vector instructions for RVV 0.7. However, the compiler is built by default to work with `SEW` up to 64, which is not compatible with the *Allwinner D1* hardware. For experiments related to the updated RVV 1.0 standard, we employed the newer versions of GCC (14.2), LLVM (19.1.1) and LLVM-EPI (v. 2024-09-28), which fully support RVV 1.0.

3) Benchmarks setup: In order to evaluate the autovectorization capabilities of modern compilers, we followed a comprehensive analysis based on TSVC loops categories as we mentioned in Section III-A. To aggregate the results of each loop into categories, we followed the same formula specified by Siso et al. [4]. We compiled TSVC loops using different compilers with and without autovectorization (Table IV). The scalar code is compiled only using `-O3` and `-fno-vectorize` and `-fno-tree-vectorize` for GCC and LLVM respectively. For each loop t in TSVC we computed the *median time* of 5 runs for both the auto-

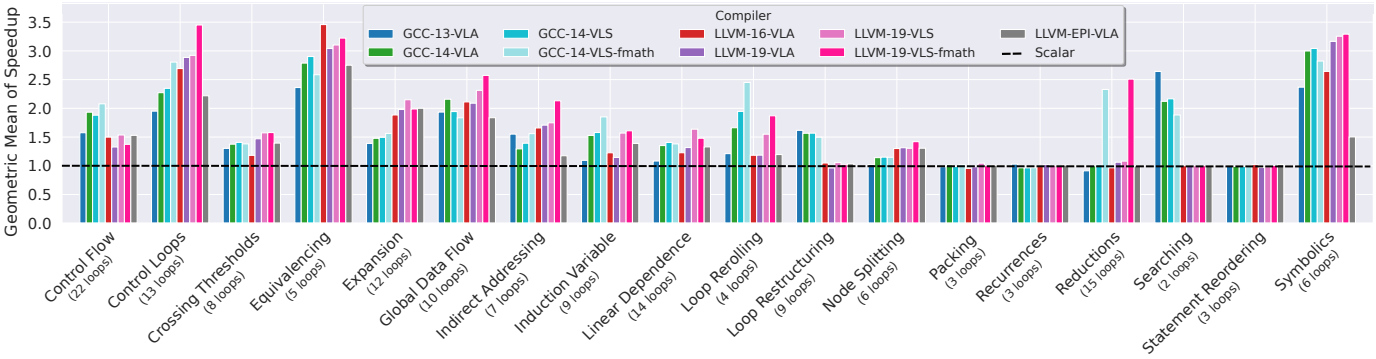

Fig. 1: Geometric mean of speedup achieved through autovectorization across different loop categories and compilers using RVV 1.0

vectorized and non-vectorized code. Then we calculated the speedup of the auto-vectorized version ($\overline{E}_t^{\text{vec}}$) using as a baseline the non-vectorized code (scalar) $\overline{E}_t^{\text{scalar}}$. For computing the speedup of codes autovectorized by a specific compiler, we considered the scalar version generated by the same compiler. The speedup of the loop t is defined as $\eta = \frac{\overline{E}_t^{\text{scalar}}}{\overline{E}_t^{\text{vec}}}$. As we mentioned all 151 loops were divided into 18 distinct categories. Therefore, for each category c, we aggregated the speedup of loops using the *geometric mean* $(\prod_{i=1}^{n} \eta_i)^{\frac{1}{n}}$, where n is the number of loops in each category c (Figure. 1 and 3). Finally, we computed the overall *geometric mean* for each compiler in the 18 categories to have a comprehensive comparison of the autovectorization support of each compiler (Figure. 7). Likewise, for real-word application we followed the same approach by computing the speedup of the autovectorized code using as baseline the scalar one. The input size of each benchmark is defined in Table II.

The following sections explore the compiler's autovectorization capabilities for RISC-V RVV 1.0 and 0.7, assessed using TSVC and real-world benchmarks.

