Vectorized Nonlinear Functions with the RISC-V Vector Extension

Eric Bavier, Nicholas Knight, Hugues de Lassus Saint-Geniès, and Eric Love
Algorithms & Libraries Team, SiFive, Inc.
{eric.bavier, nick.knight, hugues.delassus, eric.love}@sifive.com

***Abstract*—The RISC-V Vector instruction set extension (RVV) provides scalable data-parallel instructions suitable for accurate and performant implementations of numerical algorithms across many application domains [1]. The primary objective of this paper is to share our experience implementing vector C99 `<math.h>` (libm) functions using RVV. Our contributions are threefold: First, we contributed an RVV port of SLEEF, a multi-platform open-source vector libm. Second, we show that while SLEEF simplifies porting efforts, it also precludes some RVV-specific optimization opportunities. With SiFive's X280 vector processor micro-architecture as a case-study, we highlight RVV features that optimized code can use. We also expand the discussion to how these features might be used differently when optimizing for other cores. Third, we compare the performance of our SLEEF RVV port to our own RVV-native routines. We present results from 1-ulp accurate implementations of Libm functions in a cycle-accurate simulation of the X280 pipeline to show the impact of RVV-enabled optimizations.**

***Index Terms*—floating point, vector mathematical functions, RISC-V vectors, scalable vectors.**

I. Introduction

RISC-V is an open standard instruction set architecture (ISA) designed to be modular and extensible, featuring a small, mandatory base ISA and many optional extensions [2]. As most modern ISAs support data parallelism in the form of single instructions that apply to multiple data (SIMD), also called vector processing, RISC-V has ratified the Vector extension (RVV) in November 2021. RVV contrasts with the fixed-width SIMD ISAs that have proliferated since the late 1990s, in many ways more closely resembling the vector ISAs of the two decades prior [3]. The C standard library provides a collection of mathematical functions ('Libm') defined on scalars. They cannot leverage vector processing without sophisticated compiler intervention. Even if a compiler could automatically vectorize a highly optimized scalar Libm routine, the result may not be competitive with an algorithm designed to target a vector machine.

As such, a number of vector math libraries have appeared over the years, open-source and proprietary [4], [5]. SLEEF's algorithms are expressed in terms of a set of vector primitives ('intrinsics'), selected for being shared by many vector ISAs. The objective is to simplify the effort of porting SLEEF to a new ISA: all that is required is to provide a header file that maps SLEEF intrinsics to the corresponding (sequences of) instructions in the target ISA. In particular, no serious experience with numerical algorithms is required. And in fact we were able to port SLEEF to the RISC-V vector ISA (RVV) in a few weeks[1].

We hypothesized that the SLEEF implementations, designed around ISA-agnostic intrinsics, might not take full advantage of any particular ISA, like RVV. We set out to analyze its implementations, going back to the underlying mathematics, then reimplementing them using RVV from the outset, bypassing the SLEEF intrinsics abstraction. Indeed, for all routines our 'native' RVV implementations obtained considerable speedups, without loss of accuracy, over their SLEEF relatives. This effort illuminated a few facets of RVV, which SLEEF was not able to leverage, but that we think are particularly useful to high-performance vector math library design.

II. Vectorizing Nonlinear Functions with RVV

In Section II-A, we highlight four features of RVV relevant for vector math algorithm design: vector register grouping, vector-scalar operations, mixed-width arithmetic, and dynamic vector length. Then, in Section II-B, we examine how these features impact two problems at the heart of Libm algorithms: polynomial evaluation and range reduction.

A. RVV ISA Features Relevant to Libm Construction

1) Vector Register Grouping: RVV adds 32 architectural vector registers `v0`–`v31` to the base ISA. Each vector register has VLEN bits, where VLEN is an implementation-defined power of two from 128 to 65536. A configurable *length multiplier* (LMUL) allows multiple vector registers to be grouped together and operated on by a single vector instruction. It trades a gain of *data-level parallelism* (DLP), in the form of longer application vectors, for a potential loss of *instruction-level parallelism* (ILP), due to exacerbated register pressure. This DLP is exploited *temporally* rather than *spatially*, i.e. additional vector elements are processed over multiple cycles of occupancy of the machine's datapath — whose physical width does not change — rather than simultaneously across more parallel hardware. Increasing the temporal vector length can improve utilization of long-latency functional units by reducing stalls due to read-after-write (RAW) hazards between dependent instructions, like chained multiply-adds arising in polynomial evaluation, without increasing code size or control flow complexity.

