VENTANA MICRO

# GCC 14 RISC-V Vectorization Improvements and Future Work

Dr. Robin Dapp
<rdapp@ventanamicro.com>

# Preface

- Focus on the RISC-V Vector Extension (RVV) and related improvements.

- Summary of the work of numerous contributors: RiVAI, Intel, SiFive, Rivos, ESWIN, StarFive, VRULL, Embecosm, RAU, Ventana and many others.
- Active community and several new contributors.  Can always use helping hands, so feel free to reach out, contribute.

# Improvements Since GCC 13

VENTANA MICRO

- Wired up all suitable auto vectorization primitives for integer/floating point; loads/stores, gathers, binary operations etc. **Loop and SLP vectorization work, GCC's vector testsuite passes.**
- Vectorized `memcpy`, `strlen`, `strcmp` etc.
- Vector calling convention.
- Vector crypto intrinsics, XTheadVector (RVV 0.7) integrated.
- OOO instruction scheduling model.
- Many improvements to the vsetvl pass, fully based on GCC's LCM.
- Dynamic register group size (LMUL) selection based on register-pressure estimation.
- Pre- and post-commit CIs.

# Vectorization Example

```
foo (int *x, int *y, int *z,
     int *pred, int n)
 {
   for (int i = 0; i < n; ++i)
     x[i] = pred[i] != 1
            ? y[i] + z[i]
            : y[i];
 }
```

Compiled with

```
gcc -march=rv64gcv -O3
```

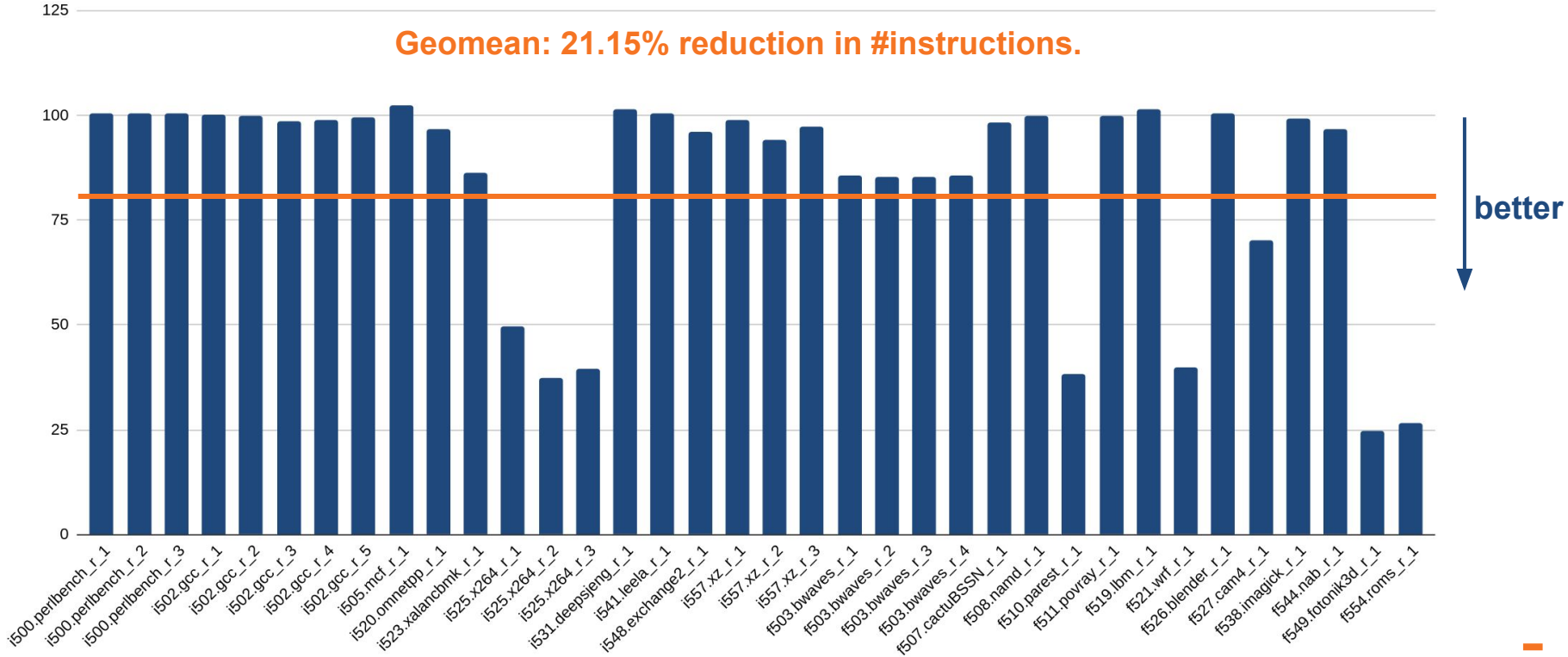
```
.L132:
vsetvli a5,a4,e32,m1,ta,mu
slli a6,a5,2
vle32.v v0,0(a3)
vle32.v v1,0(a1)
vmsne.vi v0,v0,1
vle32.v v2,0(a2),v0.t
vadd.vv v1,v2,v1,v0.t
vse32.v v1,0(a0)
add a3,a3,a6
add a1,a1,a6
add a2,a2,a6
add a0,a0,a6
sub a4,a4,a5
bne a4,zero,.L132
```

