RISC-V SUMMIT
NORTH AMERICA 2025

Moving to RISC-V Vector

- A Practical Journey of AI Operator Optimization

Guodong Xu,
RISCstar Solutions

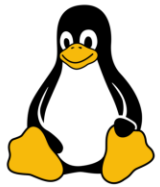
RISCSTAR™ SOLUTIONS

Who Am I ?

GUODONG XU

Chief Engineer,
Director of China Operations

RISCstar Solutions



Linux

Intel® ISA-L Library

BLIS

Agenda & Goals

Let's explore **RISC-V Vector (RVV)**'s practical application and performance by analyzing its implementation in two key open-source libraries: **OpenBLAS** (linear) and **SLEEF** (nonlinear).

- **Part 1: Analyze** the design logic for RVV in contrast to its Original Vector Code counterpart.
- **Part 2: Compare** RVV implementations side-by-side with code snippets.
- **Part 3: Demonstrate** vectorization speedup (Vector vs. Scalar) on RVV.
- **Part 4: Share** practical tooling insights using the *perf* utility.

A Note on Performance: We will focus on the *speedup ratio* within each architecture. Direct cross-ISA performance is not compared due to differing hardware settings

Performance Measurement Environment & Toolchain

RISC-V SUMMIT NORTH AMERICA 2025

Software Platform

fedora remix

Fedora 42 Remix

Released by **Fedora-V Force** team

[Download Link: images.fedoravforce.org]

Hardware Platform

K1 SoC (RISC-V)

Manufactured by **SpacemiT**

RISCstar Toolchain

RISCSTAR SOLUTIONS

Built by experts with cross-architecture fluency, our toolchain is finely tuned for maximum performance on RISC-V Vector hardware.

Explore Now:

https://riscstar.com/toolchain/

Special thanks to **Mr. Haibin Liu** from the Fedora-V Force team, who provided invaluable support in conducting the performance measurement in OpenBLAS and using Perf.

PART I

RISC-V Vector: Considerations when Porting

Key Architectural Design Contrasts: RISC-V Vector vs. Source Platform

RISC-V SUMMIT NORTH AMERICA 2025

Vector Length Control

Source Platform (Reference)

Fixed at runtime (Hardware-Defined). Software discovers and adapts but cannot dynamically adjust.

RISC-V Vector (Advantage)

Dynamically Configurable: Software controls VL via the `vsetvl` instruction, allowing an optimal match between algorithm and hardware.

Register Grouping (

Source Platform (Reference)

Fixed Register View: Lacks flexibility for logical grouping of vector registers.

Visual: Register Element Layout (e32m2 - 2 registers grouped)

VLEN=128: VLMAX=8, vl=8

Register 0 (4 element capacity):

0	1	2	3

Register 1 (4 element capacity):

4	5	6	7

Total: 8 elements (256 bits: 2 registers × 128 bits)

VLEN=256 (K1): VLMAX=16, vl=8 (constrained by AVL)

Register 0 (8 element capacity):

0	1	2	3	4	5	6	7

Register 1 (8 element capacity):

.	.	.	.	.	.	.	.

3 elements processed (vl) | 8 available but not used

Total: 512 bits (2 registers × 256 bits)

ves developers complete
ntrol over the execution
gth, which is essential for
rsuing maximum manual
rformance optimization.

nstruction

512 bits of data per operation (2× wider)

LMUL=4 (4 registers grouped): 4× data per instruction

- v0 (256 bits)
- v1 (256 bits)
- v2 (256 bits)
- v3 (256 bits)

1024 bits of data per operation (4× wider)

Understanding Application Vector Length (AVL)

RISC-V vectors are **VLEN-agnostic** through the `vsetvl` instruction, which takes an **Application Vector Length (AVL)** — the number of elements the software requests to process. The hardware returns the actual vector length (vl) based on this request and its capabilities.

How vsetvl Works:

Software Request

`AVL = 8`

elements needed

→

vsetvl Calculation

`VLMAX = (LMUL×VLEN)/SEW`

`vl = min(AVL, VLMAX)`

→

Hardware Returns

`vl`

actual length

- ✓ Vector operations process exactly `vl` elements
- ✓ Same code adapts to different VLEN implementations

Code Comparison: VL-Agnostic Philosophy

RISC-V SUMMIT NORTH AMERICA 2025

Step 1: Loop Control

- **Loop control**: svcntw() vs returned vl.
- **Guarding work**: predicate operand bundles (SVE) vs masked vector results using explicit vl (RVV masks only when needed).
- **Code impact**:

  - SVE keeps vector math “clean”, but demands predicated variants (svmla_f32_m);

  - RVV keeps operations non-predicated but requires explicit length plumbing (__riscv_vfmacc_vv_f32m1).

Source Platform / Original Vector Code (predicate-driven)

- VL discovery is implicit. Loop advances using svcntw().
- Per-iteration predicate pg = svwhilelt_b32(i, n) drives every instruction.
- Early exits (denormals, special cases) use svptest_any + svcompact to segregate active lanes.

```
for (int i = 0; i < n; i += svcntw()) {
  svbool_t     = svwhilelt_b32(i, n);
  svfloat32_t x = svld1(pg, &input[i]);

  // polynomial under pg svmla_f32_m, etc.
  svst1(pg, &output[i], acc);
}
```

RISC-V RVV (length-driven)

- Loop front-loads vl = __riscv_vsetvl_e32m1(n - i).
- vl controls both loop increment and each RVV intrinsic.
- Tail shrinks naturally via smaller vl; no predicates, but special handling uses vmslt_vx_ + __riscv_vcompress.

```
for (size_t i = 0; i < n; i += vl) {
  size_t     = __riscv_vsetvl_e32m1(n - i);
  vfloat32m1_t x = __riscv_vle32_v_f32m1(&input[i], vl);

  // polynomial using __riscv_vfmacc_vv_f32m1, etc.
  __riscv_vse32_v_f32m1(&output[i], acc, vl);
}
```

The core shift is from a **predicate-driven** model to an **explicitly length-driven** model.

Practical Guide: Key Porting Steps & Common Pitfalls

Step 2: Key Intrinsic & Concept Mapping

, where the list can be very long !

