

Portable SIMD Acceleration of Sparse Linear Algebra Libraries on RISC-V

Patricia Siwinska

Universitat Politècnica de València

Adrián Castelló

`adcastel@disca.upv.es`

Universitat Politècnica de València

Héctor Martínez

Universidad de Córdoba

Andrés E. Tomás

Universitat Politècnica de València

Enrique S. Quintana-Ortí

Universitat Politècnica de València

Research Article

Keywords: Single-Instruction Multiple-Data (SIMD) Parallelism, Vector Intrinsics, Google Highway, Sparse Linear Algebra, High Performance

Posted Date: October 14th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11082690/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Portable SIMD Acceleration of Sparse Linear Algebra Libraries on RISC-V

Patricia Siwinska · Adrián Castelló ·
Héctor Martínez · Andrés E. Tomás ·
Enrique S. Quintana-Ortí

the date of receipt and acceptance should be inserted later

Abstract In this paper, we investigate the use of Google Highway (GHW) to develop SIMD-accelerated, performance portable implementations for sparse linear algebra libraries. Concretely, we vectorize three key solver operators and the underlying kernels from three representative libraries which differ substantially in application domain, programming model, and/or hardware support –namely, Ginkgo, PETSc, and SLEPc– discussing the main encountered caveats. We experimentally evaluate the resulting performance benefits using a representative subset of the SuiteSparse Matrix Collection on a SpacemiT K1 processor, a commercial RISC-V chip. Our results characterize the trade-offs between portability, programmability, and performance, offering practical guidance for adopting GHW in production-quality sparse linear algebra software targeting emerging open-ISA hardware such as RISC-V.

Keywords Single-Instruction Multiple-Data (SIMD) Parallelism · Vector Intrinsics · Google Highway · Sparse Linear Algebra · High Performance

1 Introduction

Contemporary sparse linear algebra libraries, including Trilinos¹, Ginkgo², PETSc³, and SLEPc⁴, increasingly depend on SIMD (Single Instruction, Mul-

Patricia Siwinska, Adrián Castelló, Andrés E. Tomás, Enrique S. Quintana-Ortí
E-mail: {psiwins,adcastel,antodo,quintana}@disca.upv.es
Universitat Politècnica de València, Spain
Héctor Martínez
E-mail: el2mapeh@uco.es
Universidad de Córdoba, Spain

¹ <https://trilinos.github.io/>

² <https://ginkgo-project.github.io/>

³ <https://petsc.org/release/>

⁴ <https://slepc.upv.es/release/>

multiple Data) vectorization [1, 2] to achieve high performance across modern CPU architectures. Concretely, in kernels such as the sparse matrix-vector multiplication (SPMV), the dot product, AXPY-like operations, etc., which are central to iterative solvers for linear systems and eigenvalue problems, effective vectorization is essential to maximize throughput and memory efficiency.

Our interest on RISC-V stems from its status as an open, royalty-free Instruction Set Architecture (ISA). At the same time, we do not want our efforts to be confined to RISC-V alone. SIMD extensions vary across processor families and ISAs, and achieving performance portability for SIMD-accelerated sparse linear algebra libraries remains a challenging, yet paramount, goal. We seek a solution that can extend beyond RISC-V to other ISAs.

Google Highway (GHW)⁵ is designed to deliver exactly this kind of performance portability. As a portable, open-source C++ abstraction layer, it enables architecture-agnostic vectorized code close to native intrinsics in style, allowing solutions developed for RISC-V to be mapped onto other ISAs.

In previous work [3], we developed SIMD implementations of SPMV using GHW and compared them against compiler auto-vectorization and hardware-specific intrinsics on x86-64 AVX2, Arm NEON, Arm SVE2, and RISC-V RVV 1.0. Our results there suggest that GHW can serve as a foundation for enabling SIMD-accelerated, portable sparse linear algebra libraries.