B. Measuring Vectorization Performance of TSVC Loops

1) TSVC on RISC-V RVV 1.0: Figure 1 illustrates the autovectorization capabilities of the compilers listed in Table IV for each category of the TSVC benchmark for RVV 1.0. With regard to Figure 1, the principal findings can be summarized as follows.

```
int j= -1;

for(int i=0; i<N;i++){
 if(b[i]>0){
   j++;
   a[j]=b[i];
 }
}
```
Packing

```
for(int i=1;i<N;i++){
 a[i]=b[i-1]+c[i]*d[i];
 b[i]=a[i]+c[i]*e[i];
}
```
Recurrences

```
for(int i=1;i<N;i++){
 a[i]=b[i-1]+c[i]*d[i];
 b[i]=b[i+1]-e[i]*d[i];
}
```
Statement Reordering

```
float x;
int index;

for(int i=0;i<N;++i){
 if(a[i] > x) {
   x = a[i];
   index = i;
 }
}
```
Reduction

Fig. 2: TSVC patterns that are not vectorized by any compiler

Code not vectorized by any compiler: For loop patterns such as `Packing`, `Recurrences`, `Reductions`, and `Statement Reordering` (as shown in Figure 2), neither GCC nor LLVM could effectively vectorize the code on RISC-V.

In the `Packing` pattern, the compiler is unable to vectorize the code because the conditional statement introduces uncertainty about the control flow and the data access pattern associated with the variable `j`.

For the `Recurrences` pattern, vectorization is hindered by loop-carried dependencies: the value of `a[i]` depends on `b[i-1]` from the previous iteration, and `b[i]` relies on the value of `a[i]` calculated within the same iteration. These dependencies enforce sequential execution, making vectorization impossible.

The `Statement Reordering` pattern introduces additional write-after-read dependencies into the `Recurrences` pattern, complicating the compiler's ability to reorder computations to eliminate read-after-write and write-after-read dependencies, further obstructing vectorization.

Lastly, in the `Reduction` pattern without using `-fast-math` option most of the loops are not vectorized due to non associativity of floating point operations. Out of 15 reduction patterns, only two are vectorized by GCC 14, and four by LLVM 19 (Table I). The vectorized code might generate instructions where elements are aggregated in a different order than in the scalar version. This reordering can yield different numerical results, discouraging the compiler from vectorizing the code to maintain accuracy. Enabling `fast-math` optimizations, both GCC and LLVM compilers are able to vectorize loops in the Reduction category, leading to a speedup of up to 3x. The fast-math optimizations also enhances other categories, particularly those involving reduction operations, by increasing the overall speedups.

To further explore this limitation, we also evaluated these patterns on x86 architecture and observed the same behavior. This implies that these loop categories inherently represent a challenge for autovectorization, posing optimization difficulties across both x86 and RISC-V architectures.

Codes vectorized only by GCC: GCC demonstrates a notable speedup in patterns such as `Loop Restructuring` and `Searching`, indicating its advantage in handling these

types of loops compared to LLVM.

In `Loop Restructuring` 5 out of 9 loops are not vectorized by the GCC and LLVM 16 and 19 compilers (Table I). GCC is able to vectorize the other 4 loops, while LLVM 16 and 19 vectorize only one loop. LLVM-EPI fails to vectorize any loops.

```
for(int i=1;i<N;i++)
{
 a[i]+=b[i]*c[i];
 e[i]=e[i-1]*e[i-1];
 a[i]-=b[i]*c[i];
}
```

Listing 1: TSVC loop s222 (Loop Restructuring)

```
for(int i=0;i<N;++i){
 for(int j=1;j<N;j++){
  a[j][i]=
    a[j-1][i]+b[j][i];
 }
}
```

Listing 2: TSVC loop s231 (Loop Restructuring)

Listings 1 and 2 show the TSVC `s222` and `s231` loops that are not vectorized by LLVM. For `s222`, the LLVM compiler is unable to vectorize the code due to loop-carried dependencies in the middle part of the loop (line 4), while the GCC compiler separates the loop into two, enabling automatic vectorization.

In loop `s231` LLVM is not able to apply loop interchange due to the data dependencies between iterations (line 4) missing the opportunity to vectorize the code.

