



Accelerator Architectures for Machine Learning (AAML)

Lecture 4: RISC-V VPU

Tsung Tai Yeh

Department of Computer Science
National Yang-Ming Chiao Tung University



Acknowledgements and Disclaimer

- Slides was developed in the reference with
Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019
tutorial
Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin
Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC
Berkeley, 2020
CS231n Convolutional Neural Networks for Visual Recognition,
Stanford University, 2020



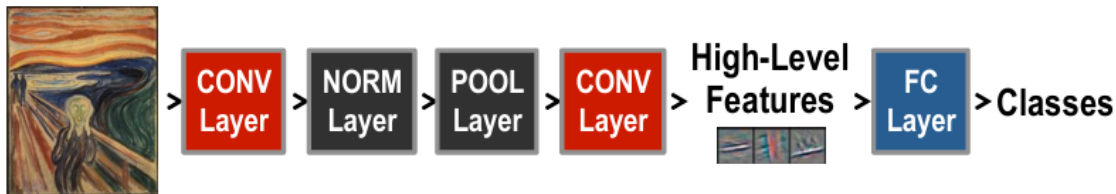
Outline

- HW/SW Domain-Specialization
 - AI kernel Codes
 - Mixed-Precision
 - Customized RISC-V ISAs
- Vector Processors



Accelerating AI Through HW/SW Domain-Specialization

- **Efficient SW kernels on specialized RISC-V cores/systems should:**
 - Algorithm optimization to increase operational intensity (#Ops/byte)
 - Regularize memory accesses whenever possible
 - Optimize data layout to minimize data movement costs -> **moving data is energy inefficient**
 - Exploit the ISA/uarchi capabilities of the underlying HW
 - Maximize data reuse at RF level
 - **Efficiently use domain-specialized instructions**





Naïve 6-Loop Implementation of CONV Layers

```
for (m=0; m<M; m++) {  
  for (x=0; x<F; x++) {  
    for (y=0; y<E; y++) {  
      O[n][m][x][y] = B[m];  
  
      for (i=0; i<R; i++) {  
        for (j=0; j<S; j++) {  
          for (k=0; k<C; k++) {  
            O[n][m][x][y] += I[n][k][Ux+i][Uy+j]  
                           × W[m][k][i][j];  
          }  
        }  
      }  
      O[n][m][x][y] =  
      Activation(O[n][m][x][y]);  
    }  
  }  
}
```