In this paper, we build on this direction by investigating the use of GHW to develop SIMD-accelerated versions of three sparse linear algebra libraries: Ginkgo, PETSc, and SLEPc. These libraries differ substantially in scope and design, featuring a diversity that provides a broad testbed to expose both the difficulties involved in vectorizing sparse linear algebra kernels with GHW and the benefits that can be derived from doing so. In pursuing this goal, our paper makes the following contributions:

- We introduce SIMD acceleration with GHW through a simple, yet representative, kernel for linear algebra, and use it to compare GHW against the architecture-specific intrinsics of the RISC-V Vector (RVV) extension.
- We discuss the major caveats involved in vectorizing the target sparse linear algebra libraries, and in particular those arising from the use of GHW intrinsics.
- We vectorize three key solvers from the target libraries: BiCGStab [4] from Ginkgo and GMRES [4] from PETSc, both for the solution of sparse linear systems; and Krylov–Schur [5] from SLEPc for the solution of sparse eigenvalue problems. For the individual kernels underlying these operators, we manually develop vectorized implementations using GHW.
- We conduct an experimental evaluation of the benefits of SIMD acceleration with GHW intrinsics, using a representative subset of the SuiteSparse Matrix Collection [6], on a SpacemiT K1 processor, a commercial RISC-V chip integrated into the Banana Pi BPI-F3 board.

The rest of the paper is structured as follows. In Section 2, we expose the features, functionality and differences of the three sparse linear algebra

⁵ <https://github.com/google/highway>

libraries. In Section 3, we introduce GHW and discuss the aspects of exploiting SIMD-acceleration, via this framework, in the target libraries. In Section 4 we evaluate the acceleration attained via SIMD-vectorization on the Banana Pi F3 board. Finally, we close the paper with a list of remarks in Section 5.

Table 1: Comparison of Ginkgo, PETSc, and SLEPc.

Feature	Ginkgo	PETSc	SLEPc
Primary focus	Sparse linear systems on heterogeneous architectures	Linear, nonlinear, and time-dependent PDE solvers	Eigenvalue and singular value problems
Programming model	Modern C++ (object-oriented templates)	C with object-oriented design	Built on PETSc (C, object-oriented style)
Parallelism model	Node-level parallelism (multicore CPUs)	(GPUs, Distributed-memory hybrid (OpenMP/GPU)	+ Inherits PETSc (MPI-based distributed parallelism)
Heterogeneous support	Native multi-backend (CUDA, HIP, SYCL, OpenMP)	CUDA, GPU support via backends (CUDA, HIP, SYCL, Kokkos)	Inherits PETSc capabilities
Abstraction model	Linear operator (<code>LinOp</code>) abstraction	Matrix/vector solver/preconditioner objects	+ Extends PETSc abstractions for spectral problems
Solver types	Iterative solvers (CG, GMRES, BiCGStab, etc.)	+ direct (via external packages) + nonlinear solvers	Eigensolvers (Krylov-Schur, Arnoldi, JD, etc.)
Preconditioning	Built-in (Jacobi, ILU, AMG, etc.)	Extensive built-in (Hypre, MUMPS, SuperLU)	+ external Relies on PETSc preconditioners
Mixed precision	Native support and design focus	Supported, but less central design	Inherits PETSc capabilities
Runtime configurability	Limited (compile-time driven)	+ API-Extensive database	runtime options Same as PETSc
Scalability target	Single-node and accelerator-focused performance	Large-scale distributed systems	HPC Large-scale eigenproblems on HPC systems
Ecosystem integration	deal.II, MFEM, OpenFOAM	Broad HPC ecosystem (many simulation codes)	Integrated with PETSc-based applications
Extensibility	Modular, easy addition of kernels/backends	Highly extensible via interfaces and external packages	Extends PETSc for advanced spectral solvers
Typical use case	High-performance sparse solvers on GPUs/accelerators	Large-scale multiphysics simulations	Large-scale eigenvalue and SVD computations

