Vectorizing Nonlinear Functions with the RISC-V Vector ISA

September 5, 2023 • Portland, OR
Eric Bavier, Nick Knight, Hugues de Lassus and Eric Love
Introduction and Contributions

AI/ML/HPC are data- and compute-intensive

Data parallelism in numerical methods

Nonlinear mathematical functions (exp, log, erf...) often invoked many times on independent data.
- Scientific simulation
- Data analytics
- Machine learning

Vector Math Library: numerical methods selected for efficient mapping to vector hardware
- Navigate tradeoffs differently: e.g., may avoid control-flow divergence at cost of more arithmetic.

Contributions

- Ported SLEEF, an open-source vector math library, to the RISC-V Vector ISA (RVV).
- Described key RVV features to optimize code for nonlinear functions on SiFive X280:
  - Dynamic vector length
  - Register grouping
  - Vector-scalar instructions
  - Broadcasting with zero-strided loads
  - Mixed-width arithmetic instructions
- Demonstrated FP32 performance benefits of optimized code against SLEEF and Newlib at <1 ulp maximum error.
In this talk

- Short introduction to RISC-V and its Vector Extension
- Overview of SiFive’s X280 processor
- Case-study I: polynomial evaluation
- Case-study II: Cody-Waite range reduction
- Accuracy and Performance Evaluation
- Perspectives
RISC-V Instruction Set Architecture
A modular, extensible, open standard ISA

- Started c. 2010, the fifth generation of reduced instruction set computing (RISC) research projects at UC-Berkeley.
- RISC-V International, a Swiss nonprofit, owns and maintains the specs.
- No restrictions on use of the ISA for development of hardware or software.
- RISC-V Vector extension (RVV) ratified in November 2021.

Programming model

<table>
<thead>
<tr>
<th>32 integer registers</th>
<th>32 floating point registers</th>
<th>32 vector registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0-x31</td>
<td>f0-f31</td>
<td>v0-v31</td>
</tr>
</tbody>
</table>

- **Scalable** vector register length (VLEN): 128b to 64kib.
- Vector registers can be dynamically regrouped into longer logical registers (LMUL = length multiplier).
- Dynamic vector length (vl) simplifies loop stripmining.
SiFive’s X280 Processor

Dual-Issue In-Order Scalar Pipeline with Loosely-Coupled Vector Unit

- Main focus of our tuning
- RV64GCV, Zfh, Zvfh, and more
- Vector & scalar pipelines share common memory system
- Vector execution happens after scalar commit
- Three vector pipelines:
  - Load, store, ALU
  - Operate independently

![Diagram of the SiFive’s X280 Processor](image-url)
Case Study I: Polynomial Evaluation

Splatting Coefficient Operands

- Polynomials at core of nonlinear functions:
  \[ P(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_0 \]

- Many ways to evaluate:
  - Estrin’s Scheme increases parallelism
  - Horner’s Scheme minimizes # operations

- Inductively computes \( P_0 \) as
  \[ P_n = c_n \; ; \; P_i = c_i + x P_{i+1} \; ; \; P_0 = P(x) \]

- Form chain of dependent fused mul-adds:
  \[
  \begin{align*}
  P_{n-1} &= \text{FMA}(x, P_n, c_n) \\
  P_{n-2} &= \text{FMA}(x, P_{n-1}, c_{n-1}) \\
  \cdots
  \end{align*}
  \]

  Each 1 instruction

  Vector Values | Scalar Value

- Problem: no RVV FMA with scalar addend

- Consider three possible solutions...