Listing 3 shows the loop `s331` from `Searching` pattern. LLVM is not able to vectorize the code while GCC achieves 4.5x speedup. The GCC compiler uses mask operation to vectorize the code, while LLVM is unable to deduce that `j` retains only the value for the last negative element. To help LLVM vectorize, all the indexes for which `a[i]<0` can be stored in a vector as defined in the commented code and then use the last element of the `indices` vector.

```
// std::vector<int> indices(N);
for(int i = 0; i < N; i++){
   if(a[i] < 0){
    // indices.push_back(i);
    j = i;
   }
  }
```

Listing 3: TSVC loop s331 (Searching)

Code vectorized only by LLVM: In the `Expansion` category, LLVM demonstrates a clear performance advantage over GCC, particularly in loop `s255` (Listing 4).

```
 for(int i = 0; i < N; i++){
  a[i] = (b[i] + x + y) * 0.333;
  y = x;
  x= b [i];
 }
```

Listing 4: TSVC loop s255 (Expansion)

In this loop, variables x and y introduce interdependency between consecutive iterations, meaning that x and y cannot be computed in parallel throughout the loop. Due to this loop-carried data dependence, GCC is unable to vectorize `s255`, whereas LLVM overcomes this limitation, resulting in a 7x speedup. Compared to GCC, LLVM identifies the dependencies introduced by `x` and `y` and separates the scalar portion of the code from the vectorizable part, allowing automatic vectorization. Furthermore, LLVM demonstrates better performance in the `Expansion` category, especially in loops `s252` and `s253`, achieving speedups ranging from 4x to 6x. In contrast, GCC only manages speedups of 1.5x to 2.5x due to a different selection of the `LMUL` parameter. While LLVM-EPI shows a slowdown compared to LLVM 16 and 19 for the `Expansion` category, on loops `s256` and `s257`, it performs more efficiently by avoiding unnecessary vectorization.

VLA vs VLS: For both compilers, specifying the vector length at compile time generates a more optimized code for `VLS`, compared to `VLA`. In `VLA` since the vector length is variable at runtime, the compiler has to generate more generic code to handle different possible lengths. This introduces overhead, as the compiler cannot optimize for a fixed length and may introduce run-time checks or masking to handle different vector lengths efficiently. Furthermore, knowing the vector length can affect the `LMUL` parameter tuning leading to drastic differences in performance (Section IV-C).

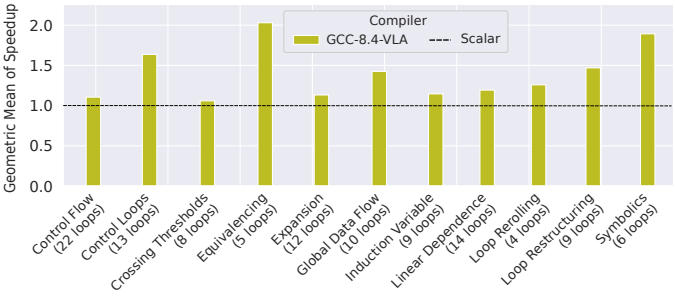

Fig. 3: Geometric mean of speedup achieved through auto-vectorization across different loop categories and compilers using RISC-V RVV 0.7

2) TSVC on RISC-V RVV 0.7: Figure 3 shows the TSVC categories that achieve some speedup on RISC-V RVV 0.7. We removed the GCC 10.2 compiler since it is unable to vectorize any of the loops. Additionally, the missing categories are not vectorized by GCC 8.4 either. The support for autovectorization for the RVV 0.7 extension is significantly poorer compared to the newer version 1.0.

C. Improving performance with LMUL

To further enhance flexibility and performance, RVV defines the `LMUL` parameter which specifies the size of a vector register group, allowing single vector instructions to operate on operands that span multiple registers.

RVV 0.7 supports integer `LMUL` values of 1, 2, 4, or 8, forming vector register groups consisting of 1, 2, 4, or 8 registers, respectively. Additionally, RVV 1.0 introduces support for fractional `LMUL` values: 1/2, 1/4, and 1/8. These fractional values reduce the effective size of the vector register, allowing more vector registers to be used simultaneously,