2 Overview of Libraries for Sparse Linear Algebra

In this section, we summarize the main features and functionality of the three sparse linear algebra libraries targeted in our work:

- Ginkgo is a modern high-performance C++ library designed for the solution of sparse linear systems on heterogeneous architectures.
- PETSc (Portable, Extensible Toolkit for Scientific Computation) provides a comprehensive suite of data structures and solvers for the numerical solution of partial differential equations and related problems.
- SLEPc (Scalable Library for Eigenvalue Problem Computations) extends PETSc’s capabilities to eigenvalue and singular value problems.

Table 1 compares the design and target of the three libraries, and Table 2 highlights the main features and functionality that each library offers.

3 SIMD-Acceleration of Sparse LA Libraries using GHW

The pursuit of performance portability in SIMD-accelerated sparse linear algebra software is complicated by the substantial heterogeneity of vector extensions across today’s processor families and ISAs. Vector intrinsics have long

Table 2: Feature coverage of Ginkgo, PETSc, and SLEPc. A checkmark (✓) indicates availability.

Feature / Solver / Preconditioner	Description	Ginkgo	PETSc	SLEPc
Iterative Solvers				
Conjugate Gradient (CG)	Krylov solver for SPD systems	✓	✓	
Bi-Conjugate Gradient (BiCG)	Two coupled Krylov sequences for general systems	✓	✓	
BiCG Stabilized (BiCGSTAB)	Stabilized version of BiCG	✓	✓	
GMRES	Generalized minimal residual method	✓	✓	
Flexible CG (FCG)	Allows varying preconditioners between iterations	✓		
Iterative Refinement (IR)	Residual-correction outer loop, good for mixed precision	✓	✓	
MINRES	Solver for symmetric indefinite systems		✓	
TFQMR	Transpose-free quasi-minimal residual		✓	
Richardson	Stationary iterative method		✓	
Preconditioners				
Jacobi / Block-Jacobi	Diagonal or block-diagonal preconditioning	✓	✓	
SOR / SSOR	Relaxation-based preconditioners		✓	
Incomplete LU / IC	Incomplete factorization	✓	✓	
Parallel ILU / IC	GPU/multicore parallel ILU/IC	✓		
Algebraic Multigrid (AMG)	Multilevel preconditioner	✓	✓	
Additive Schwarz (ASM)	Domain decomposition		✓	
ISAI	Explicit sparse approximate inverse	✓		
External PCs (Hypre, MUMPS, SuperLU)	Interfaces to external libraries		✓	
Eigen/SVD Solvers				
Krylov-Schur	General eigenvalue solver			✓
Arnoldi	Non-symmetric eigenvalue solver			✓
Lanczos	Symmetric eigenvalue solver			✓
Jacobi-Davidson	Interior eigenvalues			✓
SVD	Singular value decomposition			✓
Polynomial Eigenproblems (PEP)	Eigenproblems of polynomial type			✓
Nonlinear Eigenproblems (NEP)	Nonlinear eigenproblems			✓
Advanced Features				
Mixed precision	Lower precision for speed with accuracy	✓	Partial	Partial
GPU / heterogeneous support	CUDA, HIP, SYCL, multicore CPUs	✓	Partial	Partial
Distributed MPI parallelism	Large-scale HPC		✓	✓
Runtime configurability	Solver options at runtime	Limited	Extensive	As PETSc

served as a middle ground between raw hardware control and higher-level programmability: embedded directly in languages such as C/C++, they expose vector operations to the programmer while still benefiting from type checking, readable syntax, and compiler-driven optimization passes. Their principal drawback, however, is that they remain fundamentally bound to the architecture and vector extension for which they were designed. As a result, library developers must write and maintain a separate implementation for every hardware target they wish to support, a burden that not only inflates code complexity but also stands as one of the main obstacles to building sparse linear algebra frameworks that are both portable and performant.

