AudioMark Coding Challenge:
RISC-V Vector Implementation of Q15 AXPY

Implementation Overview

This report evaluates a Q15 fixed-point AXPY operation ($y = a + \alpha \cdot b$) with saturating arithmetic, representative of audio signal-processing workloads. A scalar baseline is compared against an implementation using the RISC-V Vector Extension (RVV) v1.0.

Scalar Baseline

```
void q15_axpy_ref(const int16_t *a, const int16_t *b,
                  int16_t *y, int n, int16_t alpha) {
    for (int i = 0; i < n; ++i) {
        int32_t acc = (int32_t)a[i] +
                      (int32_t)alpha * (int32_t)b[i];
        y[i] = sat_q15_scalar(acc);
    }
}
```

RVV Implementation

Strategy: LMUL=4, widening accumulation to prevent overflow, fused multiply-accumulate, and saturating narrowing back to Q15.

```
vl = __riscv_vsetvl_e16m4(n);
vint16m4_t va = __riscv_vle16_v_i16m4(a, vl);
vint16m4_t vb = __riscv_vle16_v_i16m4(b, vl);
vint32m8_t v_acc = __riscv_vwcvt_x_x_v_i32m8(va, vl);
v_acc = __riscv_vwmacc_vx_i32m8(v_acc, alpha, vb, vl);
vint16m4_t v_res = __riscv_vnclip_wx_i16m4(
    v_acc, 0, __RISCV_VXRM_RNU, vl);
__riscv_vse16_v_i16m4(y, v_res, vl);
```

Theoretical Performance Considerations

Instruction Count Analysis

For VLEN = 128 bits and LMUL = 4:

$$VL = \frac{128}{16} \times 4 = 32 \quad \text{elements per iteration}$$

Scalar: Approximately 13 instructions per element, including memory accesses, arithmetic, saturation, and loop control.

Vector: Approximately 12 instructions per vector iteration, amortized across 32 elements.

Effective vector instruction density:

$$I_{v,\text{eff}} = \frac{12}{32} \approx 0.375 \quad \text{instructions per element}$$

This comparison highlights the substantial reduction in instruction density enabled by vectorization.

Memory Traffic

Each element requires two 16-bit loads and one 16-bit store, resulting in 6 bytes of memory traffic per element. Scalar and vector implementations therefore exhibit identical memory traffic, though vector execution can improve memory-level parallelism for streaming access patterns.

Experimental Results

QEMU Emulation

Measurements were obtained using QEMU user-mode emulation. Because QEMU does not model vector parallelism, pipelines, or realistic cache behavior, the results below are intended solely to verify functional correctness.

Implementation	Speedup	Correctness
Scalar Reference	1.00×	Baseline
RVV Vectorized	∼1.2×	**PASS**

Table 1: QEMU results demonstrate functional correctness only; performance is not representative of real RVV hardware.

Implementation Challenges

During the implementation of the RVV-based Q15 AXPY kernel, several practical challenges were encountered, primarily related to tooling and simulation rather than algorithmic complexity.

Initial attempts to build and run the vectorized implementation using `g++` resulted in inconsistent behavior for RISC-V Vector intrinsics. After investigating these issues, the toolchain was switched to `clang`, which provided more reliable support for the required RVV features and enabled correct execution of the kernel.

Additionally, early performance measurements using the `rdcycle` instruction were limited when running under the Spike simulator, as accurate cycle counting requires a bare-metal environment. To proceed with functional validation and basic measurement, the workflow was migrated to QEMU user-mode emulation. While QEMU does not provide cycle-accurate performance data for vector execution, it allowed verification of correctness and completion of the implementation.

Working through these issues required iterating on the toolchain and execution environment and provided practical insight into the current state of RISC-V Vector software support.

Conclusion

This work demonstrates a correct RVV implementation of a Q15 AXPY kernel using widening arithmetic, fused multiply-accumulate operations, and saturating narrowing. Static analysis shows a substantial reduction in instruction density compared to a scalar baseline.

On real hardware, overall performance will be influenced primarily by memory bandwidth, vector setup overheads, and vector load/store throughput rather than raw arithmetic capability. Quantitative performance characterization therefore requires execution on physical RVV-capable processors.