[1] See https://github.com/shibatch/sleef/pull/449

Conceptually, similar benefits can be achieved via program transformations like loop unrolling or software pipelining. However, these techniques tend to increase code size — and thus pressure on the instruction memory system — in proportion to the unrolling factor or pipeline depth. Considering how typical math library routines range from dozens to hundreds of instructions, with long chains of dependent arithmetic, these transformations should be applied conservatively. While speculative, out-of-order execution and register renaming can, in principle, implement these transformations dynamically, pressure from large code size and long dependence chains effectively downstreams onto the reorder buffer and branch predictor.

2) Vector-Scalar Operations: Many arithmetic instructions in RVV have "vector-scalar" forms, which source one of their operands from the (scalar) integer or floating-point register files. This avoids a common pattern in other vector ISAs, where scalars must be explicitly broadcast ('splatted') to a vector before being used in vector arithmetic. RVV vector-scalar instructions avoid consuming a vector register in this manner, thus reducing register pressure.

3) Mixed-Width Arithmetic: In order to produce accurate results for mathematical functions, it is often necessary to perform intermediate calculations with higher precision than the input or output data. When the data is already at the widest supported precision, error-free transforms (EFTs) are especially useful [6], [7]. Otherwise, intermediates can leverage a wider precision, e.g., using IEEE $\mathbb{F}_{64}$ intermediates when targeting $\mathbb{F}_{32}$. Such widening and narrowing conversions are straightforward in most scalar ISAs, but more challenging in many vector ISAs. For example, it may be required to split the vector in two, perform widening conversions and subsequent arithmetic on each half individually, finally followed by narrowing conversions and merging.

RVV supports mixed-width arithmetic instructions that can avoid such overheads. For example the following operations produce a $2 \cdot \text{SEW}$ result: `vfwadd.vv` and `vfwmul.vv` from two inputs of width SEW, the `vfwadd.wv` from a $2 \cdot \text{SEW}$ input and an SEW input, and `vfwmacc.vv` from two multiplicands of width SEW and an addend of width $2 \cdot \text{SEW}$. Widened results are stored in a register group with twice the registers, up to a maximum of 8. This limits their use to functions using $\text{LMUL} \leq 4$.

4) Dynamic Vector Length: RVV supports a programming style that is VLEN-agnostic. The same software can be executed without recompiling across hardware with different VLEN. For fixed-width SIMD ISAs, this portability is usually achieved with multiple implementations of the same function for different SIMD widths, and run-time dispatch, in 'fat' libraries. With RVV, there is no need to dispatch based on VLEN nor bloat the library with VLEN-specific implementations. A scalable vector ISA like RVV naturally simplifies library design.

RVV has a control register, `vl`, that governs how many elements instructions process. `vl`-setting instructions take in the application vector length (AVL) (e.g. a billion $\mathbb{F}_{32}$ numbers) and set `vl` based on what the hardware actually supports (perhaps just four). This removes any special handling for loop remainders: the same loop body can execute using a `vl` as small as 1.

In the context of a vector math library, we can support a traditional 'array API', where the user passes pointers to buffers and their length and the library does the stripmining, or a 'vector API', where vectors are passed and returned along with `vl`, and the user performs the stripmining. In practice, the latter facilitates compiler optimizations.

B. Case Studies in Vectorizing Libm Functions

We now consider how the RVV features described above apply to two tasks at the core of most Libm functions: evaluating polynomial approximations, and accurately reducing the range of the input argument to a finite interval.

1) Case Study: Polynomial Evaluation: Almost all numerical algorithms for Libm functions require evaluating a polynomial approximation of potentially large degree, making it a critical component of runtime [8]. A common method is Horner's scheme, which minimizes the number of arithmetic operations. Specifically, it evaluates $P(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$ inductively, computing P_0 with $P_n = c_n$ and $P_i = c_i + x P_{i+1}$, usually with an FMA instruction. As such, Horner's scheme consists entirely of back-to-back dependent FMAs.