The solution adopted in our work, GHW, was designed specifically to address this obstacle. Distributed as an open-source C++ library, it offers a portable abstraction layer that lets developers express vectorized algorithms once and have them mapped onto a variety of ISAs, all while keeping a pro-

<pre> 1 // DOT product using RVV 1.0 2 3 vfloat64m1_t vx, vy, vacc, 4 vzero, vred; 5 // Initialize vector accumulator 6 vacc=fmv_v_f_f64m1(0.0, 1); 7 size_t i=0; while (i<n) { 8 // Set vector length 9 size_t vl=setvle64m1(n-i); 10 // Load vector chunks 11 vx=le64_v_f64m1(&x[i], vl); 12 vy=le64_v_f64m1(&y[i], vl); 13 // Fused multiply-accumulate 14 vacc=fmacc_vv_f64m1(vacc, 15 vx, vy, vl); 16 i+=vl; 17 } 18 // Reduce vector acc. to scalar 19 vzero=fmv_v_f_f64m1(0.0, 1); 20 vred=fredusum_vs_f64m1_f64m1(21 vacc, vzero, 1); 22 alpha=fmv_f_s_f64m1_f64(vred); </pre>	<pre> // DOT product using GHW const ScalableTag<double> d; Vec<decltype(d)> vx, vy, vacc; // Initialize vector accumulator vacc = Zero(d); size_t i=0; while (i<n) { // Load vector chunks vx=LoadN(d, &x[i], n-i); vy=LoadN(d, &y[i], n-i); // Fused multiply-accumulate vacc=MulAdd(vx, vy, vacc); i+=Lanes(d); } // Reduce vector acc. to scalar alpha=ReduceSum(d, vacc); </pre>
---	---

Fig. 1: Vector DOT product using RISC-V RVV 1.0 (left) and GHW (right).

programming style closely reminiscent of writing native intrinsics directly. Adopting such an abstraction, however, is not free of trade-offs: before integrating GHW into production-grade sparse linear algebra software stacks, it is important to carefully weigh the balance it strikes between portability, ease of programming, and the performance ultimately delivered.

In this section, we first provide a brief introduction to GHW intrinsics for SIMD acceleration. Integrating the GHW kernels into the target sparse LA libraries requires a considerable engineering effort that poses some critical decisions, which are discussed in the remainder of the section.

3.1 GHW

We next utilize a simple vector kernel to introduce parallel programming with vector intrinsics and highlight the main differences between GHW and RVV 1.0. Concretely, Figure 1 illustrates these differences using the DOT product of two vectors, $\alpha = x^T y$, a kernel from the Level-1 BLAS⁶ that is heavily utilized in sparse linear algebra [4, 7]. There we consider that the scalar α is contained in variable **alpha** and the starting addresses of the two vectors, both of dimension **n**, respectively, in **x**, **y**. The base data type is **double** (FP64) and the vector codes in the figure are formatted to ease their visual comparison.

We summarize the differences between the two implementations as follows:

- **Abstraction level:** RVV intrinsics expose RISC-V-specific types and functions directly (e.g., `vfloat64m1_t`), while GHW uses portable, architecture-agnostic types (e.g., `Vec<decltype(d)>`) that compile down to whatever ISA is targeted (RVV, AVX, NEON, etc.).