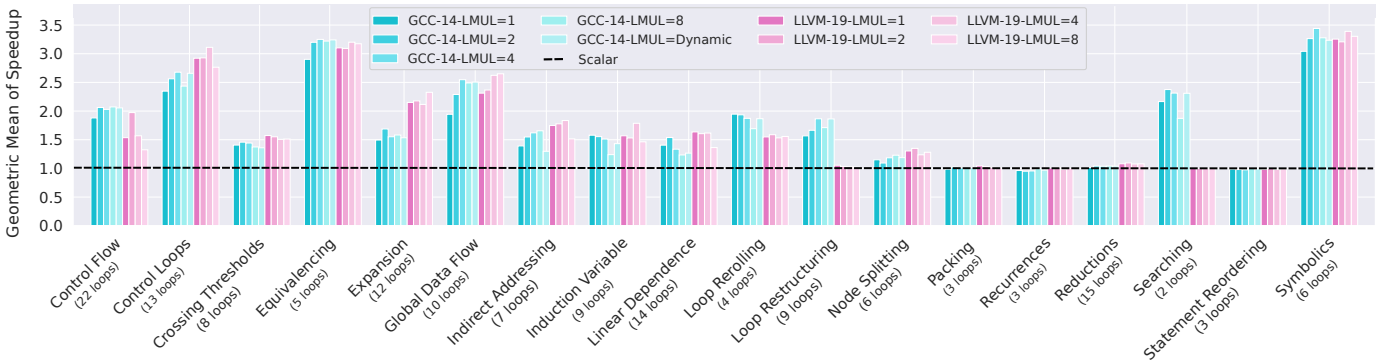

Fig. 4: Geometric mean of speedup achieved through autovectorization across TSVC loop categories, comparing performance for GCC-14 and LLVM-19 with various LMUL configurations with RVV 1.0

which can be beneficial in scenarios where register pressure is high, and smaller data sets are being processed.

Figure 4 shows the *geometric mean* of the speedup for each TSVC category as we increase the `LMUL` parameter from 1 to 8 for GCC-14 and LLVM-19. For both compilers, we can set the suggested `LMUL` as option using `-mrvv-max-lmul` and `-riscv-v-register-bit-width-lmul`. The GCC compiler benefits from higher `LMUL` values in 10 out of 18 TSVC categories while in LLVM only 5 categories show improvement. With higher `LMUL` values, each instruction can process a larger subset of data, reducing the number of instructions needed to handle a full data set. This leads to a more compact code for data-parallel sections. Furthermore, during loop strip mining, a higher `LMUL` reduces the number of iterations required, as more elements can be processed in each iteration, resulting in performance improvements. For `Induction Variables`, `Loop Rerolling` and `Node Splitting`, GCC does not benefit from increased `LMUL` due to high register pressure introduced by the use of `LMUL`. In fact, by increasing `LMUL` from 1 to 8 we also decrease the number of available registers from 32 to 4. Therefore, having fewer registers available can result in memory spilling as it becomes challenging to keep all variables in registers simultaneously. The same occurs in LLVM for the `Indirect Addressing` and `Linear Dependencies` categories. Other categories remain unaffected by `LMUL`, as they are not vectorized by any compiler.

Since fractional `LMUL` cannot be specified as a compiler option, we developed a small benchmark using vector intrinsics to demonstrate the effectiveness of fractional `LMUL`. The benchmark consists of a loop that runs a vector addition on 8-bit integers followed by add, sub, and multiply operations on `int64` array. Without fractional `LMUL` the compiler, in order to work on the same number of elements, needs to use `LMUL=1` for the first operation and `LMUL=8` for the others. However, using `LMUL=8`, only 4 registers are available. This compromises the performance due to high register pressure introduced by 64-bit operations.

Figure 5 shows the speedup of three versions of the benchmark: the `no-fraction LMUL` version sets the `LMUL` to

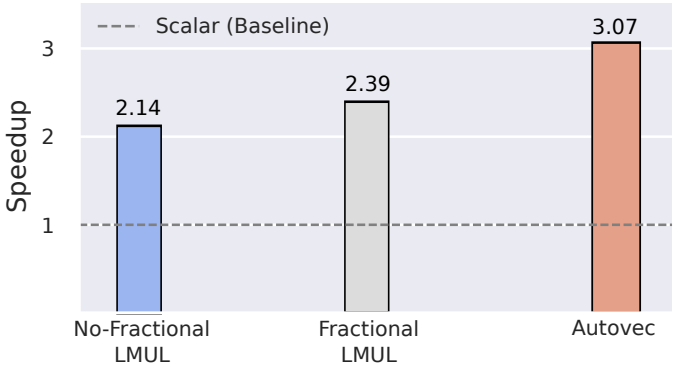

Fig. 5: Speedup of code with and without fractional LMUL

1 and 8; the `fractional LMUL` sets the `LMUL` parameter to 1/8 and 1, the `autovec` version, which automatically select the `LMUL` size. `Fractional LMUL` achieves a 2.39x speedup compared to the 2.14x speedup of the code that cannot take advantage of fractional `LMUL`. Furthermore, leveraging autovectorization, the compiler uses an `LMUL` of 1/4 and 2, resulting in even higher performance compared to the code implemented with intrinsics.

