RISC-V Vectorization Coverage for HPC: A TSVC-Based Analysis

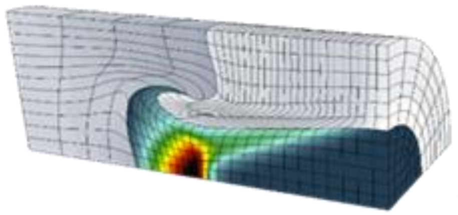
RISCV-HPC @ SC25

11/17/2025

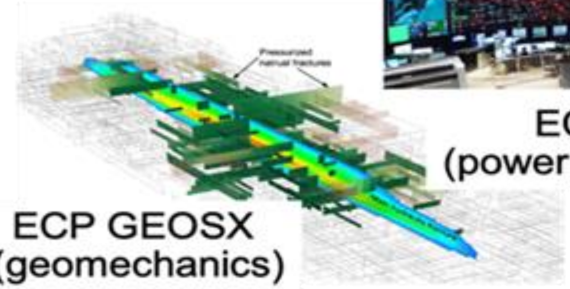
Hung-Ming Lai[1,4], **Pei-Hung Lin[1]**, Maya B. Gokhale[1], Ivy Peng[2], Hiren Patel[3], Jenq-Kuen Lee[4]
Center for Applied Scientific Computing LLNL[1]
KTH[2], University of Waterloo[3], NTHU[4]

LLNL-CFPRES-2013400
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

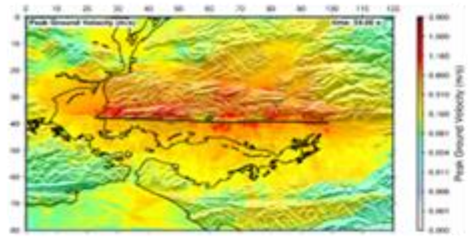


LLNL ECP/ATDM
(high-order ALE hydro)



ECP GEOSX
(geomechanics)

ECP ExaSGD
(power grid optimization)



ECP SW4
(earthquake modeling)

plus others...