⁶ Basic Linear Algebra Subprograms: <https://www.netlib.org/blas/>

- **Naming conventions:** RVV uses verbose, encoded intrinsic names that embed type/width information (e.g., `vfmac_vv_f64m1`), whereas GHW uses short, generic function names (e.g., `MulAdd`) that read like ordinary C++ calls.
- **Vector length handling:** RVV requires explicitly querying and setting the vector length on each iteration with `vsetvl_e64m1(n-i)`, tracking it in a `size_t vl` variable threaded through every operation. GHW abstracts this away: `LoadN` handles the tail/remainder internally, and `Lanes(d)` returns the step size, eliminating the need for a manual VL variable.
- **Reduction step:** RVV requires an extra zero vector (`vzero`) and a call to `fredusum_vs_f64m1_f64m1`, followed by a separate scalar-extraction step (`vfmv_f_s_f64m1_f64`). GHW condenses this into a single call: `ReduceSum(d, vacc)`.
- **Load semantics:** RVV loads use explicit segment/width-coded names (`le64_v_f64m1`) with the vector length passed separately; GHW’s `LoadN` takes the descriptor `d`, pointer, and remaining count directly.
- **Type declaration:** RVV declares the vector type once per architecture width/LMUL combination; GHW derives the vector type generically via `Vec<decltype(d)>`, decoupling the code from a specific register grouping.

Overall philosophy, the RVV version is a low-level, hardware-tied API mirroring the ISA’s actual instructions, whereas the GHW version is a higher-level C++ wrapper designed for portability across vector ISAs, trading some explicit control for readability and portability.

3.2 Ginkgo

Several aspects of the library must be taken into account when applying GHW to vectorize the main computational kernels:

- Ginkgo employs an executor-based design to target different hardware backends without modifying the underlying algorithm code. Among them, since our interest lies in CPU execution, we focus on the **Reference** executor, which runs a simple, non-optimized sequential implementation used mainly for correctness and consistency checks, and the **OMP** executor, which targets multi-core CPUs via OpenMP.
- Ginkgo makes extensive use of C++ templates and lambda functions. While this design yields backend-portable code, it also complicates static analysis and manual vectorization: template instantiation can obscure the actual code path executed for a given type combination, and lambda-based kernels hinder the direct insertion of explicit vector intrinsics.
- To reduce code bloat and improve functionality, some Ginkgo kernels include conditional invariants within their innermost loop. This is inefficient in itself and, moreover, hinders vectorization.

To address these limitations, we ultimately opted to develop new, vectorization-friendly kernels for the target Ginkgo routines, rather than attempting to

retrofit vectorization onto the existing implementations. Specifically, we avoid the use of templates and lambda functions and eliminate conditional invariants from the innermost loops, resulting in kernels whose code structure is directly amenable to vectorization with GHW.

3.3 PETSc (and SLEPc)

In this section, we discuss the main features of PETSc in connection with the acceleration of the routines in this library via vectorization using GHW. As SLEPc basically follows the same design philosophy of PETSc, most of the comments below also apply to both libraries:

- PETSc uses an object-oriented interface albeit implemented in standard C. Concretely, PETSc routines operate with data structures for each linear algebra object: vectors, matrices, preconditioners, solvers, etc. These objects are black-box C data structures and all operations are performed through general routines.
- Each object in PETSc can have different implementations to choose from, for example a sparse matrix can be stored in CSR format or in SELL format. Similar to C++ virtual functions, the general routines use a pointer table to call the specific routine for the implementation requested. PETSc has an extensive set of program parameters so the implementation and most attributes of all objects can be set in execution time.
- PETSc supports real or complex arithmetic and double or single precision, but it can be compiled for only one combination. PETSc routines use macros for the main data type and conditional compilation could be required by particular optimizations.
- PETSc parallel code is organized such that optimizing the sequential version of an operation also benefits its parallel implementation. PETSc does not exploit multithreaded parallelism, but it supports distributed memory parallelism via MPI.

In summary, PETSc does not provide explicit vectorization but, for vector and dense matrix operations, it can benefit from these optimizations if supported by the underlying BLAS/LAPACK library. PETSc specific routines are implemented in portable C and carefully organized so they may be optimized extensively by modern compilers. Because of the design, vectorizing some of the self-contained C routines for sparse linear algebra using GHW was a simple yet time-consuming task necessary to extract optimal performance.

4 Performance Evaluation

In this section, we illustrate the benefits attained via the SIMD-acceleration of three representative numerical solvers from Ginkgo, PETSc and SLEPc.