Other methods, such as Estrin's scheme or 'higher-order' variants of Horner's scheme, mitigate these data dependences by exposing more ILP at the cost of increasing the overall arithmetic [9], [10]. These techniques are crucial to effectively utilize a machine with multiple arithmetic units (ALU). However, our primary optimization target, SiFive's X280, is a microarchitecture with a single vector ALU[2]: while the increased ILP of the alternatives enables hiding most of the pipeline stalls, we see all of the arithmetic on the critical path, and the loss of Horner's optimality is evident.

On the other hand, we found that Horner's scheme with $\text{LMUL} = 4$ completely hides the latency of dependent FMAs, and seems to be the best-performing option on the X280. In general, for a chain of dependent arithmetic instructions that process w bits per cycle over a depth-d (cycles) pipeline, RAW hazard stalls are avoided when the vector length (in bits) is sufficiently large, $\text{LMUL} \cdot \text{VLEN} \geq wd$. Of course, VLEN, w, and d are machine parameters, and w and d generally differ with instructions, so the best choice of LMUL will vary. This motivates a programming style that keeps LMUL a tunable parameter.

RVV vector-scalar FMAs only support scalar multiplicands, but Horner's scheme accumulates with coefficients in the addend, so explicit splats are necessary for all but maybe c_n. RVV provides two mechanisms for this: splatting from a scalar register and splatting from memory. The better approach depends on the microarchitecture and the context.

Consider splatting from a scalar register as in Listing I. As mentioned above, X280 has a single vector ALU; furthermore, it uses this ALU to process the splat (`vfmv.v.f`), with

[2]See https://www.sifive.com/cores/intelligence-x280.

comparable occupancy to the subsequent FMA (`vfmadd.vv`). Consequently, the splat and FMA execute in series, increasing the arithmetic cost on the critical path.

LISTING I
SINGLE STEP OF HORNER'S EVALUATION SCHEME

```
# Assume register f0 contains coeff. c0
# and v4-v7 contains x, and v8-v11 P(x)
vfmv.v.f v12, f0      # v12[i] = c0
vfmadd.vv v8, v4, v12 # P(x)*x + c0
```

Since this code is executed within a stripmined loop, rather than rematerializing the constant vector each iteration, we consider hoisting the splat above the loop to amortize this extra arithmetic. High vector register pressure may require us to spill the constant vector, reloading it in each iteration prior to its associated FMA. On the X280 this is often a profitable choice: the vector load unit is independent of the vector ALU, so while the latter is occupied with an FMA, the former can proceed concurrently, loading the next (splatted) coefficient.

RVV's strided vector memory operations enable a further simplification, which we call splatting from memory. Rather than spilling the splatted coefficient, we can simply store the scalar to memory, and load it with a stride-zero vector load:

LISTING II
BROADCAST WITH ZERO-STRIDED LOAD

```
# Assume t0 holds pointer to coeff. c0
vlse32 v12, (t0), zero
vfmadd.vv v8, v4, v12 # P(x)*x + c0
```

As opposed to general strides, X280 optimizes the zero-stride case to execute at high throughput, enabling us to perform the splat using the vector load unit, rather than the vector ALU.

We have glossed over some contextual assumptions here, that these loads hit in a nearby cache so that the memory latency is readily hidden by X280's decoupled microarchitecture, and that there is little other contention for the vector memory system. As with LMUL, we think the correct splatting technique should be viewed as a tuning parameter. For example, if we added another vector ALU to X280, we might prefer an alternative to Horner's scheme to expose vector ILP, but the single vector load unit would not provide sufficient coefficient bandwidth, so we might end up with a combination of splatting techniques.

2) Case Study: Range Reduction with Mixed-Width Arithmetic: Polynomial approximations tend to provide acceptable accuracy only for inputs inside a narrow interval, so most Libm functions also include a *range reduction* step to map their arguments to a subset of the reals. For example, a `sin` function for $\mathbb{F}_{32}$ data might reduce its input to the $[-\pi/2; \pi/2]$ domain. Conceptually, this is accomplished by computing $x - z\pi$ for some integer z. If π is approximated by a single floating point number this may introduce intolerable roundoff error, so it is necessary to use an extended-precision approach with more bits of π, e.g. as an unevaluated sum $C_1 + C_2 + C_3 \approx \pi$. This could be accomplished with heavyweight EFTs as in SLEEF, or by using $\mathbb{F}_{64}$ intermediate values. RVV makes the latter option attractive because its support for widening mixed-precision and vector-scalar operations eliminates some of the complexity that might otherwise be required. For example, Listing III from our native RVV Libm computes the above expression efficiently.