Perlmutter (LBL)
AMD Milan CPUs +
NVIDIA Ampere GPUs

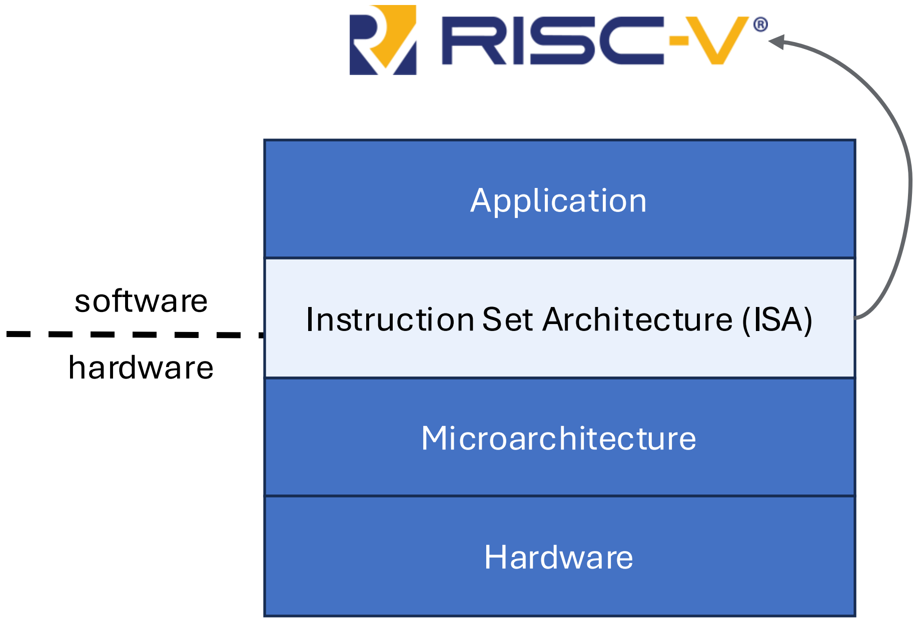
Astra (SNL)
ARM architecture

Sierra (LLNL)
IBM P9 CPUs + NVIDIA Volta GPUs

Aurora (ANL)
Intel Xeon CPUs + Xe GPUs

Frontier (ORNL) &
El Capitan (LLNL)
AMD CPUs + GPUs

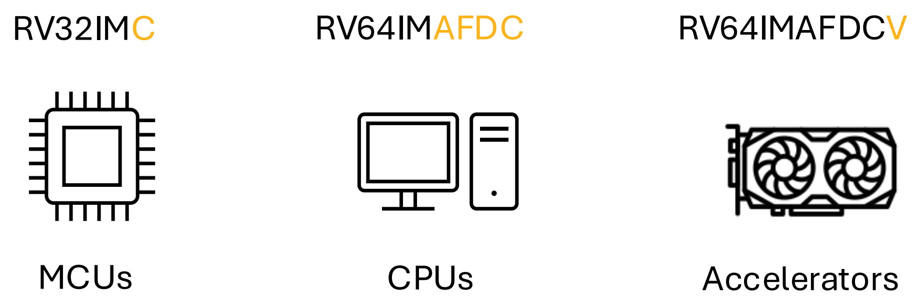
The potential of RISC-V's open, modular design in next-gen HPC



