Vector-length agnostic varint decoding on RISC-V

Slide-and-compress decoding for the RISC-V Vector Extension

Marco Vogel Lena Oden

RISC-V for HPC @ ISC

FernUniversität in Hagen

Background

Variable-length integer encoding

Each integer is stored in 1 to 5 bytes. Every byte carries 7 payload bits and a continuation bit in the MSB, where MSB = 1 means another byte follows.

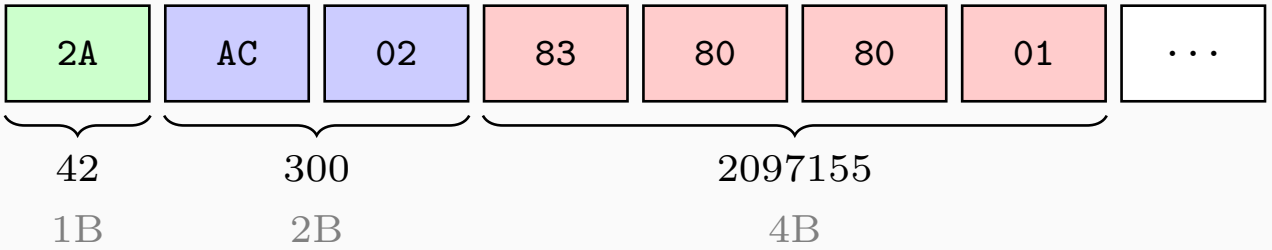
Integer **300** in binary: 10 010 1100

Encoded: `[0xAC, 0x02]`, 2 bytes instead of 4

Default wire format in Protocol Buffers, WebAssembly and Lucene posting lists, so its decoding throughput matters.

Why decoding is hard to vectorize

Encoded byte stream:

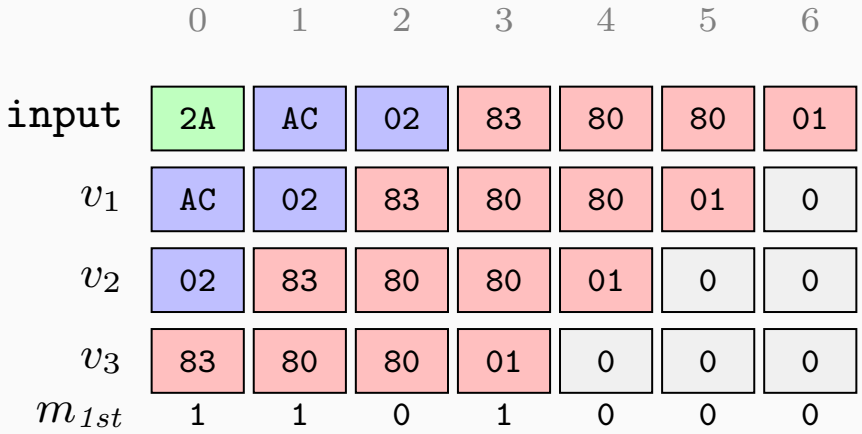


Challenge: Integer boundaries are unknown until continuation bits are inspected sequentially, inhibiting parallel decoding.

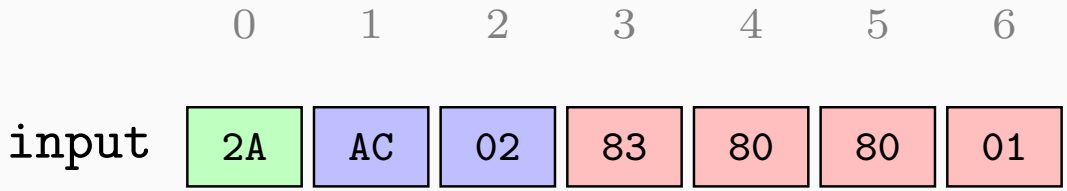
Slide-and-compress

Data transformation

The idea is to build shifted views of the input, then compress all of them with the same first-byte mask.