Option 1: Scalar→Vector Move

\[
\begin{align*}
\text{vfmv.v.f} & \quad v12, f0 \quad # v12[i] = c0 \\
\text{vfmadd.vv} & \quad v8, v4, v12 \quad # P(x)x + c0
\end{align*}
\]

Both occupy 1 vector ALU on X280 (½ throughput)

Option 2: Hoist Moves out of Loop:

\[
\begin{align*}
\text{vfmv.v.f} & \quad v12, f0 \quad # v12[i] = c0 \\
\text{vfmv.v.f} & \quad v16, f1 \quad # v16[i] = c1 \\
\text{vfmadd.vv} & \quad v8, v4, v12 \quad # P(x)x + c0 \\
\text{vfmadd.vv} & \quad v8, v4, v16 \quad # P(x)x + c1
\end{align*}
\]

Too much register pressure for high degree

Option 3: Splat from Mem. w/ 0-Stride Load:

\[
\begin{align*}
\text{v1se32} & \quad v12, (t0), \text{zero} \\
\text{vfmadd.vv} & \quad v8, v4, v12 \quad # P(x)x + c0
\end{align*}
\]

Load and FMA use sep. pipes in parallel
Case Study I: Polynomial Evaluation

Choosing a Vector Length Multiplier (LMUL)

- Now splat+FMA operate in parallel:
  
  ```
  vlse32    v12, (t0), zero
  vfmadd.vv v8, v4, v12 # P(x)*x + c0
  vlse32    v16, (t1), zero
  vfmadd.vv v8, v4, v16 # P(x)*x + c1
  ...
  ```

- But: each FMA depends on prior’s result
  - can’t enter pipeline until prior’s 1st beat exits
  - ⇒ 6 stall cycles / term @ LMUL=1

- How to fix:
  - Unroll stripmine loop, interleave iterations... or
  - Increase LMUL to 4 ⇒ 8 beats / instruction

(LMUL * VLEN / DLEN) >= Pipeline Depth * Hides Latency
Case Study I: Polynomial Evaluation

Alternative Design Choices

- Might make different choices on other platforms
- If more than 1 ALU, would want to:
  - Unroll & interleave stripmine iterations
  - Replace Horner with Estrin or “Higher-Order Horner” (accuracy consequences)
- If cheap scalar → vector splat:
  - From micro-arch. tricks
  - Or from 2nd non-FMA ALU
  - Prefer vfmv.v.f (1) to other methods
- If no simultaneous LD+FMA: prefer “hoisting” (2)
- If OoO execution: probably still keep LMUL≥4!
  - Good for energy reasons

Option 1: Scalar→Vector Move

```plaintext
vfmv.v.f  v12, f0  # v12[i] = c0
vfmadd.vv v8, v4, v12  # P(x)*x + c0
```

Option 2: Hoist Moves out of Loop:

```plaintext
vfmv.v.f  v12, f0  # v12[i] = c0
vfmv.v.f  v16, f1  # v16[i] = c1
...
stripmine_loop:
  vfmadd.vv v8, v4, v12  # P(x)*x + c0
  vfmadd.vv v8, v4, v16  # P(x)*x + c1
...
```

Option 3: Splat from Mem. w/ 0-Stride Load:

```plaintext
vlse32    v12, (t0), zero
vfmadd.vv v8, v4, v12  # P(x)*x + c0
```
Case Study II: Range Reduction

Using mixed-width arithmetic for Cody-Waite range reduction

- \( r = x - zC \) reduction in FP32 precision
  - scalar float constants \( f0+f1+f2 \sim -C \)

- Use RVV vector-scalar FMA to accumulate first term:
  - `vsetvli s0,a1,e32,m4`
  - `vfmacvf v4,f0,v8 # x+z*f0`

- Use vector-scalar widening multiply for second term:
  - `vfwmul vf v16,v8,f1 # z*f1 (f64)`

- Accumulate intermediates with mixed-width addition:
  - `vfawdadd wv v16,v16,v4 # x+z*f0+z*f1 (f64)`

- Accumulate last term with widening vector-scalar FMA:
  - `vfmacvf v16,f2,v8 # ...+z*f2 (f64)`

- Continue on with double-precision reduced argument!
  - `vsetvli zero,zero,e64,m8`
SiFive Kernel Library