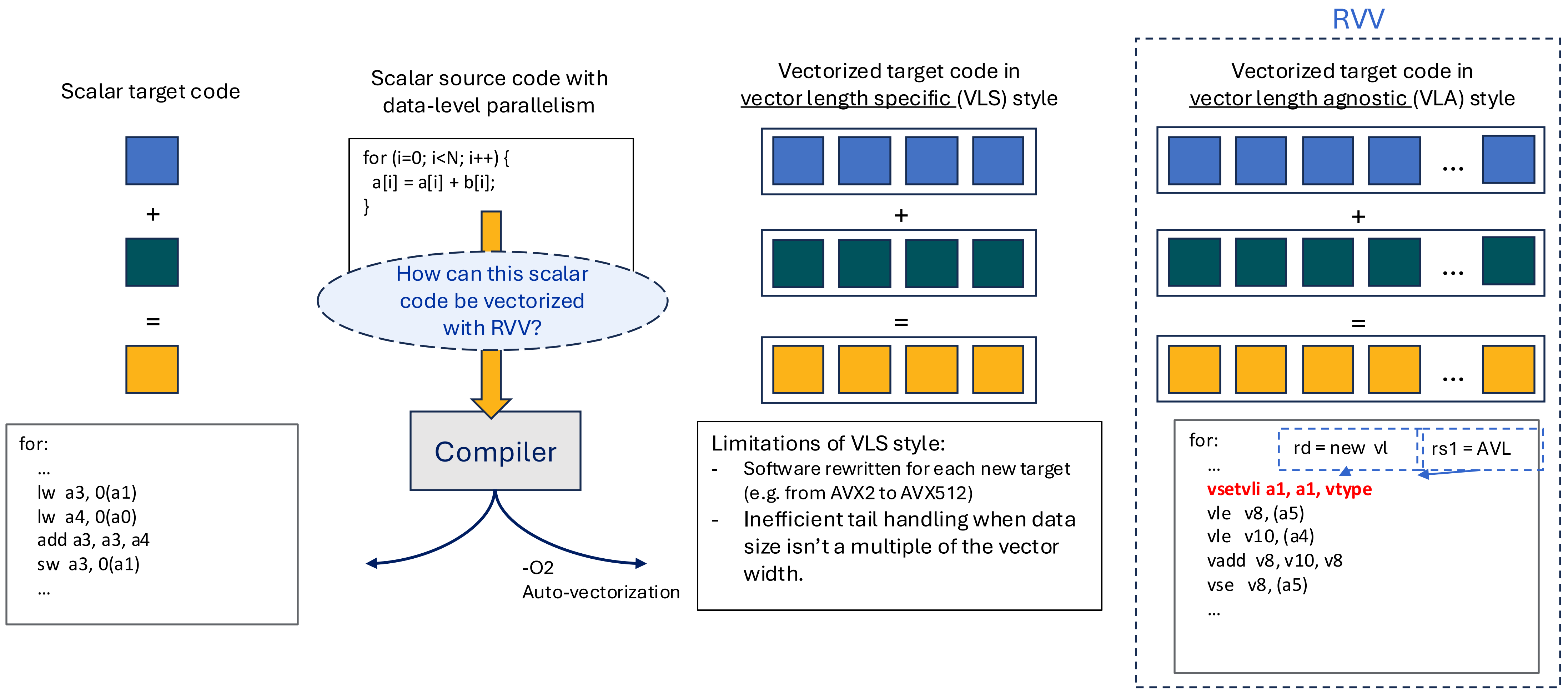
- Proprietary vs Open-standard ISA

	Proprietary (x86, Arm, ...)	Open-standard (RISC-V)
Licensing Cost	Expensive, restricted	Free and royalty-free
Customizability	Limited or restricted	Fully customizable
Vendor Lock-in	High	Low — broad vendor ecosystem
Innovation Speed	Slower, centralized	Fast, community-driven

- Modular ISA Extensions



Scalable vectorization with RISC-V vector extension and compiler auto-vectorization


RVV
Scalar target code
Scalar source code with data-level parallelism
Vectorized target code in vector length specific (VLS) style
Vectorized target code in vector length agnostic (VLA) style
+
=
…
How can this scalar code be vectorized with RVV?
Compiler
-O2
Auto-vectorization
rd = new vl
rs1 = AVL

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

```
for:
   …
   lw  a3, 0(a1)
   lw  a4, 0(a0)
   add a3, a3, a4
   sw  a3, 0(a1)
   …
```

Limitations of VLS style:

- Software rewritten for each new target (e.g. from AVX2 to AVX512)
- Inefficient tail handling when data size isn't a multiple of the vector width.

```
for:
   …
   vsetvli a1, a1, vtype
   vle  v8, (a5)
   vle  v10, (a4)
   vadd v8, v10, v8
   vse  v8, (a5)
   …
```

TSVC: Test Suite for Vectorizing Compilers

- A benchmark designed to evaluate compiler auto-vectorization capabilities.
  - Originated in the 80's
  - loops derived from real-world and synthetic codes
  - grouped by dependency and control-flow characteristics
  - Part of LLVM test suite

- Statistics
  - ~151 loop kernels across control flow, memory access, and arithmetic patterns
  - Covers scalar, array, pointer-based, and conditional loops
  - Used by research and industrial compiler teams to assess vectorization performance

- Applied in studies for various CPUs

Methodology

RVV Instruction Coverage Analysis

Missed Instruction Identification

- Compiler backend or pattern-matching limitations
- Lack of representative use cases
- Non-trivial vectorization requirements or unsupported cases

Future Benchmark Enhancements

Key Results - RVV Coverage Statistics

Instruction group	config-setting	Load & Store	Integer	Fixed-Point	Floating-Point	Reduction	Mask	Permutation	Sum of all groups
RVV Instruction count	3	21	55	13	41	16	15	15	179
GNU-VLA	2 (66%)	10 (47.6%)	9 (16.4%)	0 (0%)	10 (24.4%)	1 (6.3%)	1 (6.7%)	8 (53.3%)	41 (22.9%)
GNU-VLS	2 (66%)	11 (52.4%)	9 (16.4%)	0 (0%)	12 (29.3%)	1 (6.3%)	1 (6.7%)	8 (53.3%)	44 (24.6%)
LLVM-VLA	2 (66%)	10 (47.6%)	10 (18.2%)	0 (0%)	17 (41.4%)	3 (18.8%)	1 (6.7%)	9 (60%)	52 (29.1%)
LLVM-VLS	2 (66%)	10 (47.6%)	10 (18.2%)	0 (0%)	17 (41.4%)	3 (18.8%)	1 (6.7%)	8 (53.3%)	51 (28.5%)

