Coding Challenge - LFX Mentorship Program

(RISC-V Audiomark)

Sohail Raj Satapathy
31st January 2026
Code: GitHub - https://github.com/sohail103/lfx-rv-audiomark-coding-challenge

Problem Statement - Requirements

To implement a saturating multiply-accumulate function accelerated using RISC-V vector intrinsics.

void q15_axpy_rvv(const int16_t *a, const int16_t *b, int16_t *y, int n, int16_t alpha);

Where $y[i] = sat\_q15(a[i] + \alpha \cdot b[i])$

(a[i], b[i] and y[i] are signed 16-bit integers and α is a signed 16-bit scalar)

Ensuring bit-for-bit correctness for all tested inputs and vector length agnosticism.

This document describes the approach taken to design, implement and evaluate the solution.

Approach

I began by analyzing the computational requirements of the function and the different operations it requires. I then read through the Vector Extension chapter of the RISC-V unprivileged ISA specification in order to understand the instructions and parameters provided by RVV like LMUL, VLEN, SEW, policies for masking and tail elements, etc. I then studied the RVV C Intrinsics v1.0 specification to understand intrinsic naming conventions, typed suffixes, masking and policy variants, the use of `vsetvl` and `vsetvlmax` and fixed-point rounding behavior controlled by `vxrm`. After this, I searched for the relevant intrinsics that would best match the operations performed in the scalar reference implementation.

The scalar reference performs the following operations per element:

1. Load a[i] and b[i] (16-bit values)
2. Widen both operands to 32 bits
3. Multiply b[i] by alpha
4. Add the result to a[i]
5. Saturate the result to the Q15 range
6. Store the result

The multiplication produces a widened result, which requires addition to be performed in the wider type. Narrowing and saturation cannot be performed earlier because the scalar reference applies saturation only after the addition. To guarantee bit-for-bit equivalence, the vector implementation must preserve the same arithmetic ordering.

Thus, the vectorized sequence follows the same general structure:

1. Load 16-bit values
2. Widen to 32-bit
3. Multiply and add in 32-bit
4. Narrow back to 16-bit with saturation
5. Store the result

A strip-mined loop structure was used to ensure vector length agnostic behavior. In each iteration, `vsetvl` determines how many elements can be processed based on the current hardware vector length and the chosen element width and LMUL. `vl` elements are loaded from arrays a and b. Vector operations are applied to these elements. The pointers and loop counter are advanced by `vl`. This structure guarantees correctness for any input size and allows the same binary to run on implementations with different vector lengths. I used the `vnclip` intrinsic to implement saturation to Q15, setting the rounding mode to `RNU` (Round to Nearest Up) which is the default rounding mode. Since the operation is a single stage vector computation for each element of 'y' without recursive accumulation, the small statistical bias of RNU does not compound to cause error drift. This overall approach handles narrowing and saturating faster than manual min/max clamping.

Intrinsic Selection and the Use of Widening Multiply-Accumulate

Instead of performing a widened multiplication followed by a widened addition, I used the widening multiply-accumulate intrinsic (`vwmacc`). This fuses the multiplication and addition into a single instruction, allowing the microarchitecture to exploit fused multiply-accumulate datapaths where available. A widening multiply followed by a widening add creates a read-after-write dependency, making the Add instruction wait for the Multiply to finish which would lead to pipeline stalls.

$$vacc = vacc + \alpha . b$$

This choice:

- Reduces the total instruction count
- Avoids an intermediate widened product vector
- Lowers temporary register usage

The accumulator is initialized by widening the '`a`' vector, ensuring the subsequent multiply-accumulate step operates on correctly widened operands.

Simulation and Benchmarking Results

Initial testing was performed using QEMU. However, due to significant variability in rdcycle counts caused by JIT compilation effects, I transitioned to gem5 for more stable and interpretable performance measurements. The tests were run on MinorCPU (gem5's in-order processor model) and O3CPU (gem5's Out of Order processor model) to get an estimate of performance on both in-order and out-of-order execution cores. VLEN for both of these models is 256 bits as can be seen in the config.ini file. All the tests here were run with compiler auto-vectorization turned off and -O3 optimization on clang. Performance with compiler auto-vectorization is discussed in the next section.

I evaluated multiple LMUL configurations:

- LMUL = 1
- LMUL = 2
- LMUL = 4

LMUL = 2 provided the best performance. When using LMUL = 4, the widening operation caused widened vectors to occupy groups of eight vector registers, which significantly increased register pressure. This led to vector register spills to the stack and resulted in worse performance than LMUL = 2.

MinorCPU (in-order), L1 cache, No compiler auto-vectorization

LMUL	Scalar Cycles	RVV Cycles
1	101527	16350
2	101527	6961
4	101527	11529

O3CPU (out-of-order), L1 cache, No compiler auto-vectorization

LMUL	Scalar Cycles	RVV Cycles
2	50177	2164
4	50177	3220

Note on LMUL=1 exclusion for O3CPU:
Testing with LMUL=1 on the O3CPU model triggered a known stability issue in the gem5 simulator's instruction queue. The simulator aborted with a panic: Dependency graph … not empty! error indicating an internal failure in the simulator's dependency tracking logic for vector registers rather than a correctness issue in the user code. As this is a toolchain limitation, these specific results were omitted to preserve the integrity of the performance analysis.