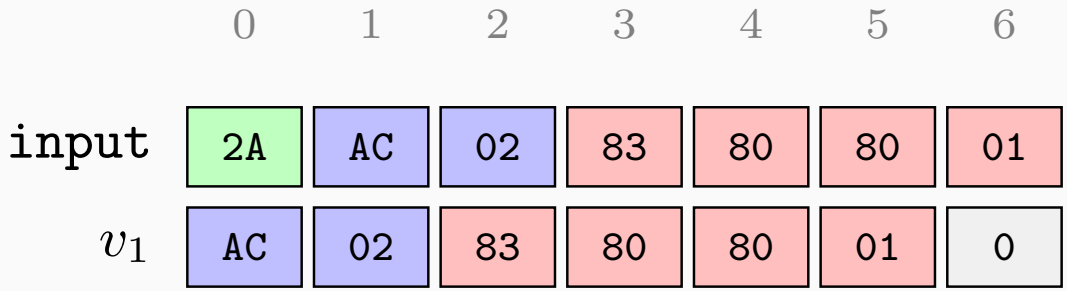


Slide, then compress



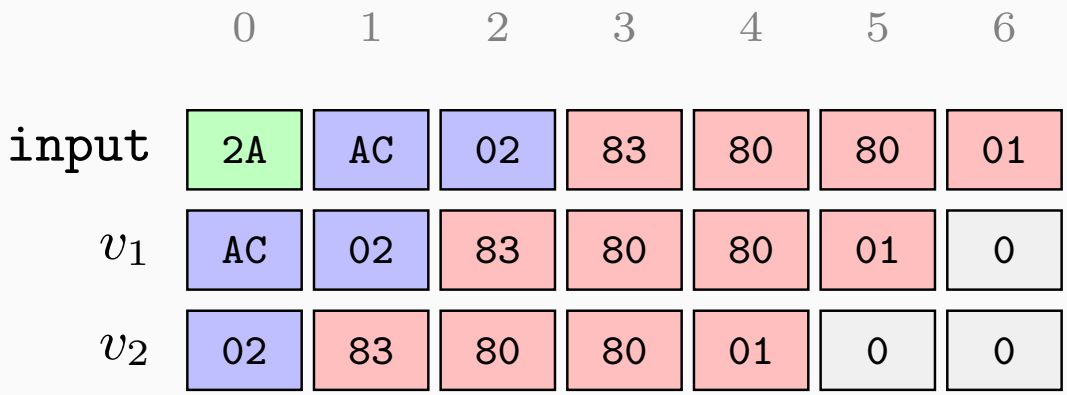
(a) load the input window

Slide, then compress



(a) `v1 = vslide1down(input)`

Slide, then compress



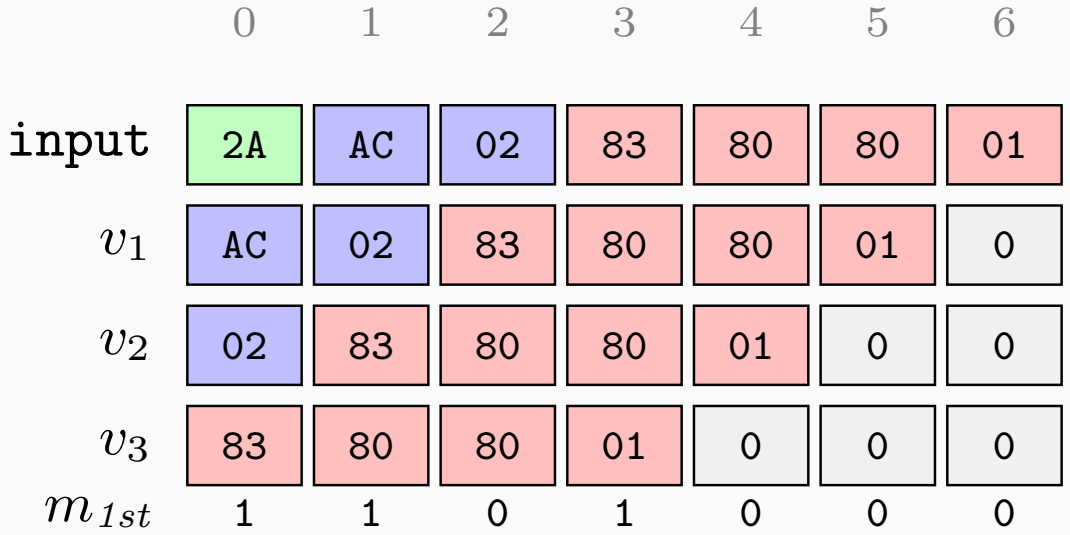
(a) `v2 = vslide1down(v1)`

Slide, then compress

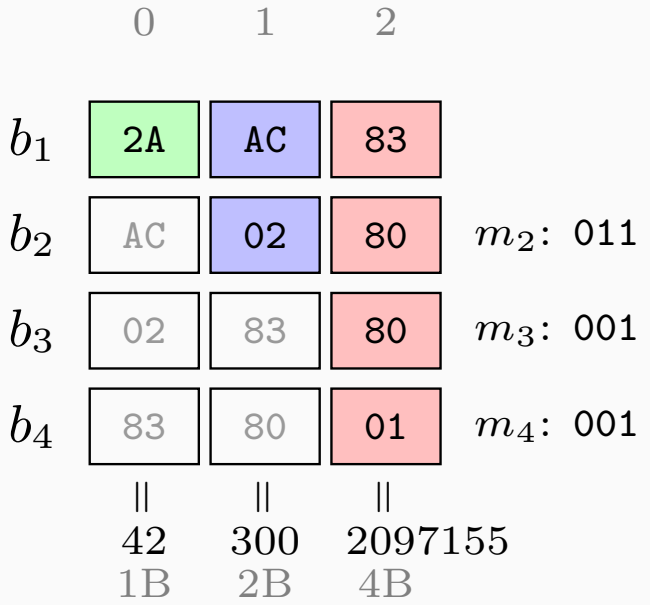
	0	1	2	3	4	5	6
`input`	2A	AC	02	83	80	80	01