Key Results - Detailed Observations

- Config-setting: Only vsetvli/vsetivli observed, not vsetvl
- Load/Store: Missed mask register, segmented, and advanced indexed instructions
- Integer: No widening instructions; limited by TSVC's floating-point focus
- Fixed-point: No coverage due to lack of kernels
- Floating-point: Missing widening and FMA instructions; compilers can generate with suitable patterns
- Reduction: Basic reductions covered, but widening and advanced forms missing
- Mask: Only vid observed; advanced mask ops absent
- Permutation: vslideup/vslidedown present; single-element and some merge instructions missing

Key Results - VLA vs. VLS

- Minor differences between modes
- GNU VLS: Uses vlm for mask loads, FMA instructions
- LLVM: vrgatherei16 in VLA, vrgather with 32-bit indices in VLS
- Differences:
  - Load & Store: **vlm** employed in VLS by GNU
  - Floating-Point
    - GNU VLS: vfmadd, vfmacc
    - GNU VLA: vfmul and vfadd
  - Permutation
    - GNU VLA: vrgatherei16
    - GNU VLS: vrgather

Key Results - Vector Length (VL) in VLS

- Coverage stable across vector lengths except for wide vectors (GNU emits vmv.s.x for VL ≥ 512)
  - S4116: sum += a[off] * aa[j -1][ip[i]];
  - Code generation for reductions under VLS differs based on vector length (VL):
    - For VL < 512: Uses scalar floating-point register for accumulation, seeded with vfmv.s.f and updated via vfmv.f.s.
    - For VL ≥ 512: Keeps accumulator in vector registers, initialized with vmv.s.x and reduced with vfredosum.vs, avoiding scalar register traffic.

Classification of Auto-Vectorization Gaps

Key Failure Causes:

- Unidentified PHI
  - — cyclic data dependencies between instructions
  - — non-predictable control-flow variables
- Memory Conflict
  - — conservatively assumed aliasing
  - — Overlapping memory access patterns
- Switch Statement
- Non-intrinsic Call
- Early Exit

Failure Causes

Cause (# of loops)	loop
Unidentified PHI (37)	s116, s1213, s123, s126, s13110, s141, s161, s211, s2111, s212, s221, s222, s2233, s2251, s231, s232, s233, s235, s242, s256, s258, s261, s275, s277, s291, s292, s3110, s3112, s315, s318, s321, s322, s323, s331, s341, s342, s343
Memory conflict (7)	s1113, s114, s1161, s1244, s241, s244, s281
Switch statement (1)	s442
Non-intrinsic call (1)	va
Early exit (3)	S332, s481, s482

First failure cause encountered by the LLVM LoopVectorize pass for each non-vectorized loop in TSVC

Failure Cause: Unidentified PHI (cont'd)

- cyclic data dependencies between instructions

```
// s221
for (int i = 1; i < LEN_1D; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i - 1] + a[i] + d[i];
}
```

```
LV: Not vectorizing: Found an unidentified PHI
%0 = phi float [ %.pre, %for.cond2.preheader ], [ %6,
%for.body5 ]
```

- non-predictable control-flow variables

```
// s161
for (int i = 0; i < LEN_1D-1; ++i) {
  if (b[i] < (real_t)0.) {
    goto L20;
  }
  a[i] = c[i] + d[i] * e[i];
  goto L10;
L20:
  c[i+1] = a[i] + d[i] * d[i];
L10:
  ;
}
```

control-flow pattern not recognized

```
LV: Not vectorizing: Found an unidentified PHI
%indvars.iv = phi i64 [ 0, %for.cond2.preheader ], [
%indvars.iv.next.pre-phi, %for.inc ]
```

Examples of Other Failure Causes

- Memory conflict

```
// s1113
for (int i = 0; i < LEN_1D; i++) {
    a[i] = a[LEN_1D/2] + b[i];
}
```

- Switch statement

```
// s442
for (int nl = 0; nl < iterations/2; nl++) {
    for (int i = 0; i < LEN_1D; i++) {
        switch (indx[i]) {
            case 1:  goto L15;
            case 2:  goto L20;
            case 3:  goto L30;
            case 4:  goto L40;
        }
L15:
    ...
}
```

- Non-intrinsic call

```
// va
for (int i = 0; i < LEN_1D; i++) {
    a[i] = b[i];
}
```