D. Measuring Vectorization Performance of Real Applications

To complement the evaluation, we also measured the performance of real-world applications with autovectorization enabled, as defined in Table IV.

Figure 6 shows the speedup achieved through autovectorization across six real applications and all compilers using RVV 0.7 and 1.0.

With RVV 0.7 only the simplest code `axpy`, achieves a 1.7x speedup due to vectorization, while for all other applications vectorization is not applied. The improvement achieved in `streamcluster` is not related to vectorization but only to the use of `-ffast-math`. These results validate the limited support for autovectorization found in RVV 0.7.

Looking at the RVV 1.0 results, for `needleman-wunsch` and `particelfilter` the compilers cannot autovectorize critical sections due to memory dependencies that are considered unsafe for vectorization, resulting in minimal speedup

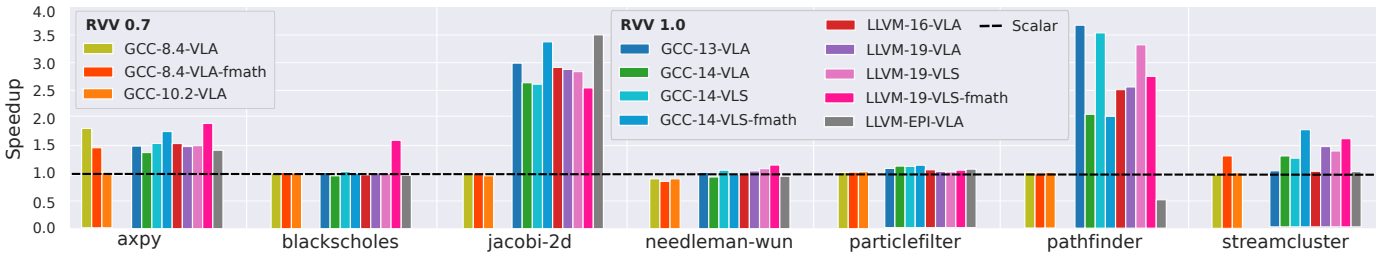

Fig. 6: Speedup achieved through autovectorization across real applications and compilers with using RVV 0.7 and 1.0

over scalar code. Although `blackscholes` is embarrassingly parallel, compilers are not able to vectorize the code due to difficulties in vectorizing math functions. LLVM-EPI achieves a speedup of 3.5x in `jacobi-2d` compared to the speedup of 3x in the other compilers, while for `pathfinder` experiences a slowdown, reducing performance to half of the scalar version. These results are justified by the different settings of the `LMUL` parameter between compilers that can drastically impact performance. In `jacobi-2d` using `LMUL=1`, LLVM-EPI is able to achieve higher performance, while in `pathfinder` selecting an `LMUL=1/2` the LLVM-EPI compiler results in a slowdown. With `VLS` configuration `pathfinder` achieves up to 3.5x speedup for both LLVM 19 and GCC 14 compared to 2.5x speedup of the `VLA` configuration. Specifying the vector length allows the compiler to select a better `LMUL` value, improving overall performance.

With `-ffast-math` enabled and the `VLS` configuration, LLVM 19 allows the vectorization of `blackscholes` application resulting in speedup up to 1.6x. The compiler unrolls math function to process them in scalar mode and then switches back to vector computation.