Operation	Original Vector Intrinsics	RISC-V RVV Equivalent
Vector Load	svld1(pg, &ptr[i])	__riscv_vle32_v_f32m1(&ptr[i], vl)
FMA	svmla_f32_m(pg, acc, x, h)	__riscv_vfmacc_vv_f32m1(acc, x, h, vl)

Horizontal Sum	float result = svaddv_f32(svptrue_b32(), acc_vec);	v_res = __riscv_vfredusum_vs_f32m8_f32m1(acc, v_res, vl);
		
		// Extract scalar result
		float result = __riscv_vfmv_f_s_f32m1_f32(v_res);

Segmented Load / Structured Load	svld3_u8(pg, base) (up to 4 fields)	vlseg3e8.v v8, (rs1), vm (up to 8 fields)

Recommended Resource for Deep Dive

For a comprehensive guide on intrinsic mapping and porting methodology, consider the upcoming **NEW COURSE: Porting Software to RISC-V (LFD 114)**. This training is developed to provide the cross-architecture fluency needed for real-world application porting.

RISCSTAR SOLUTIONS

RISC-V

THE LINUX FOUNDATION

PART II

AI Inference Operators

AI Inference Operators

Our analysis focuses on the core bottleneck of **LLM Inference**: the **Transformer Block**. Its workload can be broken down into two main operator categories:

Linear Operators (~90% of Compute)

Performance Attribution: Primarily dependent on high-performance linear algebra libraries like `OpenBLAS` / `BLIS`.

Prefill Phase (Compute-Bound)

Dominated by **GEMM**: `[seqlen, hidden_dim]`

Decode Phase (Memory-Bound)

Dominated by **GEMV**: `[1, hidden_dim]`

Non-Linear Operators

Performance Attribution: Vectorization optimization primarily depends on high-performance math libraries like `SLEEF` / `LibM`.

Key Operators

GELU | LayerNorm | Softmax

Mathematical Essence

`exp` `tanh` `sqrt` `Reduction/Sum`

PART III

Linear Operator Deep Dive (OpenBLAS)

The OpenBLAS RVV Strategy:
"Fixed Small VL" for Cache Efficiency

The OpenBLAS implementation for RVV reveals a critical engineering trade-off that prioritizes cache efficiency for compute-bound tasks.

- **BLAS1/2 (e.g., GEMV, axpy):**
  - **Strategy:** Dynamic VL.
  - **Implementation:** Uses vsetvl to set VL to the number of remaining elements, making full use of the hardware's VLEN.
  - **Use Case:** Ideal for **Memory-Bound** operations to maximize memory bandwidth.
- **BLAS3 (MatMul Micro-kernels):**
  - **Strategy:** Intentionally uses a **Fixed Small VL** (e.g., VL=4 or 8).
  - **Implementation: DYNAMIC_ARCH** is used to compile specialized kernels for targets like ZVL128B and ZVL256B. Even on hardware with a wider VLEN, the kernel intentionally constrains VL.
  - **Engineering Goal:** To **ensure the micro-kernel's tile size perfectly matches the L1 Cache**, sacrificing VLEN utilization for maximum data locality. This is the key to performance in **Compute-Bound** tasks.

Code Comparison:
SGEMM Core Computation (K-Loop Inner Body)

RISC-V SUMMIT NORTH AMERICA 2025

Source Platform Original Vector Code (predicate-driven)

kernel/arm64/sgemm_kernel_sve_v1x8.S (874 lines)

```
; Software-pipelined, 2x unrolled K-loop
.macro KERNELv1x8_M1
    ld1w   z1.s, p1/z, [pA]          ; Load A[k+1] for next iter
    add    pA, pA, lanes, lsl #2     ; Advance A pointer

    ; Compute with A[k] (z0), load B[k+1] for next iter
    fmla   z16.s, p1/m, z0.s, z8.s   ; acc0 += A[k] * B[k,0]
    ld1rw  z8.s, p0/z, [pB]          ; Broadcast B[k+1,0]
    fmla   z17.s, p1/m, z0.s, z9.s   ; acc1 += A[k] * B[k,1]
    ld1rw  z9.s, p0/z, [pB, 4]       ; Broadcast B[k+1,1]
    fmla   z18.s, p1/m, z0.s, z10.s  ; acc2 += A[k] * B[k,2]
    ld1rw  z10.s, p0/z, [pB, 8]      ; Broadcast B[k+1,2]
    fmla   z19.s, p1/m, z0.s, z11.s  ; acc3 += A[k] * B[k,3]
    ld1rw  z11.s, p0/z, [pB, 12]     ; Broadcast B[k+1,3]
    fmla   z20.s, p1/m, z0.s, z12.s  ; acc4 += A[k] * B[k,4]
    prfm   PLDL1KEEP, [pA, #A_PRE_SIZE]  ; Prefetch A
    ld1rw  z12.s, p0/z, [pB, 16]     ; Broadcast B[k+1,4]
    fmla   z21.s, p1/m, z0.s, z13.s  ; acc5 += A[k] * B[k,5]
    ld1rw  z13.s, p0/z, [pB, 20]     ; Broadcast B[k+1,5]
    fmla   z22.s, p1/m, z0.s, z14.s  ; acc6 += A[k] * B[k,6]
    ld1rw  z14.s, p0/z, [pB, 24]     ; Broadcast B[k+1,6]
    fmla   z23.s, p1/m, z0.s, z15.s  ; acc7 += A[k] * B[k,7]
    ld1rw  z15.s, p0/z, [pB, 28]     ; Broadcast B[k+1,7]
    add    pB, pB, 32                ; Advance B pointer
.endm
```

RISC-V RVV (length-driven)

kernel/riscv64/sgemm_kernel_8x8_zvl128b.c (791 lines)

```
// Simple K-loop, compiler handles scheduling
for (BLASLONG k = 1; k < K; k++) {   // Load 8 scalars from B
    float B0 = B[bi + 0];
    float B1 = B[bi + 1];
    float B2 = B[bi + 2];
    float B3 = B[bi + 3];
    float B4 = B[bi + 4];
    float B5 = B[bi + 5];
    float B6 = B[bi + 6];
    float B7 = B[bi + 7];
    bi += 8;
    // Load vector from A (8 elements @ zvl128b)
    vfloat32m2_t A0 = __riscv_vle32_v_f32m2(&A[ai], gvl);
    ai += 8;
    // 8x vector-scalar FMA: acc += A * B (scalar broadcast implicit)
    result0 = __riscv_vfmacc_vf_f32m2(result0, B0, A0, gvl);
    result1 = __riscv_vfmacc_vf_f32m2(result1, B1, A0, gvl);
    result2 = __riscv_vfmacc_vf_f32m2(result2, B2, A0, gvl);
    result3 = __riscv_vfmacc_vf_f32m2(result3, B3, A0, gvl);
    result4 = __riscv_vfmacc_vf_f32m2(result4, B4, A0, gvl);
    result5 = __riscv_vfmacc_vf_f32m2(result5, B5, A0, gvl);
    result6 = __riscv_vfmacc_vf_f32m2(result6, B6, A0, gvl);
    result7 = __riscv_vfmacc_vf_f32m2(result7, B7, A0, gvl);
}
```