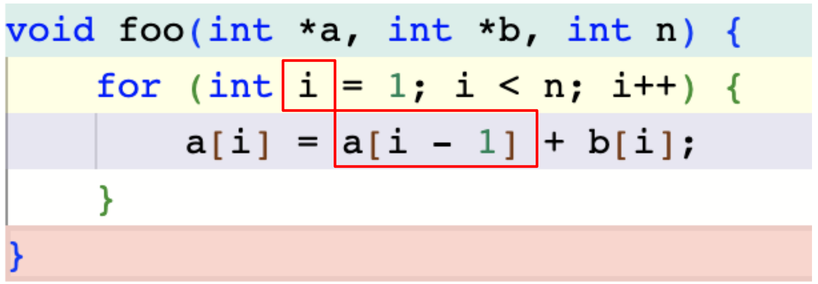
whole loop optimized to @llvm.memcpy by previous passes

- Early exit

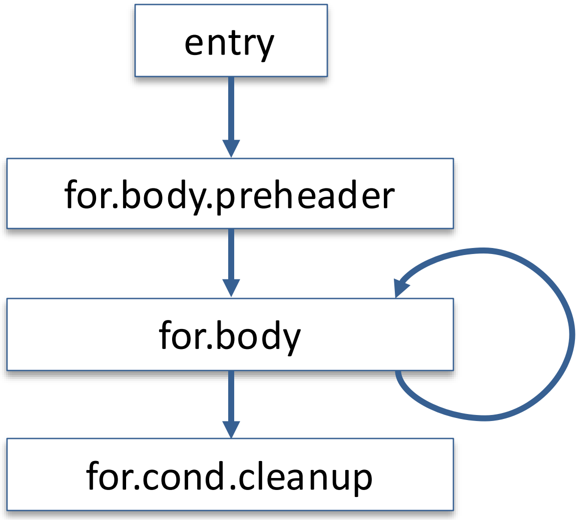
```
// s482
for (int i = 0; i < LEN_1D; i++) {
    a[i] += b[i] * c[i];
    if (c[i] > b[i]) break;
}
```