convolve
a window
and apply
activation

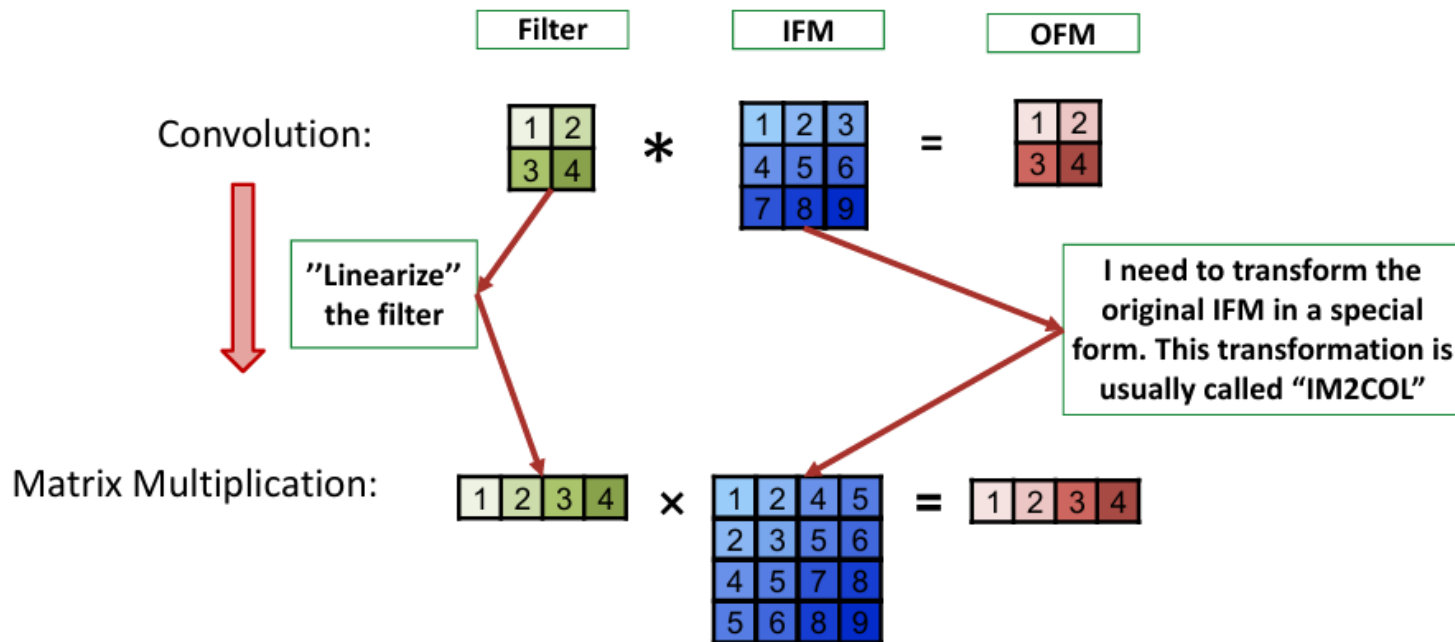
} for each
output fmap
value

- Not very efficient from a computing perspective
 - Many 'memory' accesses for few actual MAC operations
- Matrix-Vector Multiplications (MVMs) or Matrix-Matrix Multiplications (MMMs) are much more efficient for processing
 - Denser and more regular memory access patterns
 - Regular Compute structures
- Goal: Transform Convolutions into Matrix Multiplications



Convolution Layer as a Matrix Multiplication

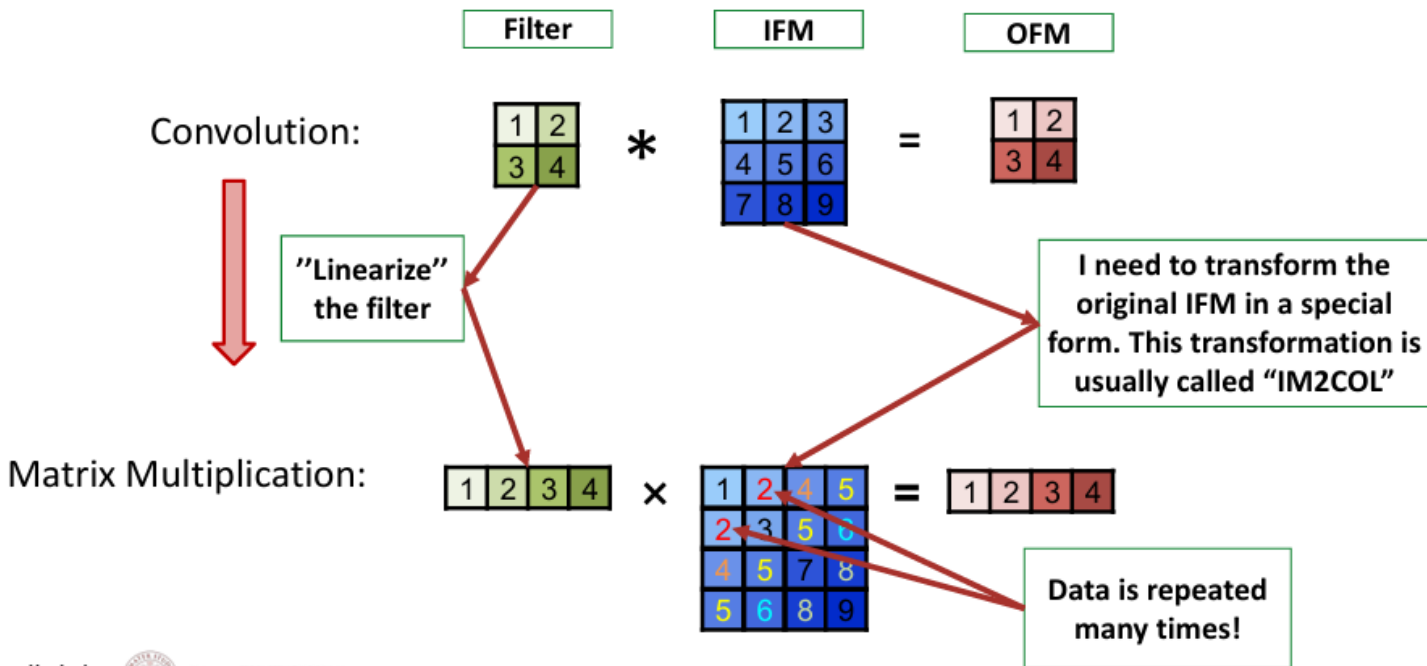
- Convert to Matrix Multiplications by using Toeplitz Matrices