Vector Load

Scalar Broadcast

Vector-Scalar FMA

Predication

p1/m vs. gvl

GEMM Performance Comparison & Attribution

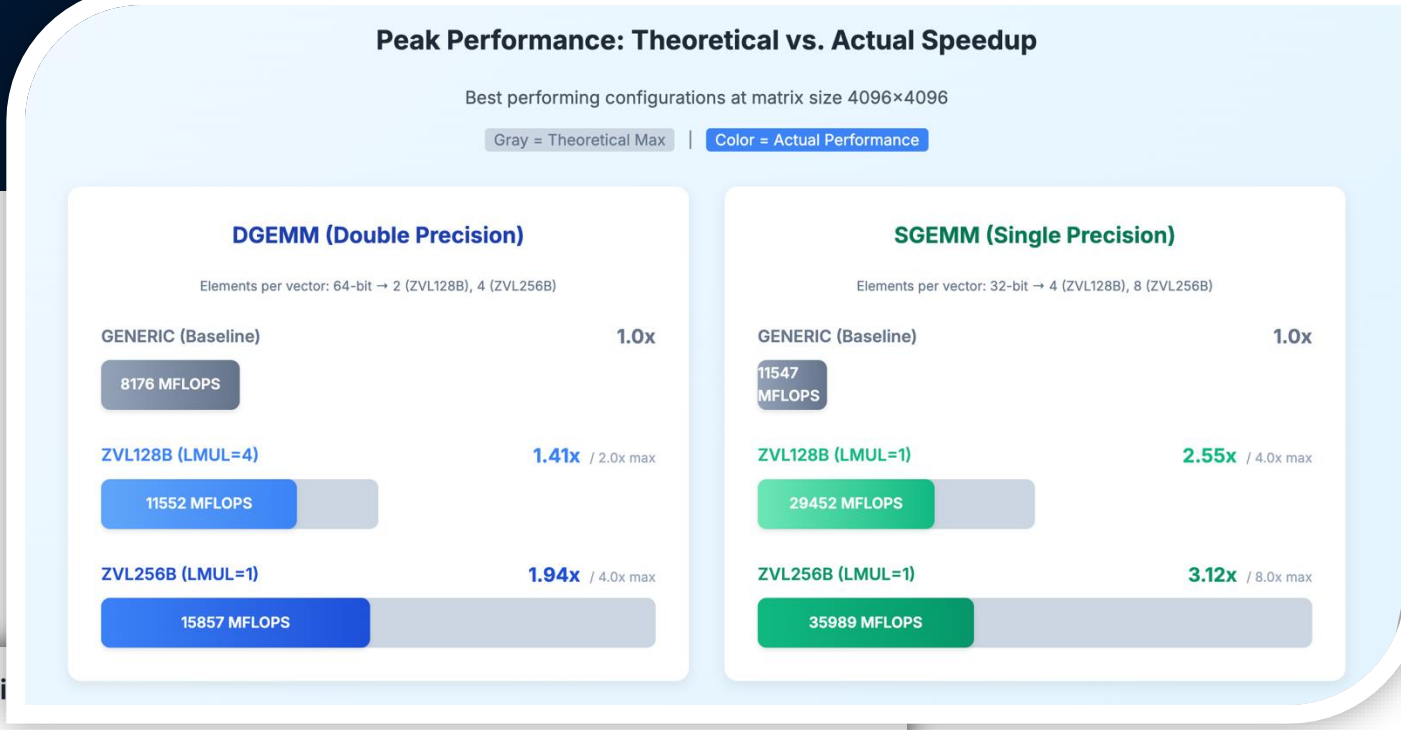
For RISC-V, key observation points are to see what's the impact of the two:

1. *VLEN (Vector Length)*

Wider is Better: Yes or No?

2. *LMUL (Vector Register Grouping, aka. Length Multiplier)*.

How much gains can we get when increasing LMUL from 1 to 2?



Note:

GENERIC means RISC-V Scalar

ZVL128B means RISC-V Vector 128bit VLEN

ZVL256B means RISC-V Vector 256bit VLEN

For Reference, the Benchmark Testing Framework for OpenBLAS: https://github.com/docularxu/OpenBLAS/tree/working.riscv64.optimizations
Raw data and infographic published here: https://docularxu.github.io/OpenBLAS/webpages/rvv_benchmark_openblas_16Oct2025.html