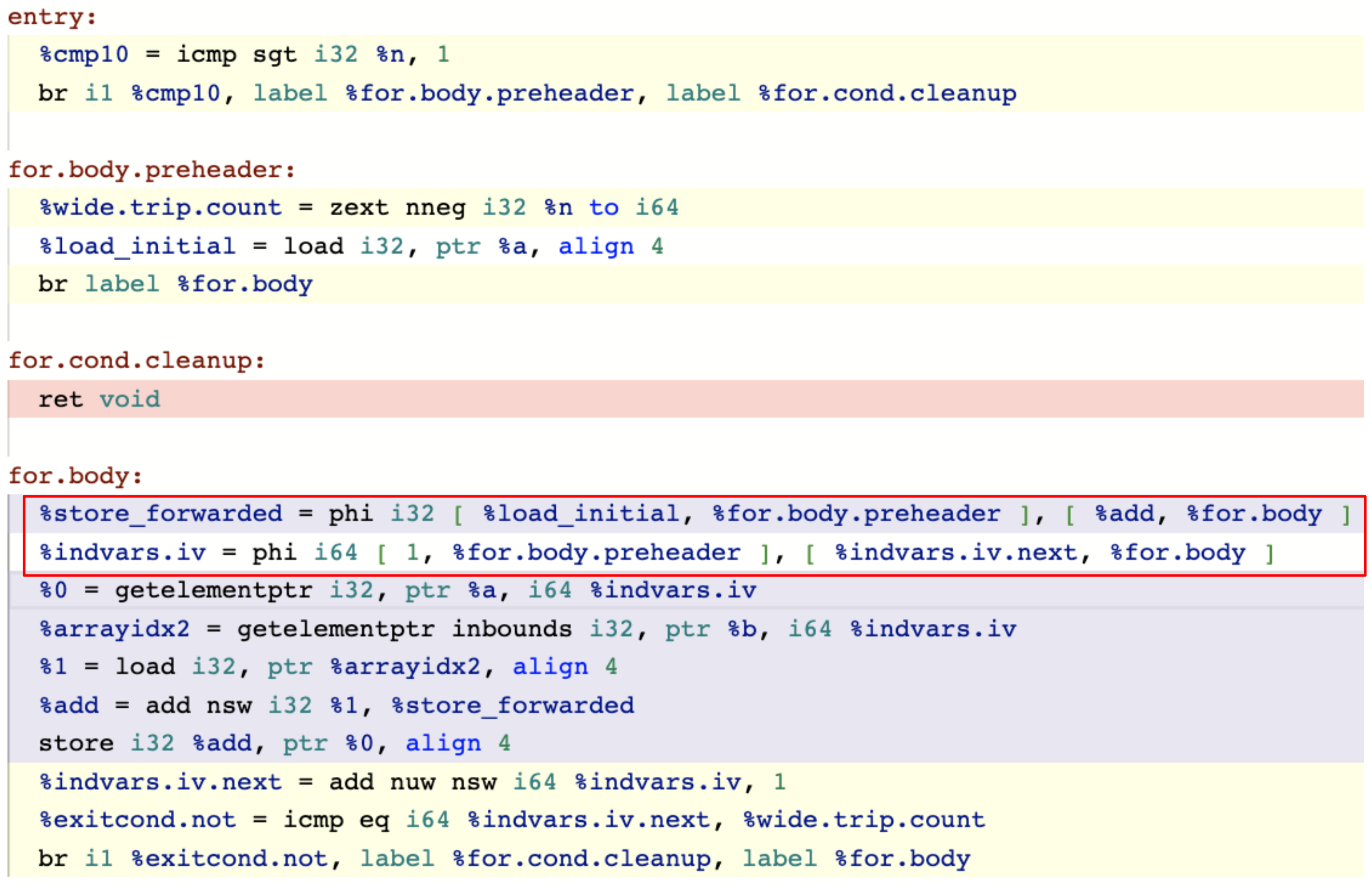
Failure Cause: Unidentified PHI

- SSA PHI nodes from a loop



```
void foo(int *a, int *b, int n) {
    for (int i = 1; i < n; i++) {
        a[i] = a[i - 1] + b[i];
    }
}
```





```
entry:
  %cmp10 = icmp sgt i32 %n, 1
  br i1 %cmp10, label %for.body.preheader, label %for.cond.cleanup

for.body.preheader:
  %wide.trip.count = zext nneg i32 %n to i64
  %load_initial = load i32, ptr %a, align 4
  br label %for.body

for.cond.cleanup:
  ret void

for.body:
  %store_forwarded = phi i32 [ %load_initial, %for.body.preheader ], [ %add, %for.body ]
  %indvars.iv = phi i64 [ 1, %for.body.preheader ], [ %indvars.iv.next, %for.body ]
  %0 = getelementptr i32, ptr %a, i64 %indvars.iv
  %arrayidx2 = getelementptr inbounds i32, ptr %b, i64 %indvars.iv
  %1 = load i32, ptr %arrayidx2, align 4
  %add = add nsw i32 %1, %store_forwarded
  store i32 %add, ptr %0, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.cond.cleanup, label %for.body
```

compiled LLVM IR

Failure Cause: Unidentified PHI (cont’d)

- “Identified” PHIs
  - induction PHI
    - induction scalar → value of i
    - induction pointer → pointer of a[i] and b[i]
  - reduction PHI → value of sum
  - first-order recurrence (PHI from optimized load with no actual data dependence) → value of a[i – 1]

```
for (int i = 1; i < n; ++i) {
  sum += a[i];
  b[i] = a[i] - a[i - 1];
}
```

Auto-Vectorization Gaps: Root Causes and Implications

1st Core Barrier:

- Control Flow Complexity
  - Divergent paths (e.g., branches, switches, early exits) prevent lockstep parallel execution.
  - Modern ISAs (e.g., RVV) mitigate via masking/predication.
  - Compilers can apply if-conversion, but success depends on control simplicity.
  - Complex cases may require speculation or explicit restructuring.