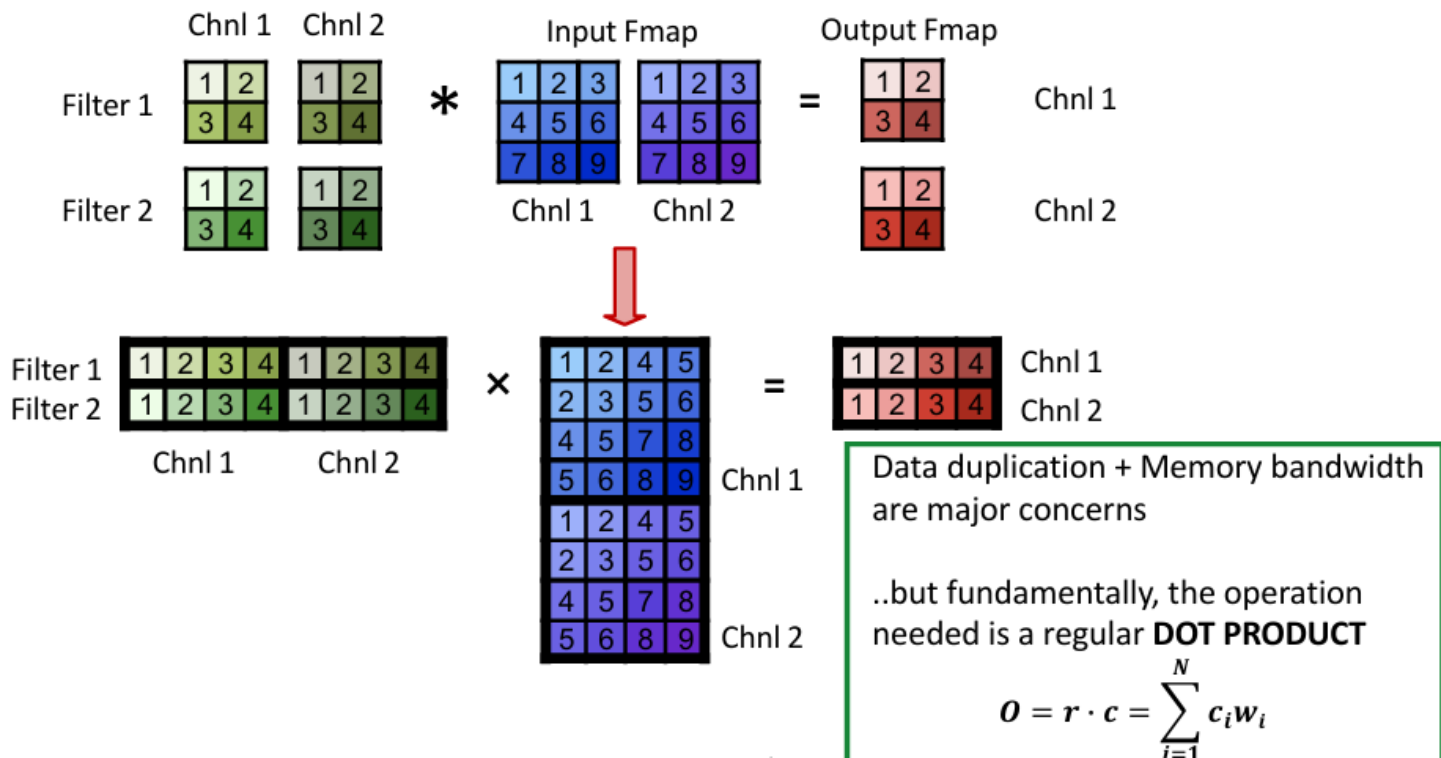
Convolution Layer as a Matrix Multiplication