v_1	AC	02	83	80	80	01	0
v_2	02	83	80	80	01	0	0
v_3	83	80	80	01	0	0	0
m_{1st}	1	1	0	1	0	0	0

(a) `v3`, then `m_1st` marks first bytes

Slide, then compress



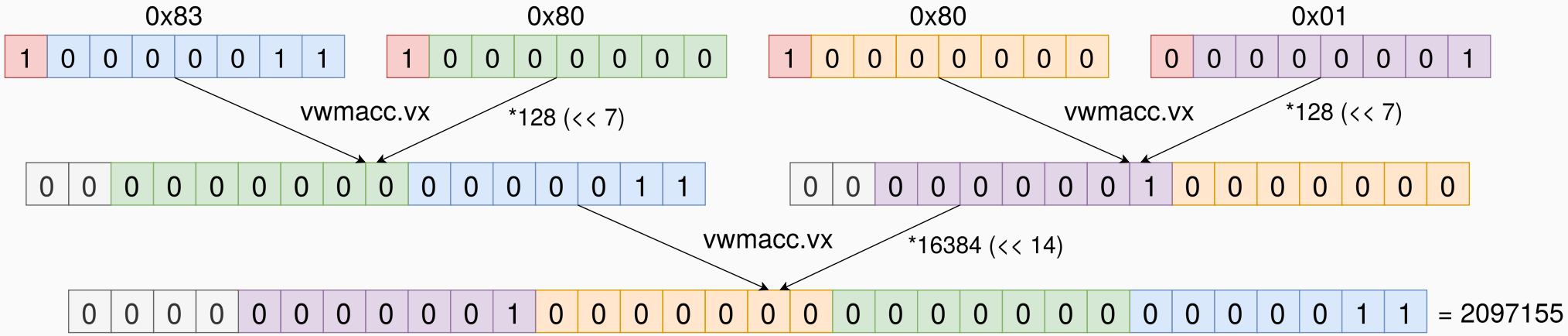
(a) `v3`, then `m_1st` marks first bytes



(b) `vcompress` with `m_1st` aligns each varint

Lane `j` of `b1` … `b4` now holds each byte of varint `j`. A 5th view is built only when a 5-byte varint appears.

