# Improvements to RISC-V Vector code generation in LLVM

Luke Lau and Alex Bradbury

Igalia

**Abstract**

*Improvements to generated code quality when targeting the RISC-V vector extension have been a major focus of recent development efforts within the RISC-V LLVM backend. This contribution summarises recent advances, outlines planned next steps, and presents benchmark results. The combination of these enhancements gives a geometric mean 8.7% performance improvement in Clang 20 over Clang 17 as measured on the SpacemiT X60.*

## Overview

LLVM has had stable support for the RISC-V vector extension for the past several releases, with both forms of autovectorization — the loop vectorizer and the SLP (superword-level-parallelism) vectorizer — enabled by default, as well as a rich set of C intrinsics for the RVV programming model. A key focus since then has been on improving generated code performance. The following sections explain the changes made in the middle-end and backend that have delivered these gains.

## SLP improvements

The SLP vectorizer was originally designed for traditional SIMD architectures like NEON and SSE, and previously only supported power-of-two vectorization factors that fit exactly into a register.

On RVV, `vl` enables vectors of arbitrary sizes, and in LLVM 20 SLP can take advantage of this to vectorize more straight-line sequences that aren't powers-of-two, e.g. an RGB pixel:

```
struct rgb { float r,g,b; };
void brighten(struct rgb *x, float f) {
    x->r *= f;
    x->g *= f;
    x->b *= f;
}

brighten:
        vsetivli zero, 3, e32, m1, ta, ma
        vle32.v v8, (a0)
        vfmul.vf v8, v8, fa0
        vse32.v v8, (a0)
        ret
```

## Loop vectorizer improvements

The RISC-V backend gained better support for bfloat16 and FP16 types, specifically in the presence of zvfbfmin and zvfhmin where the they need to be widened to FP32 first. This in turn allows the loop vectorizer to vectorize more bfloat16 and FP16 loops:

```
void f(float *dst, __bf16 *a, __bf16 *b) {
  for (int i = 0; i < 1024; i++)
    dst[i] += ((float)a[i] * (float)b[i]);
}

    vsetvli t4, zero, e16, m1, ta, ma
.LBB0_4:
    vl1re16.v   v8, (t3)
    vl1re16.v   v9, (t2)
    vl2re32.v   v10, (t1)
    vfwmaccbf16.vv  v10, v8, v9
    vs2r.v  v10, (t1)
    add t3, t3, a4
    add t2, t2, a4
    sub t0, t0, a6
    add t1, t1, a7
    bnez    t0, .LBB0_4
```

Extra care needed to be taken in the cost model since any widened bfloat16/FP16 will take up twice the LMUL, impacting both throughput and register pressure.

The loop vectorizer also learnt how to emit VLA (vector-length-agnostic) segmented loads and stores for factors of 4 and 8. Previously it could only handle factors of 2 because there wasn't a way to represent arbitrary permutations for VLA vectors, but it's now able to combine multiple interleave intrinsics to represent higher factors.

## Inlining of memcmp

LLVM had previously supported inlining calls to libc's `memcpy`/`memset` into vector loads and stores, but it also recently gained the ability to inline `memcmp` into vector instructions in cases where this is expected to be cheaper than calling the libc function:

```
int equal(char *a, char *b) {
    return memcmp(a, b, 16) == 0;
}

equal:
    vsetivli zero, 16, e8, m1, ta, ma
    vle8.v v8, (a0)
    vle8.v v9, (a1)
    vmsne.vv v8, v8, v9
    vcpop.m a0, v8
    seqz a0, a0
    ret
```

## `vsetvli` optimization

The pass that handles inserting `vsetvli` instructions previously took place before register allocation. This had the unintended side effect of preventing the scheduler from moving vector instructions across changes to `vl` or `vtype`.

To remedy this, register allocation was split into two separate passes for scalars and vectors. The `vsetvli` insertion pass now runs after vector register allocation, but before scalar register allocation, as a `vsetvli` may need to use a scalar register to set `vl=VLMAX`.

As a result, scheduling for vector instructions is now improved, and LLVM can rematerialize vector instructions that have been spilled. An interesting artefact of the separate scalar and vector register allocations is that it's possible to rematerialize non-constant instructions like `vmv.v.x`.

Additionally, a new pass was added that reduces the `vl` of instructions to only what is demanded. This works in conjuction with the previous insertion pass to reduce the number of `vsetvli`s emitted, and improves performance on microarchitectures which are sensitive to `vl` (e.g. SiFive x280)

## Stripmined loops via `vl` tail folding

Initial support for generating stripmined loops via `vl` masking has been landed behind a flag. This required teaching the loop vectorizer to emit Vector Predication intrinsics (LLVM's target agnostic way of representing mask and vector length semantics). Work is ongoing to enable it by default.

```
.LBB0_2:
 sub a5, a1, a2
 sh2add a3, a2, a0
 vsetvli a5, a5, e32, m2, ta, ma
 vle32.v v8, (a3)
 sub a4, a4, a6
 vadd.vi v8, v8, 1
 vse32.v v8, (a3)
 add a2, a2, a5
 bnez a4, .LBB0_2
```

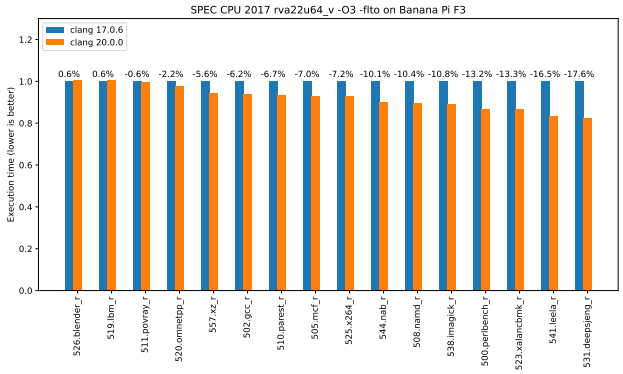


**Figure 1:** *SPEC CPU 2017 rate benchmarks on the train dataset on the BPI-F3 (SpacemiT X60), compiled with `-march=rva22u64_v -O3 -flto`*

Part of the difficulty in performing tail folding is due to the behaviour of `vsetvli` on the second to last iteration, where `vl` may be set to `ceil(AVL / 2)`. This invalidated assumptions in the loop vectorizer that only the last iteration needed masking.

## Performance results

SPEC CPU 2017 compiled with Clang 20.0.0 was measured to have a geometric mean 8.7% improvement over Clang 17.0.3 on the SpacemiT X60, shown in Figure 1.

Work on performance was aided by significant improvements to public CI, with expanded buildbot configurations and LLVM test-suite testing. Nightly performance testing on the SpacemiT X60 via LNT now also allows regressions to be caught within a day of landing.

## Upcoming and future work

While significant progress has been made, there is still a wide range of potential further optimisations, improvements, microarchitecture-specific tuning, and further features to be implemented especially as more RVV capable hardware becomes commercially available. Areas of planned work include:

- Fixing gaps in the cost model where compiling with RVV versus results in degraded performance compared to without RVV
- Proof-of-concept support for proposed extensions such as `zvzip` and `zvabd`
- Support for RVV in llvm-exegesis, LLVM's tool for automatically benchmarking instructions on hardware to drive scheduling models
- Support for VLEN=32, which isn't possible today due to the mapping of LLVM IR vector types to SEW and LMUL