4.1 Setup

The experiments presented in this work were conducted on a BananaPi F3 board, equipped with an eight-core SpaceMiT K1 processor running at 1.6 GHz. Each core features a 32-KB L1 data cache, and each of the two 4-core clusters shares a 512-KB L2 cache. Given the limited memory bandwidth of the platform, in order to prevent workload balancing effects from blurring the effect of vectorization, we restricted the experiments to a single core of the board.

The vector unit in the SpaceMiT K1 core follows the RVV 1.0 specification with a total vector length of 256 bits, implemented as two 128-bit lanes. All the experiments employ IEEE 754 FP64 arithmetic.

For the experimental evaluation of the SPMV kernels, we selected 159 sparse matrices from the SuiteSparse Matrix Collection [6]. The target matrices have dimensions exceeding 100,000 in both rows and columns and originate from a range of scientific and engineering applications, excluding graph-related problems. Each solver was executed repeatedly for at least 5 seconds, and the reported results represent values averaged over time.

The compiler was `gcc 13.2.0` (Ubuntu 13.2.0-4ubuntu3-bb2). The libraries were Ginkgo version 1.11.0, and PETSc/SLEPc version 3.22.3. From these libraries, we selected two Krylov-based, iterative solvers for linear systems and one for eigenproblems:

- BiCGStab (Ginkgo). For this solver, the maximum number of iterations was set to 100, and a non-block Jacobi preconditioner was employed, with a reduction factor of the residual set to 10^{-7} . The kernels that appear in the solver correspond to the application of the preconditioner, dot products, composed AXPY-like operations, and SPMV. For the selected configuration, the convergence tests translates to additional dot products to calculate vector norms. Ginkgo merges some of these kernels into three routines, named `Step_1`, `Step_2` and `Step_3`.
- GMRES (PETSc). This solver was configured to perform ten iterations with the Jacobi preconditioner using classical Gram-Schmidt orthogonalization with iterative refinement. The kernels used by the GMRES solver are single vector operations (`VecAXPY`, `VecScale` and `VecNorm`), multiple vector operations for the ortogonalization (`VecMAXPY` and `VecMDot`), the preconditioner application (`VecPointwiseMult`) and the sparse matrix multiplication (`MatMult`).
- Krylov-Schur solver (SLEPc). This solver was configured to perform ten iterations using classical Gram-Schmidt orthogonalization with iterative refinement. The kernels used by the Schur solver are basically the same as in GMRES, except for `VecPointwiseMult`.

4.2 Experimental evaluation

Figure 2 reports the acceleration achieved through the SIMD vectorization of the kernels and routines underlying Ginkgo’s BiCGStab solver, averaged across

the 159 selected benchmarks. For the SPMV kernel, we consider two vectorized implementations, based on either the CSR (Compressed Sparse Row) format [4] or the sliced ELLPACK (sELL-C) format [8,9]. Correspondingly, the plot also reports two global acceleration factors, “CSR-based Total” and “sELLc-based Total”, depending on which SPMV implementation is used.

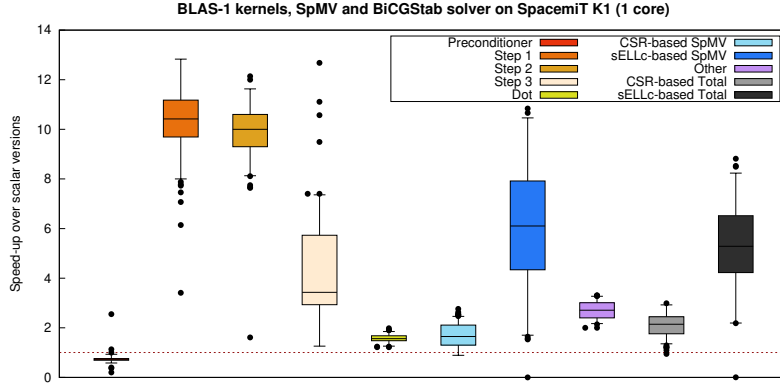


Fig. 2: Speed-up of the SIMD-accelerated BiCGStab solver in Ginkgo.