Auto-Vectorization Gaps: Root Causes and Implications

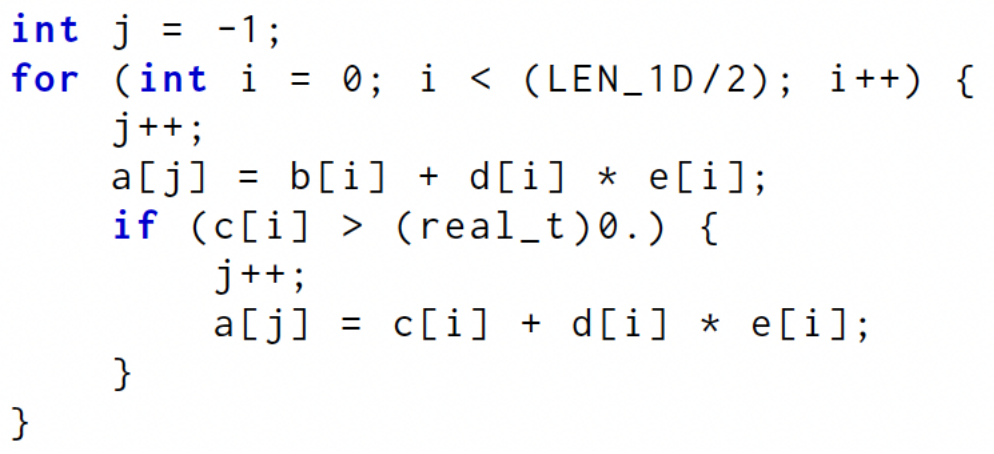
2nd Core Barrier:

- Data Dependence Ambiguity
  - Must ensure vectorization doesn't violate loop-carried dependencies.
  - Failures like Unidentified PHI and Memory Conflict stem from:
    - Unclear SSA dependencies
    - Insufficient aliasing analysis
  - Often not real barriers, but compiler conservatism.
  - Mitigation techniques:
    - Loop interchange – reorder iterations
    - Loop distribution – isolate vectorizable sections
    - Prefix-sum transformations – handle recurrences

Missed Instruction Analysis in TSVC

- "not vectorizable" due to a loop-carried dependency on j
- j depends on how many earlier iterations satisfy c[i] > 0
- This creates a scalar dependency chain → sequential updates only
- Conventional SIMD auto-vectorizers cannot break this dependency

- TSVC loop s123



```
int j = -1;
for (int i = 0; i < (LEN_1D/2); i++) {
    j++;
    a[j] = b[i] + d[i] * e[i];
    if (c[i] > (real_t)0.) {
        j++;
        a[j] = c[i] + d[i] * e[i];
    }
}
```

RVV Solution: Masked Prefix-Sum Instructions

- RISC-V Vector Extension (RVV) introduces hardware support for per-lane index computation:

Instruction	Function	Purpose
viota.m	masked exclusive prefix sum	gives each active lane its own offset
vcpop.m	population count of mask	advances base index for next vector

- Mask: m = (c > 0)
- Offsets: vIO = viota.m(m)
- Base advance: j_base += vl + vcpop.m(m)

Vectorized Form and Compiler Insights

- Recognize prefix-sum recurrence on induction variables.
- Apply if-conversion + masked index lowering.
- Extend LLVM/GCC to map these patterns to viota.m / vcpop.m.

- Vectorized pseudocode (simplified):

```
while (p < N/2) {
  vl = vsetvl(N/2 - p);
  vB = vle(b+p); vC = vle(c+p);
  m  = vgt(vC, 0);
  vIO = viota.m(m);
  idx_b = j_base + vid() + vIO;
  vsuxei(a, idx_b, vB + d*e);
  idx_c = idx_b + 1;
  vsuxei.m(a, idx_c, vC + d*e, m);
  j_base += vl + vcpop.m(m);
  p += vl;
}
```

Conclusions & Future Directions

- TSVC covers basic RVV instructions, but many advanced features missed
- Benchmark ambiguity complicates evaluation
- Recommend:
  - Extending TSVC for modern vector ISAs
  - Clearer vectorizability criteria
  - Benchmarks inspired by RAJA, Kokkos
- Systematic assessment framework needed for compiler and architecture co-design

Lawrence Livermore National Laboratory