# Lessons Learned

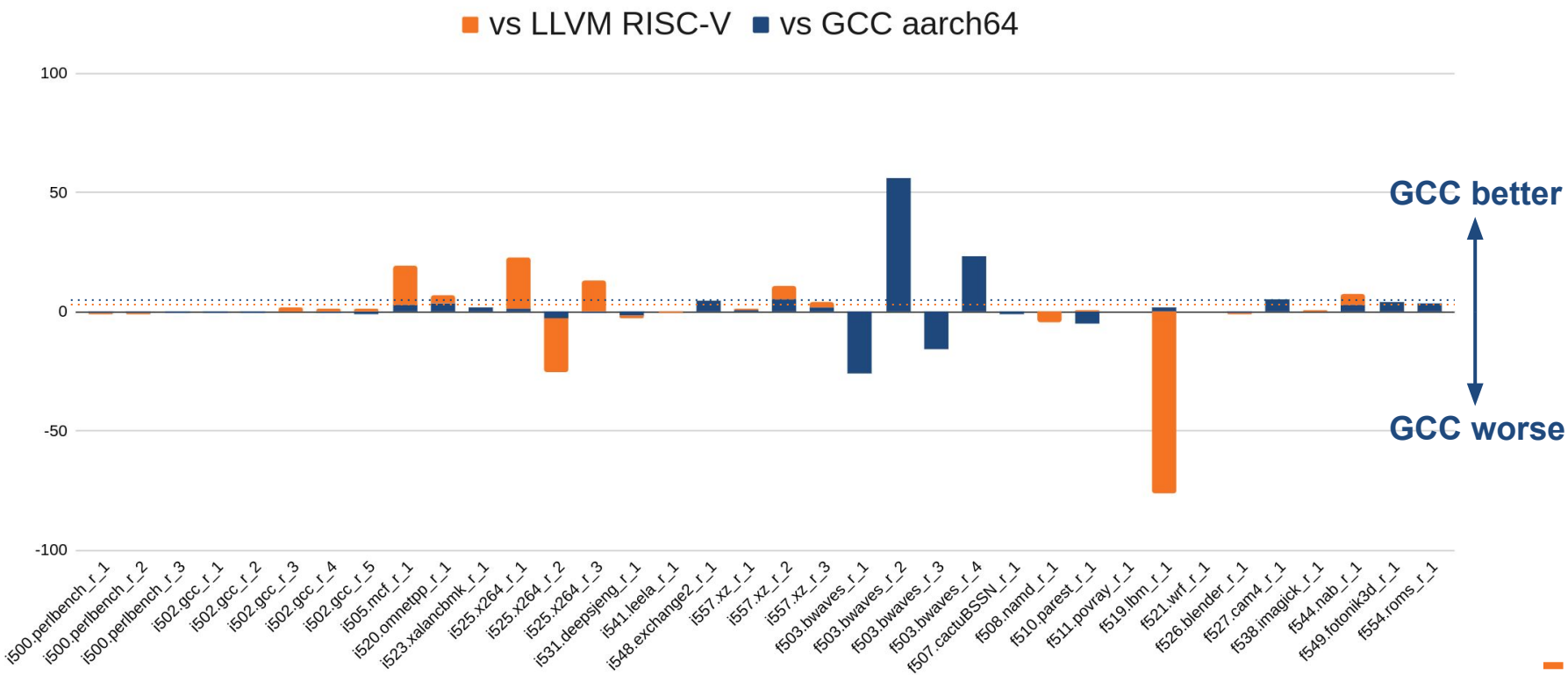
- GCC uses auto generated “instruction description” files. RVV requires huge number of instruction modes (due to LMUL) as well as operands and iterators.
- Caused generated files to blow up (almost 10x larger than next largest backend), bottleneck for compiler bootstrap time.
- Needed to adjust generators to split their output, also helps other backends.