The analysis of the results in the figure leads to the following observations:

- Overall, most vectorized kernels achieve solid speed-ups over their scalar counterparts, although the magnitude varies substantially across kernel types, ranging from a slight slowdown (Preconditioner) to over $10\times$ (Step_1 and Step_2).
- The variability also differs considerably between kernels, suggesting that the speed-up is not uniform across the 159 benchmarks and is likely sensitive to the matrix structure and sparsity pattern.
- Step_1 and Step_2 show the highest speed-ups, with outliers reaching as high as $13\times$. This is mostly due to the elimination of severe inefficiencies present in Ginkgo’s routines for the OMP executor. Step_3 shows a lower and more dispersed speed-up.
- The dot product offers a comparatively modest and tightly clustered speed-up, consistent with this kernel being memory-bandwidth-bound, which limits how much vectorization alone can help.
- A comparison of the two SPMV implementations shows that the choice of data layout is paramount from a performance standpoint: CSR’s inherently irregular row lengths limit vectorization efficiency, whereas sELL-C’s padding and slicing better align with SIMD width requirements.
- Globally, the speed-up of the CSR-based BiCGStab solver is modest and closely tracks the CSR SPMV result. In contrast, the sELL-C-based BiCGStab solver achieves a considerably higher speed-up and also closely mirrors the

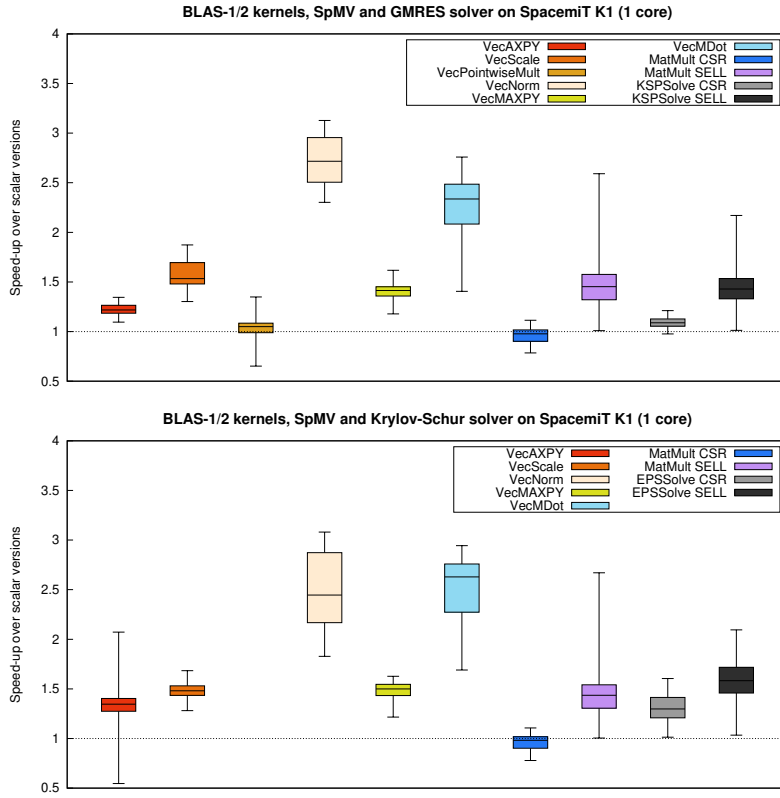


Fig. 3: Speed-up of the SIMD-accelerated GMRES solver in PETSc (KSPSolve, top) and the Krylov-Schur solver in SLEPc (EPSSolve, bottom).

sELL-C SPMV result, reinforcing that SPMV is the critical kernel determining overall BiCGStab performance.