LISTING III
ADDITIVE REDUCTION USING MIXED-PRECISION

```
# Assume registers f0+f1+f2 ~= -pi,
# v4-v7 contains x, and v8-v11 the
# reduction multiplier z
vsetvli   s0, a1, e32, m4
vfmacc.vf v4, f0, v8   # x+z*f0
vfwmul.vf v16, v8, f1  # z*f1 (f64)
vfmul.vf  v8, v8, f2   # z*f2
vfwadd.wv v16, v16, v4 # x+z*f0+z*f1
vfwadd.wv v16, v16, v8 # x+...+z*f2
vsetvli   zero, zero, e64, m8
# ...cont. w/ wide reduced input v16-v23
```

The main body of code here is just five instructions; other vector ISAs might require ten or more instructions to do something similar in a native wide precision. At that point, it may be preferable to use EFTs instead [11].

III. RESULTS

Performance and accuracy are the main tradeoffs when implementing non-linear functions. In Section III-A, we present benchmark results for four functions vectorized in both RVV-agnostic and RVV-conscious ways. In Section III-B, we discuss how we measured accuracy of our RVV-native library.

A. Performance Measurements

To illustrate the importance of the foregoing discussion, we measured throughput of the $\mathbb{F}_{32}$ exponential, natural logarithm, sine, and error functions using a cycle-accurate simulation of X280, and present speedups against SLEEF's "pure C" scalar implementations in Table I. For reference, we include Newlib's scalar Libm, compiled with `OBSOLETE_MATH_DEFAULT=0` for faithful roundings, as a more optimized scalar implementation. Next, we used our RVV port of SLEEF to establish an RVV-agnostic vector performance baseline ("SLEEF"). Due to SLEEF's architecture, we were unable to support LMUL greater than 2 in our RVV port[1], which is one aspect of its incomplete exploitation of RVV features. Finally, we benchmarked our own RVV-native Libm functions to show the benefits of RVV-conscious optimization ("SiFive").

Our benchmarks apply the functions to an array of 1024 pseudo-random inputs, uniformly distributed in $[0, 1]$. For the listed functions, a w subscript indicates that the implementation performs some computations in $\mathbb{F}_{64}$, otherwise not. Missing data in the form of a '–' indicates that a corresponding implementation could not be benchmarked. E.g. our internal EFT functions do not support LMUL $= 8$, so we show a result only for $\exp$, which is implemented without EFTs; but we suspect that for other functions excessive register spilling would make performance worse, not better.

As suggested in Section II-B1, we expected LMUL $= 4$ to be a sweet spot: indeed, the difference between LMUL $= 4$ and LMUL $= 8$ in the one feasible case was negligible compared to the speedups obtained from increasing to LMUL $= 4$. In

TABLE I
SPEEDUP OF LMUL & WIDENING VS. SLEEF SCALAR

		SLEEF		SiFive			
Function	Newlib	m1	m2	m1	m2	m4	m8
exp	–	8.1	12.2	10.8	19.4	30.0	31.7
$\exp_w$	1.8	–	–	12.5	21.0	28.9	–
log	–	4.4	4.8	7.4	11.4	13.8	–
$\log_w$	2.3	–	–	9.2	15.0	17.8	–
sin	–	6.6	9.8	9.0	15.9	20.5	–
$\sin_w$	3.4	–	–	14.1	24.7	32.7	–
erf	2.2	4.6	5.2	4.3	7.8	10.9	–
erf_w	–	–	–	16.5	28.2	28.8	–

TABLE II
FUNCTION ACCURACY IN ULPS

Function	Newlib	SLEEF	SiFive
exp	–	0.9312	0.8912
$\exp_w$	0.5016	–	0.8912
log	–	0.5906	0.5972
$\log_w$	0.8177	–	0.6182
sin	–	0.9036	0.9973
$\sin_w$	0.5607	–	0.9840
erf	0.9679	0.9998	0.9998
erf_w	–	–	0.9467

functions where higher intermediate precision was needed, widening to $\mathbb{F}_{64}$ typically provided additional performance.