- Nothing comes for free -> a lot of data duplication (mem overhead)





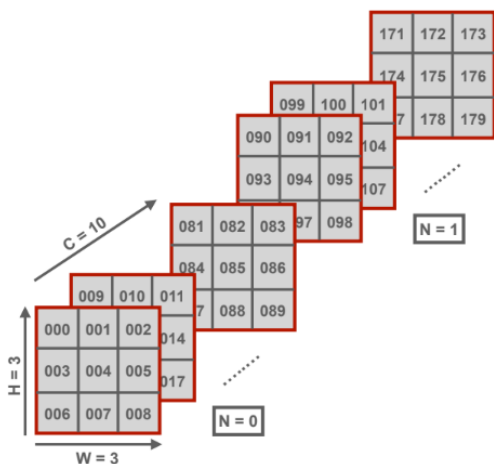
CONV Layer with Multiple Channels and Filters





Data Memory Layouts to store I(O)FMs and CONVs

- **N-C-H-W** -> stores the spatial dimensions at contiguous mem addr.
 - Less efficient for conv. as more 'memory' operations and data manipulation
 - Required for the IM2COL phase (especially for low-bitwidth arith and exploiting Packed SIMD)



NCHW: (used by default in Caffe)

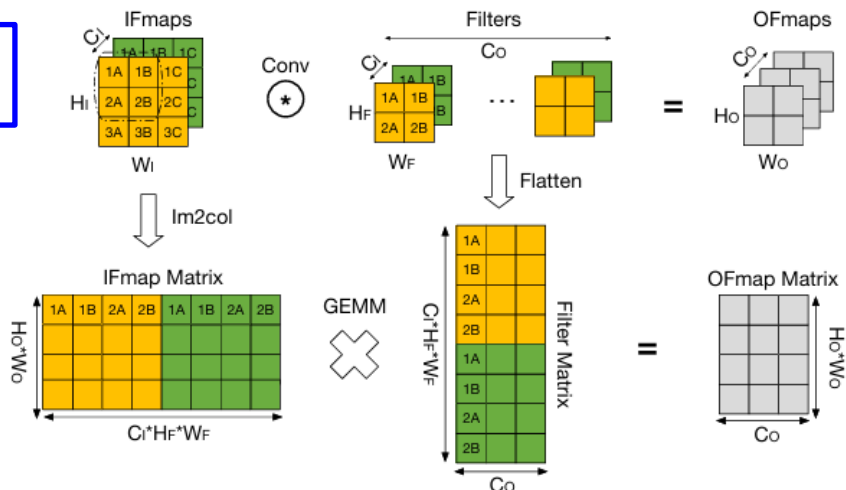
000	001	002	003	...	009	010	...	089	090	091
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

NHWC: (used by default in TensorFlow)

000	009	018	...	081	001	010	...	089	090	099
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

CHWN:

000	090	001	091	...	008	098	009	099	...	177
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

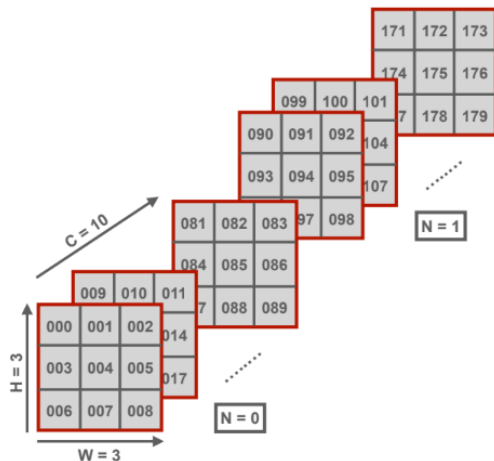


IM2COL converts CONV to GEMM operation



Data Memory Layouts to store I(O)FMs and CONVs

- **N-H-W-C->** stores channels at contiguous addr (preferred layout in edge devices)
 - More efficient data movements to build IM2COL buffers (especially for low bitwidth precisions exploiting packed-SIMD computation)



NCHW: (used by default in Caffe)

000	001	002	003	...	009	010	...	089	090	091	...	177	178	179
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

NHWC: (used by default in TensorFlow)

000	009	018	...	081	001	010	...	089	090	099	...	163	170	179
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

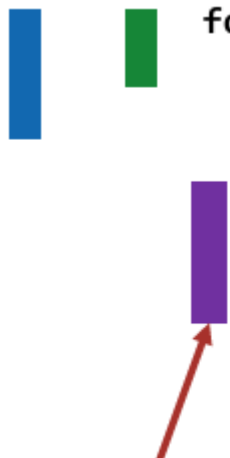
CHWN:

000	090	001	091	...	008	098	009	099	...	177	088	178	089	179
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights



```
for i in range(0, E):
    for j in range(0, F):
        for m in range(0, M):
            psum = 0
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        psum += w[m,ui,uj,n] * x[i+ui,j+uj,n]
            y[i,j,m] = act(psum)
```

ignore bias

Psum stationary in RF



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):  
    for j in range(0, F):  
        for m in range(0, M):  
            psum = 0                                # ignore bias  
            for ui in range(0, S):  
                for uj in range(0, R):  
                    for n in range(0, C):  
                        psum += w[m,ui,uj,n] * x[i+ui,j+uj,n]  
            y[i,j,m] = act(psum)
```

We can apply im2col only on activations within the currently considered receptive field (along W and H dimensions) → **Partial im2col** → Reduced memory footprint!

1. Make x access more regular: reorder in **im2col** buffer



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):
    for j in range(0, F):
        for m in range(0, M):
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]
            psum = 0
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        psum += w[m,ui,uj,n] * im2col[ui,uj,n]
            y[i,j,m] = act(psum) # ignore bias
```

1. Make x access more regular: reorder in **im2col** buffer



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):
    for j in range(0, F):
        for m in range(0, M):
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]
            psum = 0 # ignore bias
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        psum += w[m,ui,uj,n] * im2col[ui,uj,n]
            y[i,j,m] = act(psum)
```

We then use the built **im2col** buffer for computation

1. Make **x** access more regular: reorder in **im2col** buffer



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):  
    for j in range(0, F):  
        for m in range(0, M):  
            for ui in range(0, S):  
                for uj in range(0, R):  
                    for n in range(0, C):  
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]  
            psum = 0  
            for ui in range(0, S):  
                for uj in range(0, R):  
                    for n in range(0, C):  
                        psum += w[m,ui,uj,n] * im2col[ui,uj,n]  
            y[i,j,m] = act(psum)
```

Same pattern (linear) to
access **w** and **im2col** →
Linear memory access!