Figure 3 reports the acceleration achieved through the SIMD vectorization of the PETSc GMRES solver and the SLEPc Krylov-Schur solver:

- Simple, BLAS1 vector operations achieve a modest acceleration except for the dot operation which obtain a significant benefit using vectorization. This is because the VecAXPY, VecScale and VecPointwiseMult kernel must write the resulting vector in memory. In contrast, the main loop in routine VecNorm is implemented via accumulation in a vector register, greatly improving the instruction throughput.
- More complex, multi-vector operations have a similar behaviour due to the same effect as in the single vector case. Specifically, VecMAXPY must write the resulting vector in memory, while VecMDot keeps the result in vector registers.

- Similar to Ginkgo, the performance of the sparse matrix product largely depends on the storage format. The vectorization of the kernel in CSR format does not offer any advantages while the vectorized SELL obtains a relevant acceleration over its scalar implementation.
- In the end, the complete GMRES solver (KSPSolve) and Krylov-Schur solver (EPSSolve) benefit of vectorization even if the sparse matrix is stored the CSR format.

5 Conclusions

This work investigated the use of Google Highway (GHW) to introduce SIMD acceleration into three sparse linear algebra libraries—Ginkgo, PETSc, and SLEPc—targeting the RISC-V Vector architecture. The results show that GHW provides a convenient and portable abstraction for vectorized kernels, although integrating it into production-quality libraries requires careful consideration of their code structure and abstractions.

Experiments on a single SpacemiT K1 core demonstrate that SIMD vectorization can provide substantial performance benefits, but the gains depend strongly on the kernel and sparse matrix format. In particular, sELL-C generally enables greater acceleration than CSR for SpMV, while operations that keep intermediate results in registers benefit more than memory-intensive vector updates. Nevertheless, the complete GMRES and Krylov-Schur solvers benefit from vectorization even with CSR storage. Overall, GHW provides a promising basis for portable SIMD acceleration of sparse linear algebra on RISC-V and other vector architectures. The results also emphasize that achieving high performance requires not only portable vector intrinsics, but also appropriate data layouts, kernel organization, and memory-aware implementations.

Acknowledgments. This work received funding from projects PID2023-146569NB-C2 of MCIN/AEI/10.13039/501100011033, and CIPROM/2022/20 and CIGE/2026/056 of the Generalitat Valenciana; the TUM Campus Heilbronn Incentive Fund "Preparing Ginkgo for RISC-V Technologies; and the DARE SGA1 Project, from the European High-Performance Computing Joint Undertaking (JU) under Grant Agreement No 101202459 and from PCI2024-161687-3 Project funded by MICIU/AEI/10.13039/501100011033 and European's Union "NextGenerationEU"/PRTR". The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, Czechia, Italy, Netherlands, Belgium, Finland, Greece, Croatia, Portugal, Poland, Sweden, France and Austria.

References

1. M. J. Flynn, "Some computer organizations and their effectiveness," in *IEEE Transactions on Computers*, 1972, vol. C-21, no. 9, pp. 948–960.

2. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.
3. P. Siwinska, A. Castelló, H. Martínez, A. E. Tomás, and E. S. Quintana-Ortí, “Highway for sparse linear algebra? performance and portability across cases,” in *16th Int. Conf. Parallel Processing & Applied Mathematics – PPAM*, 2026, to appear.
4. Y. Saad, *Iterative methods for sparse linear systems*, 3rd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.
5. G. W. Stewart, “A Krylov–Schur algorithm for large eigenproblems,” *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 3, pp. 601–614, 2002.
6. T. A. Davis and Y. Hu, “The SuiteSparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
7. Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, revised ed. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2011.
8. N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proc. Conf. on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 18:1–18:11.
9. M. Kreutzer et al., “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units,” *SIAM J. Scientific Computing*, vol. 36, no. 5, p. C408–C432, 2014.