GEMM Performance Comparison & Attribution

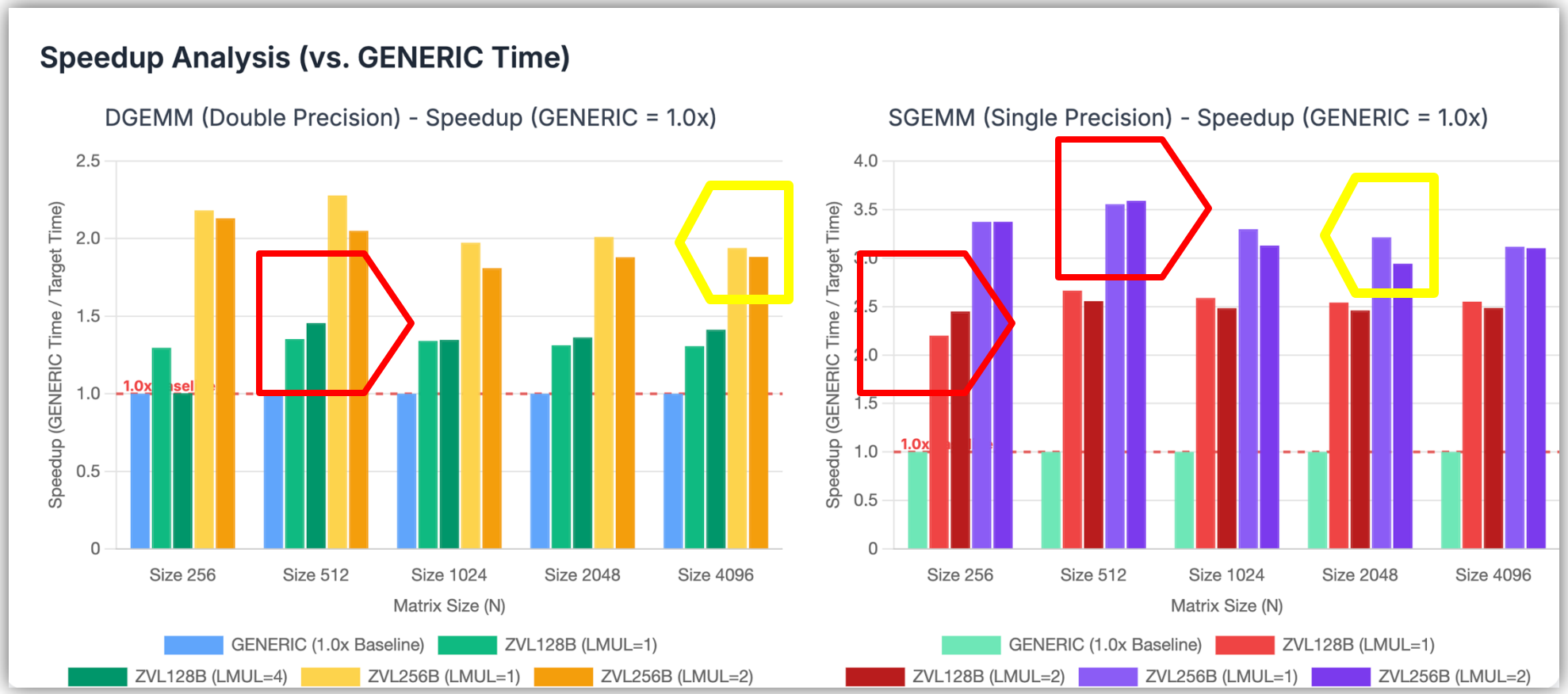
For RISC-V, key observation points are to see what's the impact of the two:

1. *VLEN (Vector Length)*

Wider is Better: Yes or No?

2. *LMUL (Vector Register Grouping, aka. Length Multiplier)*.

How much gains can we get when increasing LMUL from 1 to 2?



Note:

GENERIC means RISC-V Scalar

ZVL128B means RISC-V Vector 128bit VLEN

ZVL256B means RISC-V Vector 256bit VLEN

For Reference, the Benchmark Testing Framework for OpenBLAS: https://github.com/docularxu/OpenBLAS/tree/working.riscv64.optimizations
Raw data and infographic published here: https://docularxu.github.io/OpenBLAS/webpages/rvv_benchmark_openblas_16Oct2025.html

GEMM Performance Comparison & Attribution

RISC-V SUMMIT NORTH AMERICA 2025

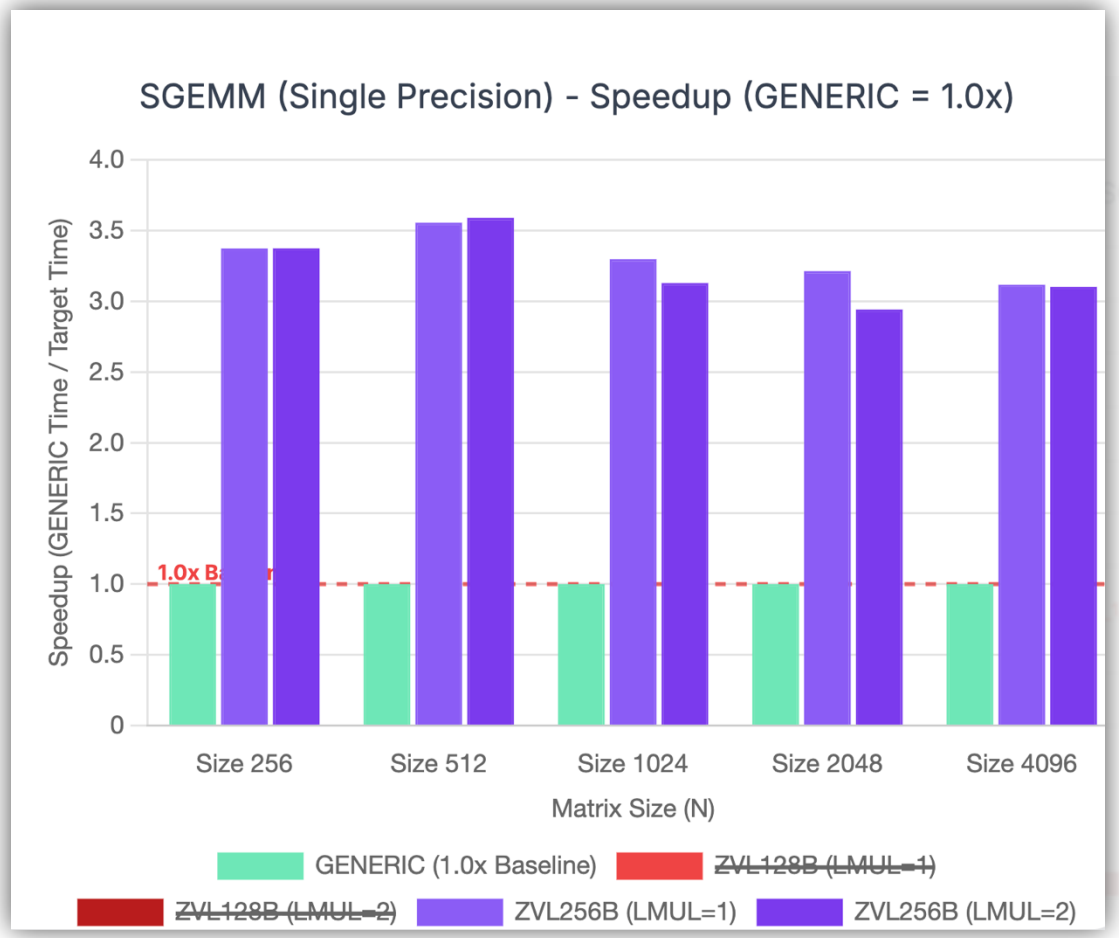
For RISC-V, key observation points are to see what's the impact of the two:

1. *VLEN (Vector Length)*

Wider is Better: Yes or No?

