Inner-Product Matrix Extensions

Andrew Waterman, SiFive, Inc.
RISC-V Integrated Matrix Extensions Task Group
Monday, September 23, 2024

If you’ve got dedicated accumulators, use outer products

- Keeping accumulators in place improves efficiency
    - Usually wider than multiplicands
    - Must be written back to memory
- Fewer structural hazards => easier to supply multiplicands
- Decoupling VRF size from accumulator RF simplifies scaling

This is an attractive design point.
This is AME.

Don’t use outer products without dedicated accumulators

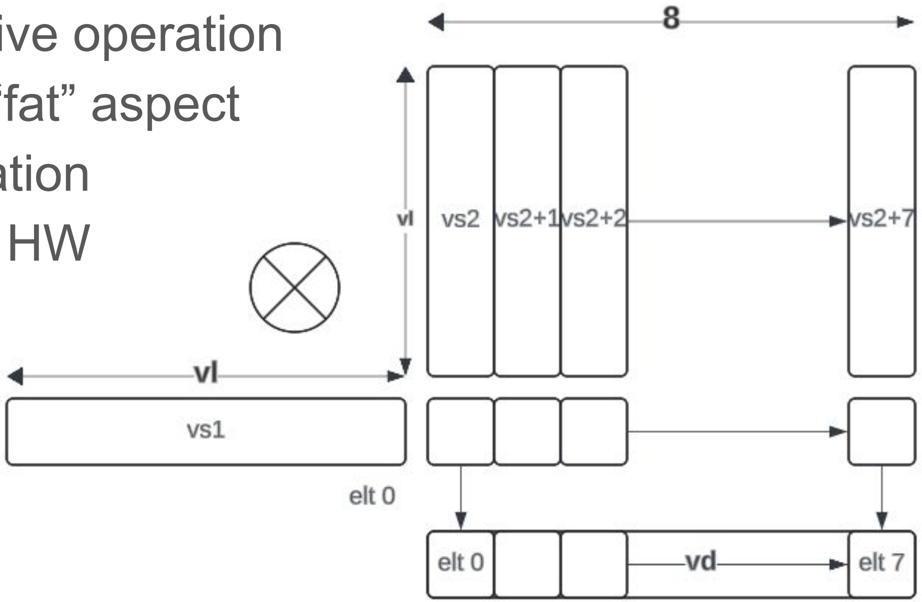
- All the upsides become downsides
- Wide accumulators more costly to xfer than narrow mpcands
- Mismatched VRF traffic pattern
  - Far more VRF writes than reads
  - Doubly problematic for renamed designs
- Poor scaling with increasing VLEN
  - Inherent misshapenness between 1D/2D constructs
  - Unnatural memory layouts for aEW > mEW

Outer-product implementation challenge is significant

- Typical RVV implementation has ~4x VRF read vs. write BW
    - Outer-product scheme has wrong shape
- Writes are even more expensive in OOO cores:
    - VPRF entries migrate => no accumulator locality savings
    - VPRF entries are allocated => performance-limiting resource
    - Rename BW is a critical resource
- µArch hacks possible, but limited in scope
    - Speculative copies of accumulators quickly grow in cost
    - If you can afford ample state, better to expose it architecturally (for transposes, etc.)

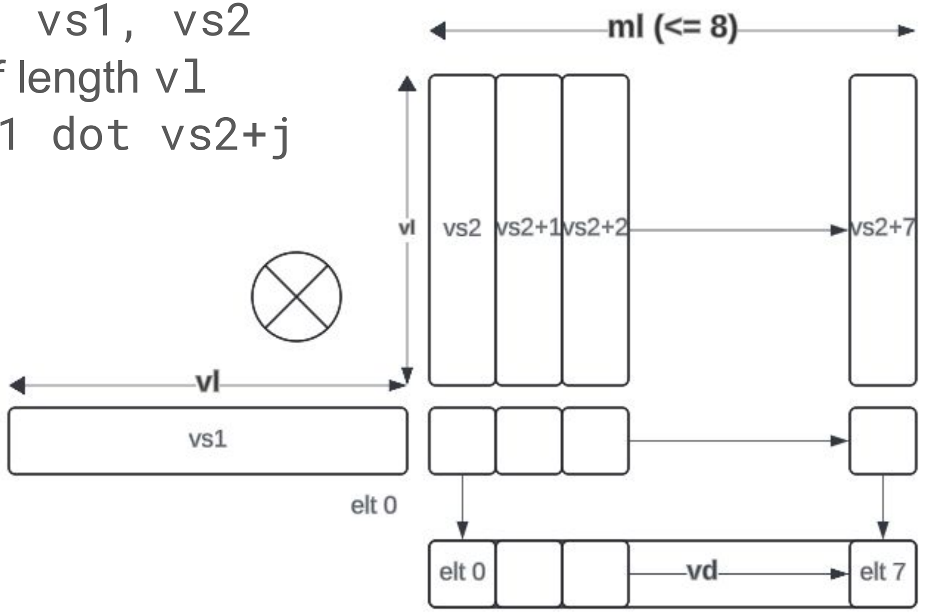
“Fat” inner products are a better choice for IME

- Straightforward scaling with vector length
- Standard memory layout for all matrices
- Better shape for conventional datapath design
- Fewer registers renamed per effective operation
- Constant-factor operand reuse via “fat” aspect
- Energy savings from bulk normalization
- Low-end impls can reuse vfredsum HW
- Speeds up GeMV, too