B. Accuracy Measurements

Implementations are tested on the open-source functional simulator QEMU against higher-precision reference functions, with their maximum absolute error quantified in units in the last place (ulps) following Muller's Definition 6 [12]. Since univariate $\mathbb{F}_{16}$ and $\mathbb{F}_{32}$ functions and bivariate $\mathbb{F}_{16}$ functions have at most 2^{32} inputs to check, exhaustive testing is feasible [13]. The input space of bivariate $\mathbb{F}_{32}$ functions and of $\mathbb{F}_{64}$ functions is too large for exhaustive testing, so we use targeted testing and pseudo-random testing. For targeted testing, inputs are sampled from domains of interest, e.g. when close to overflow or underflow, normal powers of two, infinities, zeros, NaNs, and mantissa patterns. For pseudo-random testing, we do not want to sample inputs from a uniform distribution over $\mathbb{R}$ because some binades may never be covered. Instead, integers are uniformly generated using scrambled linear pseudo-random number generators [14], then type-punned to same-width floating-point numbers and passed as inputs to library functions. This makes it highly likely all binades are covered given enough testing.

While SLEEF implementations are comparatively "mature" and presumed correct to their advertised accuracy, our testing identified a number of accuracy issues[3]. Using this framework we have grown confident that our RVV Libm functions have below 1 ulp maximum error, c.f. Table II, and that our comparison to their equivalents in SLEEF and Newlib in Section III-A is fair.

IV. CONCLUSION

We have shown how several features of RVV can be leveraged in the design of efficient vector math functions. First, the reconfigurable vector register file enables software to increase the temporal vector length, which can better hide the pipeline latency of functional units in polynomial evaluations. Second, vector-scalar instructions can reduce the pressure on vector register groups. Third, mixed-width arithmetic instructions can simplify algorithms that manipulate intermediate quantities in higher precision. Finally, dynamic vector length simplifies writing portable, high performance software.

Our SLEEF port to RVV demonstrated significant speedups on SiFive X280 over Newlib. Our implementations, designed to leverage the aforementioned RVV features, outperform SLEEF with comparable (1-ulp) accuracy. Such RVV-specific techniques run counter to SLEEF's ISA-agnostic design, thus it may be more appropriate to pursue them as a separate project.

Future work concerns extending the LLVM compiler to leverage our library via "libcall widening". For example, in a program that applies `exp` componentwise to an array, LLVM's vectorizer can fuse scalar calls into fewer invocations of a vector version.

[3]See SLEEF Issues 451, 452, 457, 460, 461, 465, and 466.

REFERENCES

[1] "RISC-V "V" Vector Extension Version 1.0," Nov. 2021.

[2] Andrew Waterman and Krste Asanović, Ed., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, Dec. 2019.

[3] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[4] N. Shibata and F. Petrogalli, "SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1316–1327, 2020.

[5] C. Lauter, "A new open-source SIMD vector libm fully implemented with high-level scalar C," in *2016 50th Asilomar Conference on Signals, Systems and Computers*, 2016, pp. 407–411.

[6] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[7] M. Lange and S. M. Rump, "Faithfully rounded floating-point computations," *ACM Trans. Math. Softw.*, vol. 46, no. 3, jul 2020.

[8] J.-M. Muller, *Elementary Functions*, 3rd ed. Birkhäuser Basel, 2016.

[9] G. Estrin, "Organization of Computer Systems: The Fixed Plus Variable Structure Computer," in *Papers Presented at the Western Joint IRE-AIEE-ACM Computer Conference*. New York, NY, USA: ACM, 1960, pp. 33–40.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.

[11] S. Boldo, M. Daumas, and R.-C. Li, "Formally Verified Argument Reduction with a Fused Multiply-Add," *IEEE Transactions on Computers*, vol. 58, no. 8, pp. 1139–1145, 2009.

[12] J.-M. Muller, "On the definition of ulp(x)," INRIA, LIP, Research Report RR-5504, LIP RR-2005-09, Feb. 2005.

[13] A. Sibidanov, P. Zimmermann, and S. Glondu, "The CORE-MATH Project," in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 26–34.

[14] D. Blackman and S. Vigna, "Scrambled Linear Pseudorandom Number Generators," *ACM Trans. Math. Softw.*, vol. 47, no. 4, sep 2021.