2. *LMUL (Vector Register Grouping, aka. Length Multiplier)*.

How much gains can we get when increasing LMUL from 1 to 2?



OBSERVATAION:

For GEMM, LMUL provides minimal benefit and sometimes hurts performance

1. **Only benefit:** DGEMM on ZVL128B sees modest 8% improvement with LMUL=4, compensating for narrow 128-bit vectors
2. **Matrix-size dependency:** LMUL=2 shows 11% gain for SGEMM ZVL128B at 256×256 (22,805 vs 20,482 MFLOPS) but 3% degradation at 4096×4096 (28,696 vs 29,451 MFLOPS)
3. **Recommendation:** For production deployments on K1, use matrix-size adaptive LMUL tuning: LMUL=2 for small matrices (<1024), LMUL=1 for large matrices (≥1024), except DGEMM ZVL128B where LMUL=4 provides 8% gain

Note:

GENERIC means RISC-V Scalar

ZVL128B means RISC-V Vector 128bit VLEN

ZVL256B means RISC-V Vector 256bit VLEN

For Reference, the Benchmark Testing Framework for OpenBLAS: https://github.com/docularxu/OpenBLAS/tree/working.riscv64.optimizations
Raw data and infographic published here: https://docularxu.github.io/OpenBLAS/webpages/rvv_benchmark_openblas_16Oct2025.html

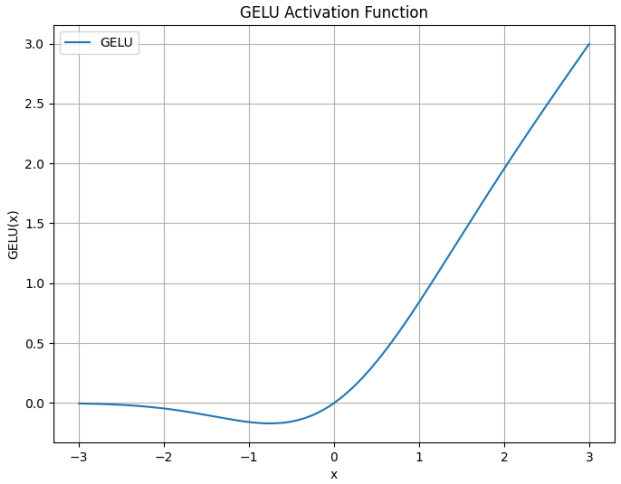
PART IV

Non-Linear Operator Deep Dive (SLEEF)

Introduction to Non-Linear Operators

RISC-V SUMMIT NORTH AMERICA 2025

Non-linear operators are essential for LLM performance, but they rely heavily on optimized standard mathematical functions (*libm*).



Softmax

Converts raw scores (logits) to probabilities.

Core Dependencies:

- `exp` (Exponential function)
- `reduction / sum`

GeLU (Gaussian Error Linear Unit)

A standard activation function in Transformer models.

Core Dependencies:

- `exp` (Often via error function approximation)
- `tanh` (Hyperbolic Tangent)

LayerNorm (Layer Normalization)

Stabilizes the learning process across feature dimensions.

Core Dependencies:

- `sqrt` (Square Root)
- `reduction / sum`

- **exp:** (Used in Softmax, GeLU)
- **tanh:** (Used in GeLU)
- **sqrt:** (Used in LayerNorm)
- **reduction / sum:** (Used in LayerNorm, Softmax)

e^x

Exponential (exp)

Widely used in **Softmax** and **GeLU** activation functions. Its vectorization is a key performance bottleneck for these critical AI operators.

tanh

Hyperbolic Tangent (tanh)

Another core component of the **GeLU** activation function, crucial for the model's non-linear expression capabilities.

$\sqrt{x}$

Square Root (sqrt)

Primarily used for variance calculation in **LayerNorm**, a key step in maintaining training stability.

Vectorizing AI Math Operators
Why SLEEF?

RISC-V SUMMIT
NORTH AMERICA 2025

The performance of non-linear operators like *GeLU*, *Softmax*, and *LayerNorm* depends on the efficient vectorization of fundamental math functions.

- **Core Challenge:** How to apply SIMD principles to complex functions like *exp*, *tanh*, and *sqrt*.
- **The Solution:** High-performance math libraries like *SLEEF* provide hand-tuned vector implementations of these functions for different architectures.
- **Our Focus:** Comparing how SLEEF leverages the unique features of RISC-V Vector and other VLA platforms to accelerate these critical AI operators.

SLEEF Performance Measured

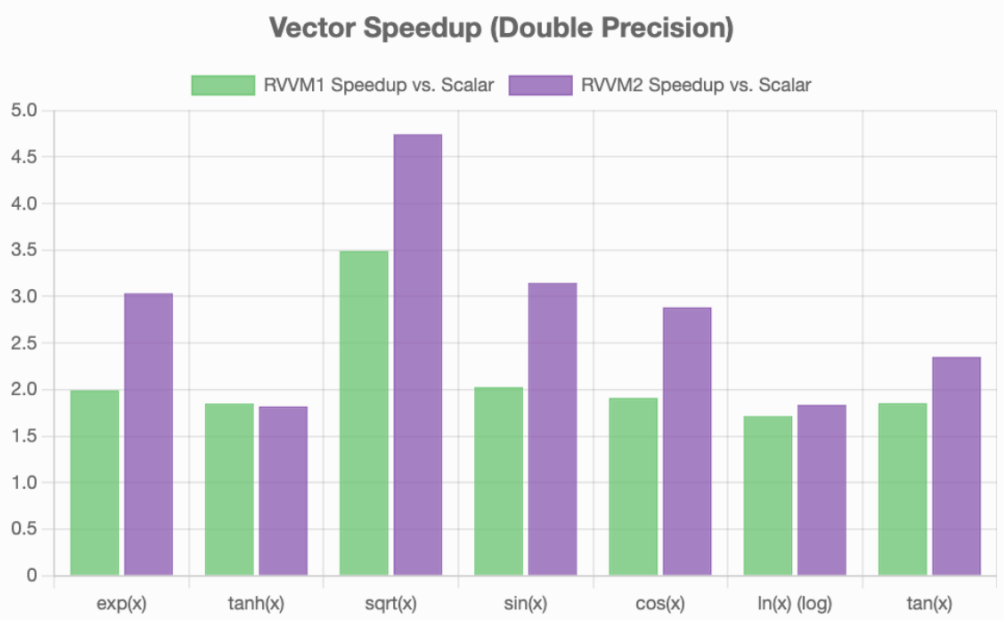
(Vector vs. Scalar) Speedup Comparison

Three categories are measured:

- Scalar
- RVV with LMUL=1
- RVV with LMUL=2

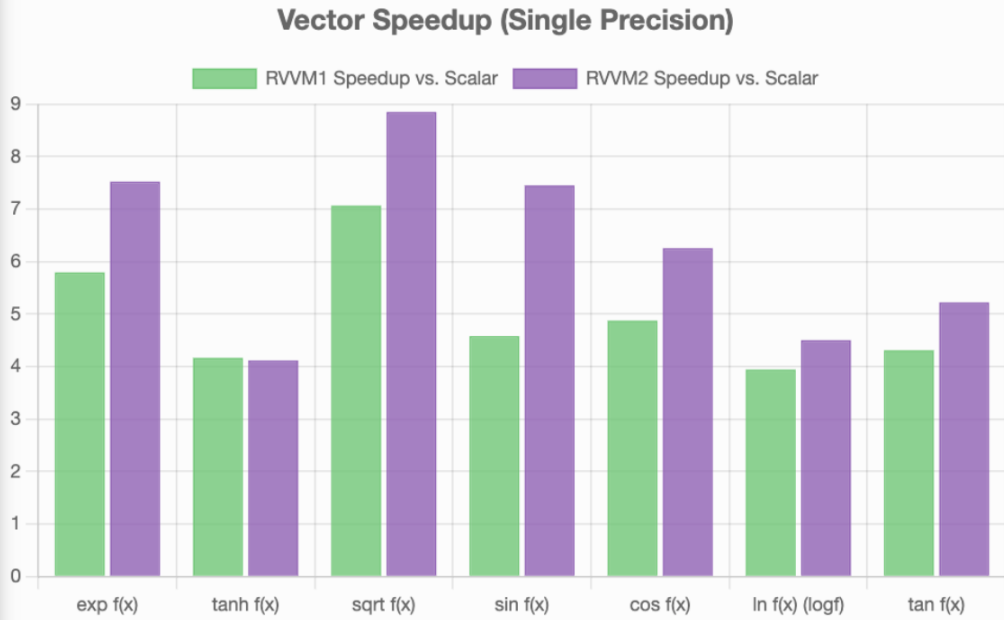
LMUL = Length Multiplier

Double Precision Speedup (vs. Sleef Scalar)



Speedup factors of RVVM1 and RVVM2 compared to the Sleef Scalar implementation for double precision functions. Higher bars indicate greater speedup.

Single Precision Speedup (vs. Sleef Scalar)



Speedup factors of RVVM1 and RVVM2 compared to the Sleef Scalar implementation for single precision functions. Higher bars indicate greater speedup.

For Reference, the Benchmark Testing Framework for SLEEF: https://github.com/docularxu/sleef/commits/working.sleef.bench/
Raw data and infographic published here: https://docularxu.github.io/sleef/webpages/rvv_benchmark_16Oct2025.html

SLEEF Performance Measured
(LMUL = 2 vs. LMUL = 1) RVVM2 Gain Over RVVM1

RISC-V SUMMIT NORTH AMERICA 2025

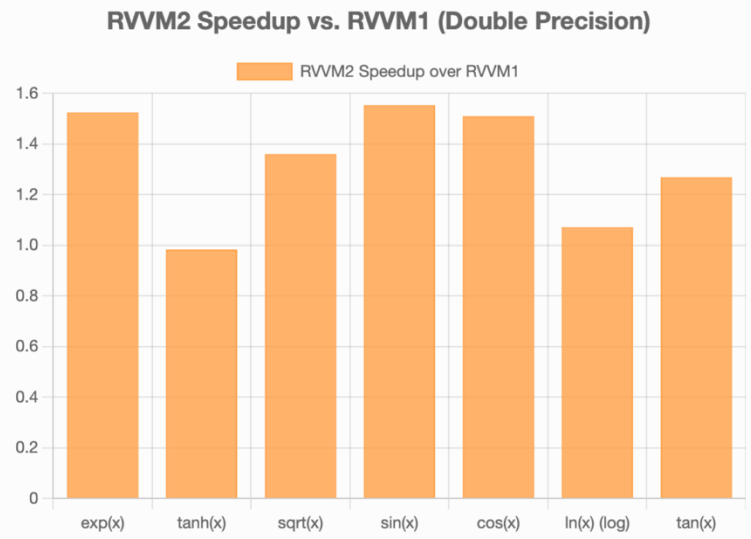
RISC-V provides a powerful feature not found in SVE: **Vector Register Grouping (LMUL, ie. Length Multiplier)**. This allows software to logically combine multiple registers to operate on wider data or reduce loop overhead.

Three categories are measured:

- Scalar
- RVV with LMUL=1
- RVV with LMUL=2

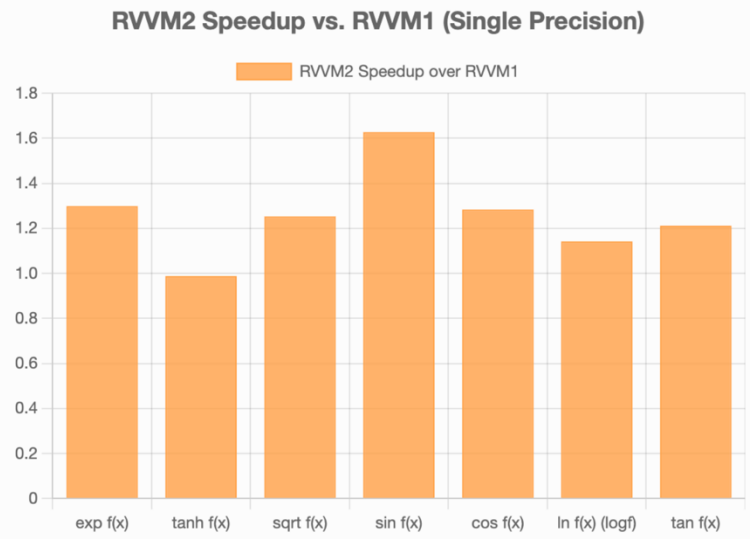
SLEEF Config	Description	Relative Performance
RVVM1 (LMUL=1)	Standard, one register per operation.	1.0x (Baseline)
RVVM2 (LMUL=2)	Groups two registers	**~1.4x** *Caution: tanh(x) sees no gains when growing LMUL*

RVVM2 Speedup vs. RVVM1 (Double Precision)



Speedup factor of RVVM2 using RVVM1 as the baseline. This highlights the performance benefit of increasing LMUL from 1 to 2.