Reconstruct with widening multiply-accumulate



$$r_{12} = b_1 + b_2 \cdot 128, \quad r_{34} = b_3 + b_4 \cdot 128,$$

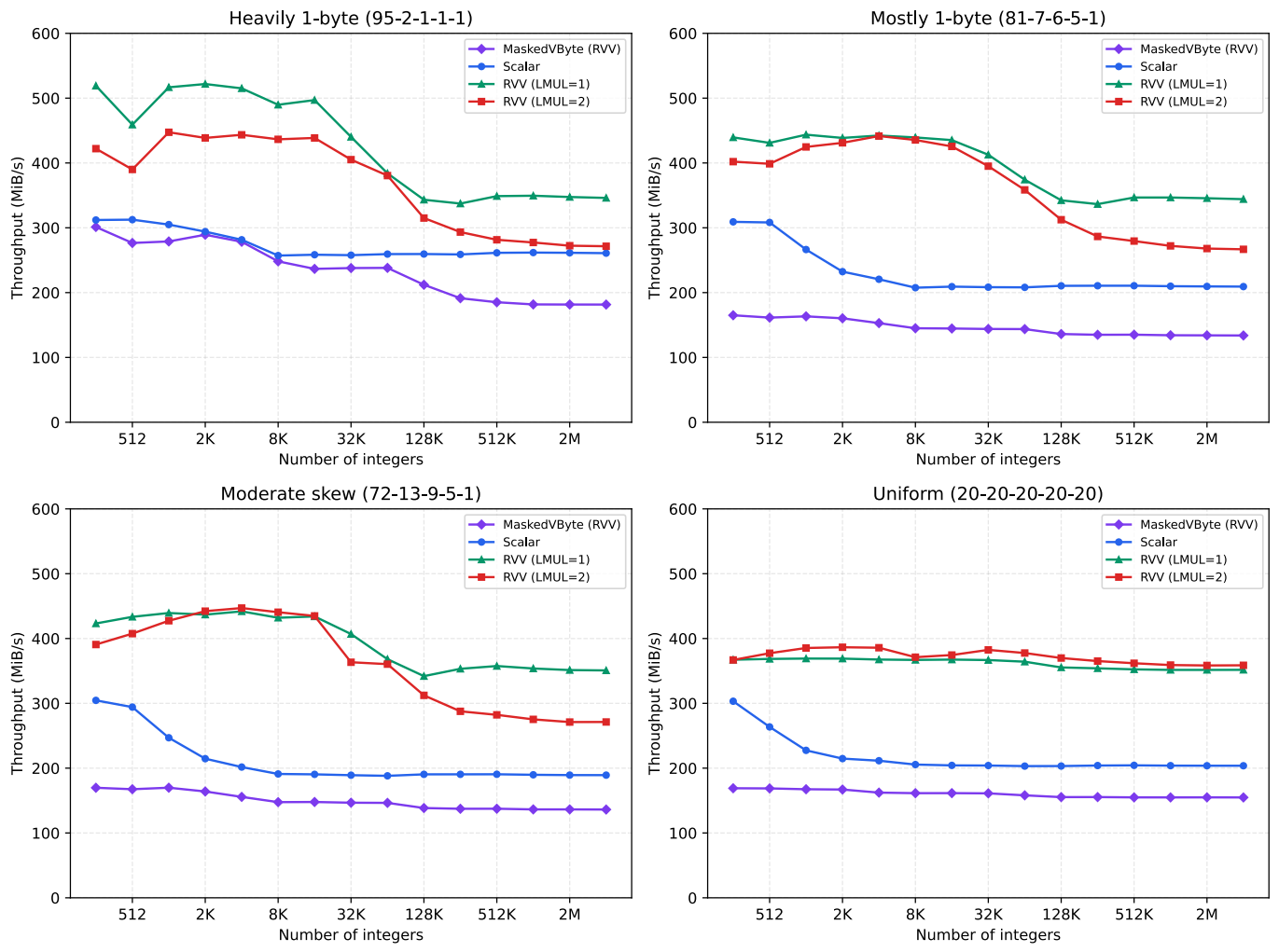
$$r_{1234} = r_{12} + r_{34} \cdot 16384 = 2,097,155$$