Cache selection (L1 vs L2) also plays a part in the performance but those details have been left out for brevity. The difference in performance for the scalar code was around 5000-6000 cycles and around 2000 cycles for RVV code.

Comparison with Compiler Auto-Vectorization

I also evaluated performance with compiler auto-vectorization enabled and disabled. Without auto-vectorization, the RVV Intrinsic implementation achieved approximately a 20x speedup over the scalar reference.

With auto-vectorization enabled, the compiler-generated vectorized version slightly outperformed the hand-written RVV version on the MinorCPU model. On the O3CPU however, the handwritten RVV code with intrinsics performs significantly better.

MinorCPU (in-order), L1 cache, Compiler auto-vectorization enabled, sign extending va into vacc

LMUL	Reference Cycles	Handwritten RVV Cycles
2	6440	6967
4	6524	11472

O3CPU (out-of-order), L1 cache, Compiler auto-vectorization enabled, sign extending va into vacc

LMUL	Reference Cycles	Handwritten RVV Cycles
2	3203	1191
4	3203	3204

A thorough analysis of the generated assembly in both cases reveals the following:

In my code, I change the vector type (`vsetvli`) twice in every single iteration. This is not intended but a consequence of sign extending `va` to get `vacc` with `vsext`. This change in the `VTYPE` CSR forces a pipeline stall on simple in-order cores like MinorCPU to ensure subsequent instructions use the correct configuration. This penalty occurs 2N times where N is the number of times the loop runs. In the code generated by the compiler, it sets the `VTYPE` once to `e16, m2` and uses instructions that naturally handle the widening (`vwmul`, `vwadd`) without needing to explicitly change `VTYPE`. However, on the O3CPU, it sees the `vsetvli` instructions, renames the necessary registers and speculatively executes the math. This effectively hides the latency of the `VTYPE` switches.

To see if I can optimize this, I looked for alternatives to initialize the accumulator with. 'vsext' is treated as a data movement/conversion instruction and the spec mandates that the SEW for this instruction must match the destination. Therefore, to turn a 16-bit vector into a 32-bit vector, the VTYPE must be changed to e32 mode. On the other hand, for a `vwadd` instruction, it is treated as a widening arithmetic instruction and SEW must match the input. Therefore, the `VTYPE` can stay in e16 mode. Therefore, to initialize the accumulator, instead of sign extending '`va`', we can use the `vwadd` instruction to add '`va`' to a scalar 0. This eliminates the VTYPE switching inside the loop.
Implementing this led to a modest ~100 cycle improvement on the MinorCPU but did not surpass the compiler-generated code. It also performed worse on the O3CPU. This shows that VTYPE switching was not the primary bottleneck but the main bottleneck is instruction scheduling and loop unrolling. The ideal implementation would combine the optimal instruction selection of my handwritten code (using `vwmacc` and `vsext` widening which proved superior on the O3CPU) with the optimal scheduling strategies of the compiler (unrolling and interleaving). For the purpose of this challenge, the handwritten solution prioritizes readability and correctness over the code complexity required for manual software pipelining.
The generated assembly for the function written using RVV intrinsics is present in the GitHub repo in `asm/rvv-generated-assembly.s`. The generated assembly for the scalar reference function with and without compiler auto-vectorization is present in `asm/scalar-no-vectorize-generated-assembly.s` and `asm/scalar-vectorized-generated-assembly.s`.

Theoretical Speedup Considerations

In the scalar version, each loop iteration processes one element and performs two loads, one multiply, one add, saturation logic and one store. In the RVV version, each vector iteration processes ‘vl’ elements using a small number of vector instructions. For SEW=16 and VLEN=256 bits, this corresponds to up to 16 elements per iteration (for LMUL=1) and 32 elements (for LMUL=2). Ignoring loop overhead and memory effects, this suggests an ideal speedup proportional to the number of elements processed per vector instruction. The measured results are consistent with this expectation, although practical speedups are lower due to memory bandwidth and behavior, control overhead and register pressure.

$$Effective\ elements\ per\ iteration\ =\ VLEN/SEW\ *\ LMUL$$

Expected speedup for LMUL=2: 32x
Practical speedup for LMUL=2: 20x-25x

Building

For assembly inspection, I compiled the program using clang with RVV support. My local GCC toolchain did not provide up-to-date RVV intrinsic headers, so Clang was used for code generation. Linking a full binary with clang requires a RISC-V sysroot (compiler-rt and C runtime) which was not available locally. Therefore, I used gcc to link the object files, clang for compiling the code, generating object files and assembly.

Example invocation:

```
clang -O3 -c --target=riscv64-elf -march=rv64gcv -mabi=lp64d -mno-relax
-o q15_axpy_challenge.o q15_axpy_challenge.c -fno-vectorize

riscv64-elf-gcc -march=rv64gcv -mabi=lp64d -static -o
q15_axpy_challenge q15_axpy_challenge.o
```

Older draft headers use a different `vnclip` signature (no explicit `vxrm` argument). These are excluded via preprocessor guards and fall back to the scalar implementation.