ignore bias

We then use the built **im2col** buffer
for computation

1. Make **x** access more regular: reorder in **im2col** buffer



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):
    for j in range(0, F):
        for m in range(0, M):
            for ui in range(0, S):
                for uj in range(0, R):
                    for n in range(0, C):
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]
                    psum = 0
                    for idx in range(0, S*R*C):
                        psum += w[m,idx] * im2col[idx]
                    y[i,j,m] = act(psum)
```

ignore bias

2. *im2col* , *w* access are linear (and identical)



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):  
    for j in range(0, F):  
        for m in range(0, M):  
            for ui in range(0, S):  
                for uj in range(0, R):  
                    for n in range(0, C):  
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]  
                    psum = 0 # ignore bias  
                    for idx in range(0, S*R*C):  
                        psum += w[m,idx] * im2col[idx]  
                    y[i,j,m] = act(psum)
```

3. Highlighted kernel is a **Matrix-Vector Multiplication!**



AI Kernels for RISC-V Devices

- Standard output stationary loop nest, HWC layout for activations, MHWC for weights

```
for i in range(0, E):  
    for j in range(0, F):  
        for m in range(0, M):  
            for ui in range(0, S):  
                for uj in range(0, R):  
                    for n in range(0, C):  
                        im2col[ui,uj,n] = x[i+ui,j+uj,n]  
                    psum = 0  
                    for idx in range(0, S*R*C):  
                        psum += w[m,idx] * im2col[idx]  
                    y[i,j,m] = act(psum)
```

ignore bias

Optimizations:

- Reuse same im2col over multiple conv filters
- Build more im2col buffers in //
→ **LOOP Unrolling + Data Reuse in RF**
→ **Until you saturate RF regs**

3. Highlighted kernel is a **Matrix-Vector Multiplication!**



PULP-NN 2 x 2 kernel with customized RISC-V ISAs

RV32IMC

N

```
addi a0,a0,1
addi t1,t1,1
addi t3,t3,1
addi t4,t4,1
lbu a7,-1(a0)
lbu a6,-1(t4)
lbu a5,-1(t3)
lbu t5,-1(t1)
mul s1,a7,a6
mul a7,a7,a5
add s0,s0,s1
mul a6,a6,t5
add t0,t0,a7
mul a5,a5,t5
add t2,t2,a6
add t6,t6,a5
bne s5,a0,1c000bc
```

RV32IMCXpulp

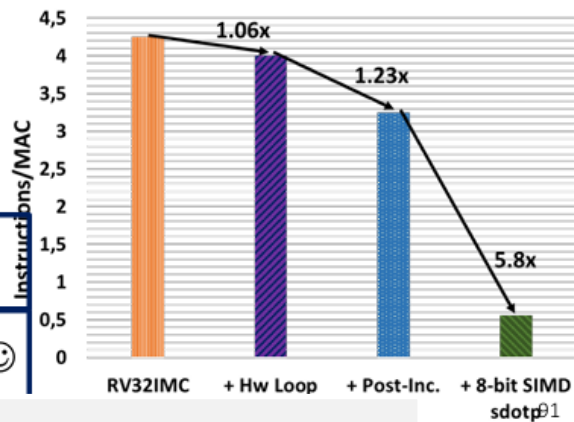
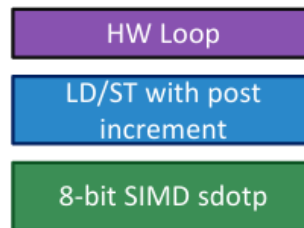
N/4

```
lp.setup
p.lw w1, 4(a0!)
p.lw w2, 4(a1!)
p.lw x1, 4(a2!)
p.lw x2, 4(a3!)
pv.sdotsp.b s1, w1, x1
pv.sdotsp.b s2, w1, x2
pv.sdotsp.b s3, w2, x1
pv.sdotsp.b s4, w2, x2
End: ...
```