E. Comparative Discussion Across Compilers

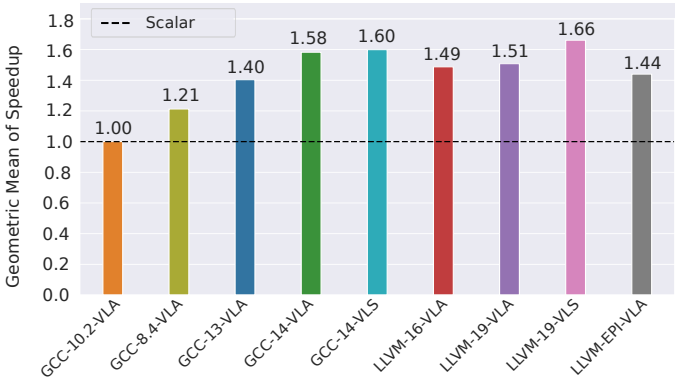

Fig. 7: Geometric mean of speedup across all categories TSVC loops and compilers

Figure 7 shows the aggregated *geometric mean* of speedup across all TSVC categories for each compiler. The compiler support for autovectorization is notably limited in RVV 0.7 compared to RVV 1.0. With GCC 10.2 code is not vectorized while with GCC 8.4, vectorization is applied only on simple patterns. In contrast, in RVV 1.0, the support for vectorization has significantly improved showing an overall speedup of up to 1.60x. Although the number of loops vectorized by GCC-14-VLS and LLVM-19-VLS are the same, the LLVM-19-VLS achieves better performance on 76 loops over 151 (Table I). These results are mostly related to a better selection of `LMUL` parameter. LLVM 19 is more influenced by vector length knowledge than GCC 14. In fact, LLVM 19 shows a consistent improvement with `VLS` across all categories. It achieves a 1.66x speedup compared to the 1.51x of `VLA`. Meanwhile, GCC 14 shows similar performance between `VLS` and `VLA`. GCC 14 shows a noticeable performance improvement over GCC 13. This suggests that the GCC 14 version has undergone optimizations that enhance performance. LLVM-EPI performance aligns closely with that of LLVM-19 in `VLA` mode.

In summary, LLVM 19 with `VLS` configuration achieves the highest performance among all compilers.

V. Related Work

Vectorization plays a critical role in enhancing the performance of applications across various fields, including HPC, machine learning, and multimedia processing. Compiler support for autovectorization [3], [10]–[14] is essential for realizing these performance gains. With the increasing adoption of `VLA` ISAs such as RISC-V RVV, recent studies focus on optimizing compiler support to fully leverage RISC-V vectorization across various hardware platforms [15]–[17]. Adit and Sampson [3] used the gem5 simulation to assess LLVM autovectorization performance for RISC-V RVV 1.0, examining both `VLA` and `VLS` configurations. Their study highlighted LLVM's limitations in handling dynamic vector lengths and shuffle patterns, comparing RVV's performance to fixed-length ISAs like AVX-512. Lin et al. [18] address the portability challenges in `VLA` programming by developing methods to adapt Arm SVE intrinsics to RVV. Ramírez et al. [7] extended gem5 to support RVV instruction, allowing customizable configurations for vector registers, memory ports, and lanes. They developed a benchmark suite of diverse HPC and embedded applications. Their findings show performance gains with vectorization, though compiler limitations impact complex data structures, highlighting the need for further compiler optimizations for RISC-V vector architectures. Lai et al. [19] presents enhancements to LLVM's autovectorization capabilities for linear recurrence programs on the RISC-V Vector RVV, focusing on scan operations to address loop-carried dependencies. Lee et al. [5] evaluated autovectorization

support of T-Head GCC-8.4 compiler compatible with RVV 0.7.1 on Allwinner D1 hardware. To facilitate a comparison between the T-Head GCC-8.4 and the LLVM compiler—which lacks support for RVV 0.7—they modified the assembly code generated by LLVM (originally compatible with RVV 1.0) to be compatible with RVV 0.7 [20]. Their study demonstrated not only vectorization speedups for HPC kernels, but also revealed limitations due to incomplete support in both tooling and hardware, including the lack of 64-bit element support and sensitivity to loop structures. Their findings emphasize the early development stage of the RVV ecosystem and the need for improved compatibility with present standards like RVV v1.0. Lin et al. [21] assess the performance of RVV on computer vision algorithms using intrinsic functions on the Xuantie C906 RISC-V. Their evaluation specifically examines the impact of using different integer `LMUL` settings showing 2.98x performance speedup against the OpenCV implementation. Shih et al. [22] developed a predictor within LLVM that automatically selects the optimal `LMUL` value to minimize register pressure.

Our work provides a comprehensive analysis of compiler autovectorization capabilities for RISC-V RVV 0.7 and 1.0 across different compiler versions and real hardware, showing the performance impact of `VLA` and `VLS` on the `LMUL` parameter tuning.