- Vector mask implementation differs from other architectures, bit-”packing” was a source of many bugs.
- Uncovered some long-standing vectorizer bugs due to disabling of vector cost model for testing (thus vectorizing more).

# SPEC 2017 Vectorization, qemu icount



# SPEC 2017 Relative Improvement



# Performance and TODOs

**Takeaway:** RVV reduces #instructions by ~20% across SPEC2017. In line with what we expected and see on other architectures. Slightly better relative improvement than GCC aarch64 and LLVM RVV.

**TODOs GCC 15 and beyond:**

- *Strided load/store* support, helps 525.x264_r and 519.lbm_r.
  Known pain point in the vectorizer.
  Somewhat uarch dependent but LLVM does better here.
- Currently revisiting some known-bad vectorizer costing decisions, working on enhanced strided-load support.
- For 525.x264_r need to improve SLP discovery and scheduling, handle stores with gaps in vectorizer.

# Performance and TODOs (2)

- GCC 15 Transition to SLP-only representation of the vectorizer (long-standing issue) will help with codegen and also require adjustments.

- Vector cost model is very generic, barely uarch-specific tuning in place.  Expecting this to improve a lot once more uarchs are available for public testing.

- Overlap handling for register groups.
- Scalar evolution for `vsetvl`.

Some improvements have already made it into GCC 15:

# Saturating Arithmetic (GCC 15)

- *coremark-pro*'s zip-test (basically zlib) key loop uses saturating sub:

```
unsigned n, m;
do {
    m = *--p;
    *p = (Posf)(m >= wsize ? m - wsize : NIL);
} while (--n);
```

- LLVM has been supporting this for a while, GCC 15 will as well, roughly 10% improvement:

```
vrgather.vv
vnclipu.wi
vssubu.vv
vrgather.vv
```

# Early-Break Vectorization (GCC 15)

- The following is vectorized upstream, `vect_a's/vect_b's` size must be known:

```
for (int i = 0; i < N; i++)
  {
     vect_b[i] = x + i;
     if (vect_a[i] > x)
         break;
     vect_a[i] = x;
  }
```

- Common, frequently used, helps vectorization across the board. More generic case is done differently still:

# Fault-First Loads (GCC 15?)

- Right now we recognize certain loop idioms and manually implemented them “optimally”
  (e.g. vectorized 2-byte `rawmemchr` in 523.xalancbmk_r).
- Similarly, 2-byte `strcmp` possible, proof of concept in place.
  Lots of similar spots, e.g. *find* in 523.xalancbmk_r.
- LLVM went a similar route for hot loop in 557.xz_r:

```
while (++len != len_limit)
    if (pb[len] != cur[len])
        break;
```

- Commonality: All those can be vectorized with early-break vectorization but *must not* read beyond array bounds.
- Requires *fault-only-first load* support, being worked on.

# More to Come (GCC 15?)

- Combination of

```
vmv.v.x v8, a4          and
vop.v.v v2, v3, v8      into
vop.v.x v2, v3, a4.
```

  Need register-pressure aware propagation of `a4` as well as uarch-specific adjustments.

- Aggressive fast-math reassociation (benefits scalar but also vector).
- Vector Crypto Extension for auto vectorization: `vwsll, vandn`.

- `min/max` reduction, if-conversion for chained conditions.
- Better widening/narrowing support in GIMPLE.

# Thank You.

## For more details and discussion drop by at my poster.