8x less instructions
than RV32IMC

Compute eff: $4\text{ops}/8\text{insn} = 0.5$ 😊

8-bit Convolution – 2x2 MatMul kernel

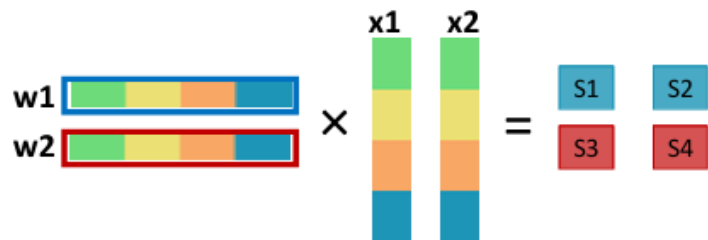




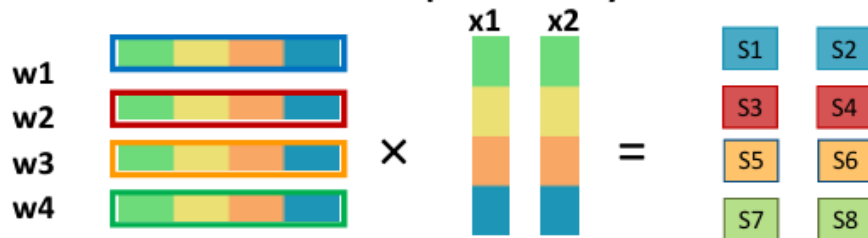
Can we Exploit More Data Reuse at RF Level?

8-bit Convolution

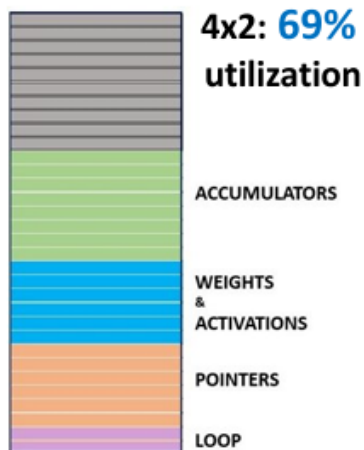
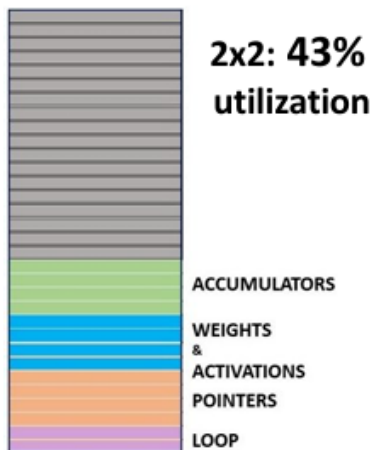
Initial Matrix Multiplication Layout (from CMSIS-NN) : 2x2