RVVM2 Speedup vs. RVVM1 (Single Precision)



Speedup factor of RVVM2 using RVVM1 as the baseline. This highlights the performance benefit of increasing LMUL from 1 to 2.

For Reference, the Benchmark Testing Framework for SLEEF: https://github.com/docularxu/sleef/commits/working.sleef.bench/
Raw data and infographic published here: https://docularxu.github.io/sleef/webpages/rvv_benchmark_16Oct2025.html

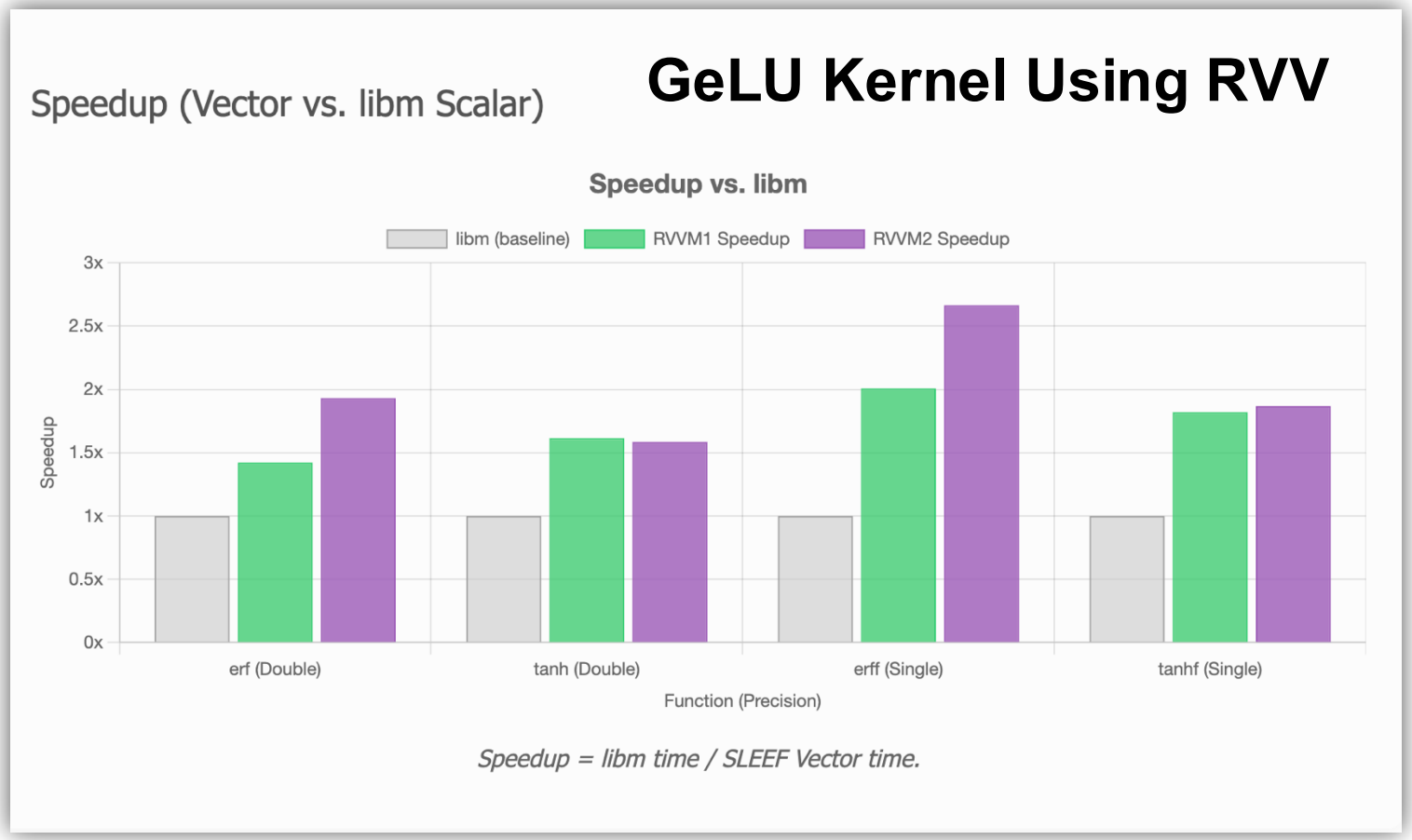
SLEEF Performance Measured

(LMUL = 2 vs. LMUL = 1) GeLU on RISC-V

Comparison of libm scalar vs. SLEEF vector implementations in GeLU.

- Scalar
- RVV with LMUL=1
- RVV with LMUL=2

SLEEF Configuration	Relative Performance
Libm (scalar)	1.0x (Baseline)
RVVM1 (LMUL=1)	1.5x sppedup
RVVM2 (LMUL=2)	**~2.0x speedup**



For Reference, the Benchmark Testing Framework for SLEEF: https://github.com/docularxu/sleef/commits/working.sleef.bench/
Raw data and infographic published here: https://docularxu.github.io/sleef/webpages/rvv_benchmark_gelu_17Oct2025.html

PART V

Tools and Toolchains

Tooling & Observability

Performance Monitoring (perf):

- **RISC-V Platform (K1):** Support is **available in their vendor tree. Refer to:**

  https://bianbu.spacemit.com/en/development/perf/

- The SpacemiT K1 provides some essential events, but the ecosystem is still evolving. Not upstreamed.
- Improving perf support is a key area for community contribution.

Examples:

(Right) K1 proprietary/vendor-specific events

vs.