Xsfvmm32a32f, an FP32 “fat” inner-product extension

- New field in `vtype`[19:16]: `ml`, “matrix length”
- New instruction: `sf.vsetml rd, rs1`
  - `ml := rd := min(8,rs1)`
- New instruction: `sf.vfmmacc.vv vd, vs1, vs2`
  - Perform up to eight dot products of length `vl`
  - `∀j ∈ [0,ml): vd[j] += vs1 dot vs2+j`
  - `vs1` is one vreg
  - `vs2` is an eight-vreg group
  - `vd` is enough vregs to hold 8 elts

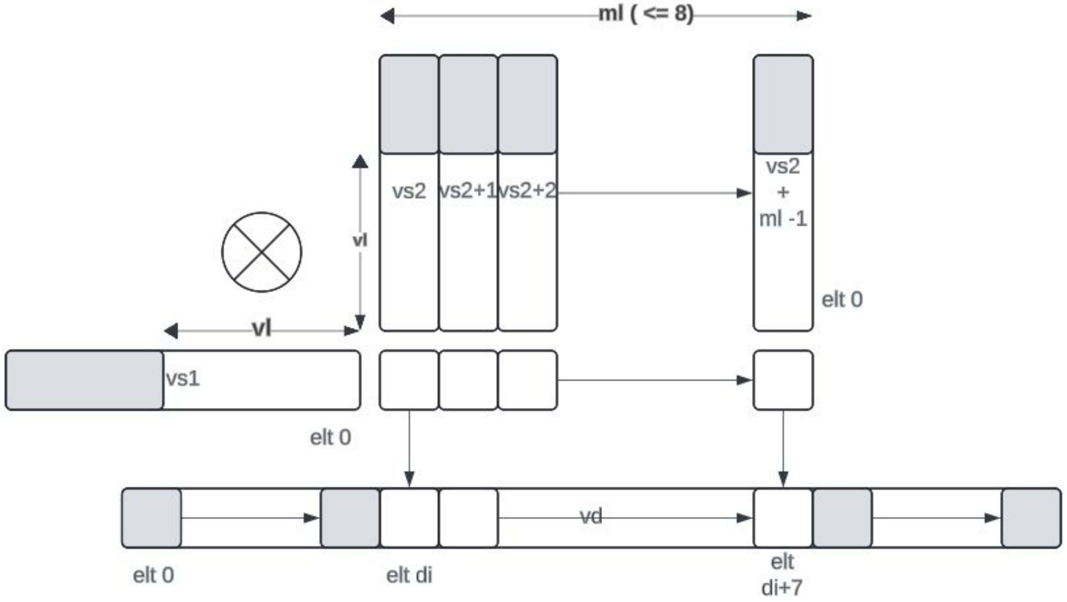


Efficient use of long vector registers

- `vs1`, `vs2` operands always fully usable, regardless of VLEN
- But with only 8 result elts, how to efficiently use all of `vd`?

Efficient use of long vector registers

- `sf.vfmmacc.vv vd, vs1, vs2, di`
- `di` is a scaled-by-eight immediate (0,8,...,56) offset into `vd`
- Can utilize all of vd for VLEN up to 2048



A quantitative comparison: SEW=AEW=32, VLEN=256

- Strawman outer-product scheme performs 8x1x8 ops; use 8x1x24 tiles
  - Inner loop performs 4 8-elt unit-stride loads, performs 3 8x1x8 matmuls
- Xsfvmm32a32f inner-product scheme performs 1x8x8 ops; use 23x8x8 tiles
  - Inner loop performs 31 8-elt unit-stride loads, performs 23 1x8x8 matmuls

	Inner product	Outer product
Loads	248	32
VRF reads	1840	240
VRF writes	432	224
MACCs	1472	192
Loads/MACC	0.17	0.17
VRF reads/MACC	1.25	1.25
VRF writes/MACC	0.29	1.17

Mixed-precision support

- Scheme extends naturally to mixed precision
- Natural memory layout for mixed precision, too
- Same shape as before (1xVLx8)
- Xsfvmm32a8i adds sf.vqmmacc[s|u][su].vv instructions (int8->int32)
- Xsfvmm32a16f adds sf.vfwmmacc.[bf.]vv instructions (FP16->FP32, BF16->FP32)

A concrete code example: SEW=8, AEW=32, VLEN=256

```
loop:
vsetvli rvl,ravl,e8,m1,ta,ma

# Load B tile (8 rows)
vle8.v v0,(rb0); add rb,rb0,rldb
vle8.v v1,(rb);  add rb,rb,rldb
…
vle8.v v7,(rb)

# Bookkeeping
add ra,ra0,rk

# Continue on next column
```

```
# Load A row; multiply (23 times)
vle8.v v31,(ra0); add ra,ra0,rlda
sf.vqmmacc.vv v8, v31, v0

vle8.v v31,(ra); add ra,ra,rlda
sf.vqmmacc.vv v9, v31, v0
…
vle8.v v31,(ra); add ra,ra,rlda
sf.vqmmacc.vv v30, v31, v0

# Bookkeeping
add rb0,rb0,rvl; add ra0,ra0,rvl
add rk,rk,rvl; sub ravl,ravl,rvl
bnez ravl, loop
```

Questions?