Results

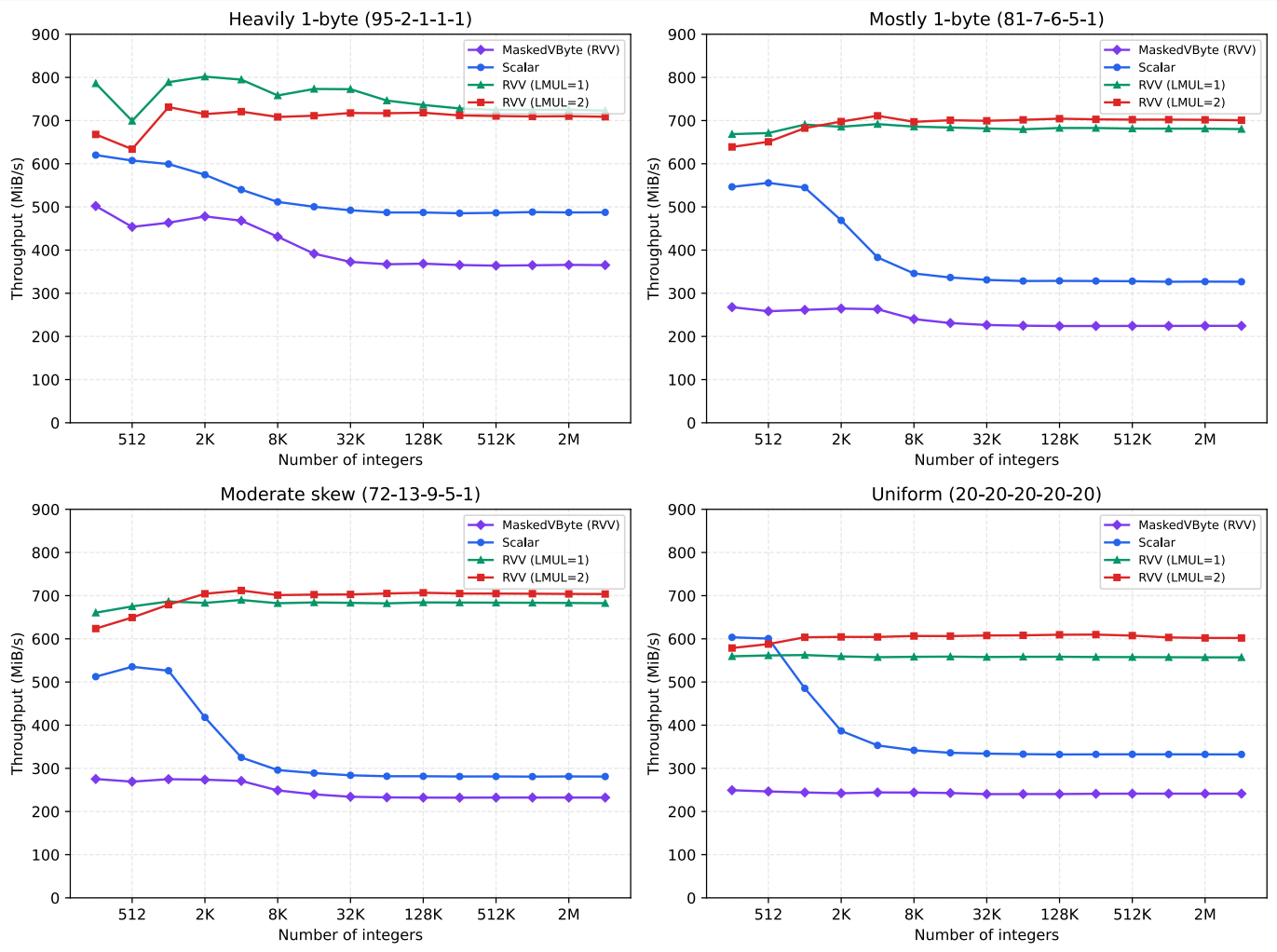
Two evaluation platforms

	K1 (X60)	K3 (X100)
Pipeline	in-order	out-of-order
ISA profile	RVA22	RVA23
VLEN	256	256
Clock	1.6 GHz	2.4 GHz

K1 (in-order)


Heavily 1-byte (95-2-1-1-1)
Mostly 1-byte (81-7-6-5-1)
Moderate skew (72-13-9-5-1)
Uniform (20-20-20-20-20)
MaskedVByte (RVV)
Scalar
RVV (LMUL=1)
RVV (LMUL=2)
Throughput (MiB/s)
Number of integers
0
100
200
300
400
500
600
512
2K
8K
32K
128K
512K
2M

K3 (out-of-order)



Porting MaskedVByte to RVV

We also tried this. The port stays below scalar on both platforms, at 0.65–0.83× of scalar throughput.

- `vrgather` emulates `pshufb` at about 4 cycles per op, and its latency grows with LMUL
- The fixed 16-byte window leaves half of a VLEN=256 datapath idle
- Two 4096-entry lookup tables pressure the L1D cache
- The design is fundamentally fixed-width, so it does not fit VLA
- The out-of-order K3 only partly hides the cost

A direct x86-SIMD port can be slower than scalar on in-order RVV, which is what motivates the VLA design.

Conclusion

Summary

1. A vectorized LEB128 decoder for the RISC-V Vector Extension (RVV).
2. Fully vector-length agnostic: one binary runs at any VLEN, with no lookup tables.
3. Up to 1.9× (in-order K1) and 2.5× (out-of-order K3) over a scalar baseline.
4. A direct MaskedVByte port that shows fixed-width SIMD does not carry over well.

Code and benchmarks are available. Thank you.

`github.com/vogma/varint_rvv`