Basically a vectorized C17 `<math.h>`

- IEEE-754 FP16, FP32, FP64 precisions
- Target accuracy: < 1 ulp (RNE only)
- Trigonometric functions
- Hyperbolic functions
- Exponential and logarithmic functions
- Power and absolute-value functions
- Error functions
- Nearest integer functions
- Remainder functions
- Manipulation functions
- etc.
Accuracy evaluation

Comparing apples to apples

- Exhaustive testing whenever reasonable (FP16 and univariate FP32)
- Pseudo-random testing otherwise
- Uncovered accuracy issues in SLEEF routines

FP32 accuracy measurements
Accuracy evaluation

Comparing apples to apples

- Exhaustive testing whenever reasonable (FP16 and univariate FP32)
- Pseudo-random testing otherwise
- Uncovered accuracy issues in SLEEF routines

FP64 accuracy measurements

![Bar chart showing accuracy measurements for different functions with SLEEF and SiFive logos.](chart.png)
Accuracy evaluation

Comparing apples to apples

- Exhaustive testing whenever reasonable (FP16 and univariate FP32)
- Pseudo-random testing otherwise
- Uncovered accuracy issues in SLEEF routines
SiFive x280 Performance Evaluation

FP32

- Geomean of 18.9x speedup vs Scalar SLEEF
- ~11x speedup over Scalar Newlib
- ~3x speedup over RVV SLEEF
SiFive x280 Performance Evaluation

FP64

- Geomean of 8.9x speedup over Scalar SLEEF
- ~6x speedup over Scalar Newlib
- ~2x speedup over RVV-vectorized SLEEF
SiFive x280 Performance Evaluation

FP16

- No FP16 library to compare to
- Compare against our FP32 implementations
- Geomean of 2.2x speedup over FP32

Reciprocal throughput speedup over RVV SiFive FP32

RVV SiFive

x SiFive FP32 reciprocal throughput

acos_f16, asin_f16, atan_f16, atan2_f16, cbt_f16, cos_f16, cosh_f16, div_f16, erf_f16, erfz_f16, exp_f16, exp2_f16, expm1_f16, frexp_f16, hypot_f16, idexp_f16, log_f16, log10_f16, log2_f16, logb_f16, nextafter_f16, pow_f16, rcp_f16, rint_f16, rsqrt_f16, sinh_f16, sqrt_f16, tan_f16, tanh_f16, geomean
Conclusions and Software Availability

- Our RVV implementations incorporated into SiFive Kernel Library
  - Currently proprietary, but planned to be open-sourced.
- SLEEF RVV port PR'ed to upstream code repository
  - [https://github.com/sifive/sifive-sleef/tree/add-riscv-v-support](https://github.com/sifive/sifive-sleef/tree/add-riscv-v-support)
  - Accuracy issues reported to upstream project
- Key application: autovectorization
  - Compiler fuses independent scalar libm calls into a vector libcall
  - SiFive intern Hannah Leung (UIUC) gave proof-of-concept using LLVM’s TargetLibraryInfo
- Related work (see paper for more references):
  - CORE-MATH (correct rounding) [https://core-math.gitlabpages.inria.fr](https://core-math.gitlabpages.inria.fr)
- Future work:
  - Finish coverage of C/C++ math library at current accuracy
  - Reduced accuracy implementations (esp. for machine learning)
  - Help design future RISC-V ISA extensions
Thank you
Additional Slides
SiFive x280 Performance Evaluation

Reciprocal throughput speedup over Scalar Sleef

- x Scalar Sleef reciprocal throughput
  - exp
  - exp_w
  - log
  - log_w
  - sin
  - sin_w
  - erf
  - erf_w
  - geomean

Legend:
- Newlib
- Sleef m1
- Sleef m2
- SiFive m1
- SiFive m2
- SiFive m4
- SiFive m8
Accuracy Evaluation

Maximum ULP error (float32)