VI. Conclusion

In this paper we analyzed the autovectorization capabilities of modern compiler toolchains for RISC-V RVV 0.7 and 1.0. This study demonstrates significant improvements in autovectorization support for RVV 1.0 over RVV 0.7, with RVV 1.0 compilers providing better support for vectorizing diverse patterns. Furthermore, our findings highlight that the compiler handling of `LMUL` parameters is a critical factor to improve the performance of autovectorized code, as selecting optimal `LMUL` values minimizes the register pressure and improves parallel processing efficiency.

Acknowledgment

The project has received funding from the Italian Ministry of University and Research under PRIN 2022 grant No. 2022CC57PY (LibreRT project).

References

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[2] T. R.-V. Foundation, "RISC-V Vector Extension," https://github.com/riscv/riscv-v-spec, 2021.

[3] N. Adit and A. Sampson, "Performance left on the table: An evaluation of compiler autovectorization for risc-v," *IEEE Micro*, vol. 42, no. 5, pp. 41–48, 2022.

[4] S. Siso, W. Armour, and J. Thiyagalingam, "Evaluating auto-vectorizing compilers through objective withdrawal of useful information," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3356842

[5] J. K. L. Lee, M. Jamieson, N. Brown, and R. Jesus, "Test-driving risc-v vector hardware for hpc," in *High Performance Computing*, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 419–432.

[6] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: a test suite and results," in *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, 1988, pp. 98–105.

[7] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A risc-v simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020. [Online]. Available: https://doi.org/10.1145/3422667

[8] XUANTIE-RV, "xuantie-gnu-toolchain," https://github.com/XUANTIE-RV/xuantie-gnu-toolchain, accessed: 2024-10-30.

[9] R. Ferrer, "Llvm-epi repository," https://repo.hca.bsc.es/gitlab/rferrer/llvm-epi, 2024, accessed: [Insert Date Here].

[10] J. G. Feng, Y. P. He, and Q. M. Tao, "Evaluation of compilers' capability of automatic vectorization based on source code analysis," *Scientific Programming*, vol. 2021, no. 1, p. 3264624, 2021.

[11] N. Brown, M. Jamieson, J. Lee, and P. Wang, "Is risc-v ready for hpc prime-time: Evaluating the 64-core sophon sg2042 risc-v cpu," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1566–1574. [Online]. Available: https://doi.org/10.1145/3624062.3624234

[12] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink, "An evaluation of current simd programming models for c++," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, pp. 1–8.

[13] A. Pohl, B. Cosenza, and B. Juurlink, "Vectorization cost modeling for neon, avx and sve," *Performance Evaluation*, vol. 140, p. 102106, 2020.

[14] ——, "Portable cost modeling for auto-vectorizers," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 359–369.

[15] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 159–164.

[16] M. Perotti, S. Riedel, M. Cavalcante, and L. Benini, "Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[17] A. Lopoukhine, F. Ficarelli, C. Vasiladiotis, A. Lydike, J. V. Delm, A. Dutilleul, L. Benini, M. Verhelst, and T. Grosser, "A multi-level compiler backend for accelerated micro-kernels targeting risc-v isa extensions," in *Proceedings of the 2025 Conference on Compiler and Generator Optimization (CGO)*, 2025.

[18] J.-K. Lin, Y.-L. Yang, H.-M. Lai, and J.-K. Lee, "Rewriting and optimizing vector length agnostic intrinsics from arm sve to rvv," in *Workshop Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP Workshops '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 38–47. [Online]. Available: https://doi.org/10.1145/3677333.3678151

[19] H.-M. Lai, J.-K. Lee, and Y.-S. Hwang, "Enhancing llvm optimizations for linear recurrence programs on rvv," in *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ser. ICPP Workshops '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 79–87. [Online]. Available: https://doi.org/10.1145/3605731.3605904

[20] J. K. L. Lee, M. Jamieson, and N. Brown, "Backporting risc-v vector assembly," in *High Performance Computing*, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 433–443.

[21] R.-S. Li, P. Peng, Z.-Y. Shao, H. Jin, and R. Zheng, "Evaluating risc-v vector instruction set architecture extension with computer vision workloads," *Journal of Computer Science and Technology*, vol. 38, no. 4, pp. 807–820, 2023. [Online]. Available: https://www.sciopen.com/article/10.1007/s11390-023-1266-6

[22] M.-S. Shih, H.-M. Lai, C.-L. Lee, C.-K. Chen, and J.-K. Lee, "Register-pressure aware predicator for length multiplier of rvv," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3547276.3548513