(Left) the Linux common events

Spacemit K1/M1 Supported Perf Events

Event	Status	Event	Status	Event	Status	Event	Status
cycles	✓	LLC-prefetches	✗	dtlb_load_miss	✓	ecall_inst	✓
branches	✓	LLC-prefetch-misses	✗	dtlb_store_miss	✓	failed_sc_inst	✓
branch-misses	✓	dTLB-loads	✗	itlb_load_miss	✓	fence_inst	✓
instructions	✓	dTLB-load-misses	✓	jtlb_miss	✓	fp_div_inst	✓
bus-cycles	✗	dTLB-stores	✗	l1d_access	✓	fp_inst	✓
ref-cycles	✗	dTLB-store-misses	✓	l1d_amr_active	✓	fp_load_inst	✓
page-faults	✓	dTLB-prefetches	✗	l1d_excl_evict	✓	fp_store_inst	✓
context-switches	✓	dTLB-prefetch-misses	✗	l1d_load_access	✓	load_inst	✓
cpu-migrations	✓	iTLB-loads	✗	l1d_load_miss	✓	lr_inst	✓
stalled-cycles-frontend	✓	iTLB-load-misses	✓	l1d_miss	✓	mult_inst	✓
stalled-cycles-backend	✓	branch-loads	✗	l1d_prefetch_hit	✓	sc_inst	✓
task-clock	✓	branch-load-misses	✗	l1d_prefetch_refill	✓	store_inst	✓
cache-misses	✗	node-loads	N/A	l1d_store_access	✓	unaligned_load_inst	✓
cache-references	✗	node-load-misses	N/A	l1d_store_miss	✓	unaligned_store_inst	✓

Summary & Call to Action

RISC-V SUMMIT NORTH AMERICA 2025

Key Takeaways from Our Porting Journey:

1. **Master VL Control:** For compute-bound tasks, **data locality is king**. Don't chase maximum LMUL. Use RVV's vsetvl to manually craft small VL loops that fit the L1 cache, just as OpenBLAS does.
2. **Leverage the Math Libraries:** Don't reinvent the wheel for non-linear functions. **Use and contribute to libraries like SLEEF.** Improving these libraries is the fastest path to unlocking RVV's performance for AI workloads.
3. **Embrace Observability:** Trust, but verify. Use perf and analyze the generated assembly to ensure your VL settings, tail policies, and memory accesses are what you expect.
4. **Engage with the Ecosystem:** RISC-V's greatest strength is its community. Submit patches and provide feedback to upstream toolchains like LLVM and GCC. Every improvement benefits everyone.

RISC-V SUMMIT

NORTH AMERICA 2025

Connect with me:

GUODONG XU

EMAIL: guodong@riscstar.com
LINKEDIN: https://www.linkedin.com/in/docularxu/
GITHUB: https://www.github.org/docularxu

THANK YOU!

RISCSTAR™ SOLUTIONS

SLEEF Performance Measured

(Vector vs. Libm reference) Speedup Comparison

RISC-V SUMMIT NORTH AMERICA 2025

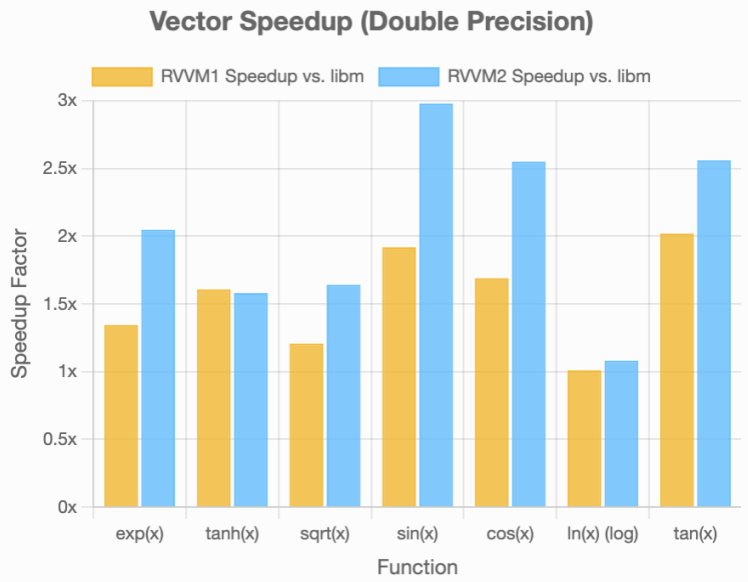
Three categories are measured:

- Libm Reference
- RVV with LMUL=1
- RVV with LMUL=2

LMUL aka. Length Multiplier

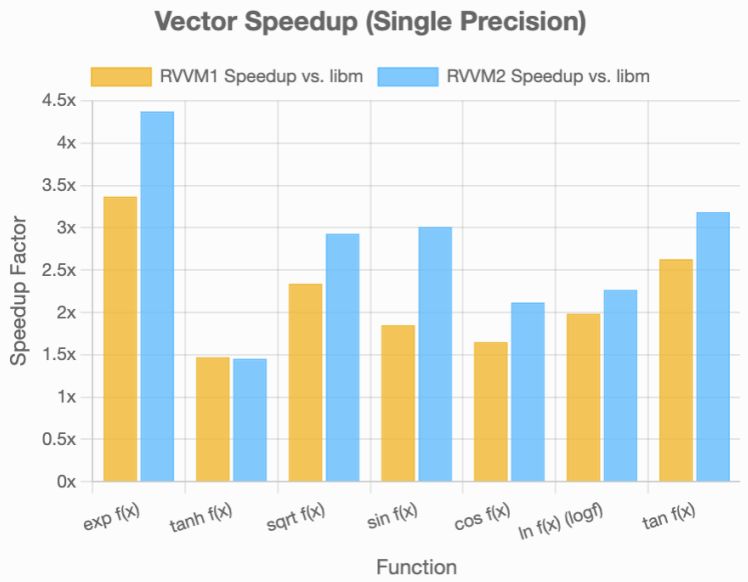
Speedup Comparison (Vector vs. libm Reference)

Double Precision Speedup (vs. libm Reference)



Speedup factors of RVVM1 and RVVM2 compared to the libm reference implementation. This shows the vector gain over standard system libraries.

Single Precision Speedup (vs. libm Reference)



Speedup factors of RVVM1 and RVVM2 compared to the libm reference implementation. This shows the vector gain over standard system libraries.

For Reference, the Benchmark Testing Framework for SLEEF: https://github.com/docularxu/sleef/commits/working.sleef.bench/
Raw data and infographic published here: https://docularxu.github.io/sleef/webpages/rvv_benchmark_16Oct2025.html