PULP-NN Matrix Multiplication Layout: 4x2



RegisterFile
of the RISCY core:
32 general purpose
registers

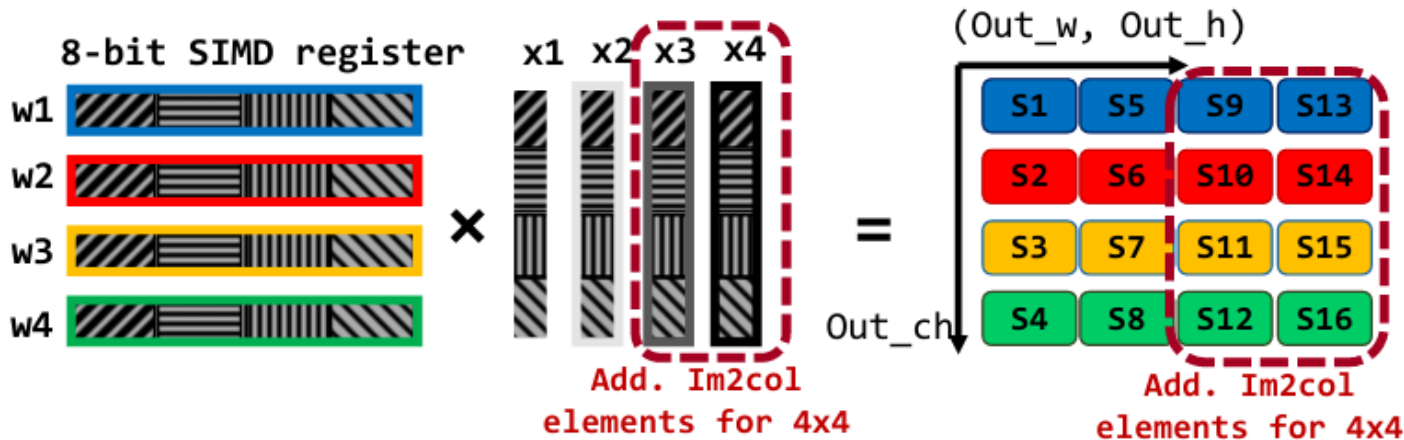


More Data Reuse
&
Higher utilization of the RF



Can We Improve Further?

MatMul Innermost loop structure (example on 8-bit)



Register File Resources required	4x2 kernel
	SIMD
Loop Setup	2
Addresses for MEM access	6
MAC operands	6
MAC accumulators	8

4x4 kernel
SIMD
2
8
8
16



Optimized Back-End AI Kernels

Target **int8** execution of CONV, FC, ... primitives

- 1) maximize **data reuse in register file**
- 2) improve **kernel regularity**
- 3) exploit **parallelism**

HWC format

lp.setup

p.lw w0, 4(W0!)

p.lw w1, 4(W1!)

p.lw w2, 4(W2!)

p.lw w3, 4(W3!)

p.lw x1, 4(X0!)

p.lw x2, 4(X1!)

pv.sdotsp.b acc1, w0, x0

pv.sdotsp.b acc2, w0, x1

pv.sdotsp.b acc3, w1, x0

pv.sdotsp.b acc4, w1, x1

pv.sdotsp.b acc5, w2, x0

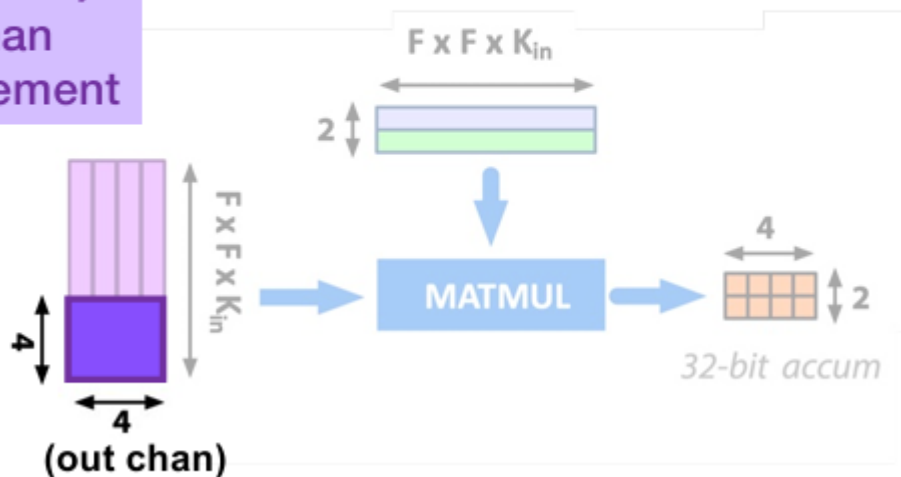
pv.sdotsp.b acc6, w2, x1

pv.sdotsp.b acc7, w3, x0

pv.sdotsp.b acc8, w3, x1

end

Load 16 weights (8-bit)
4 out chan, 4 in chan
address post-increment





Optimized Back-End AI Kernels

Target **int8** execution of CONV, FC, ... primitives

1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

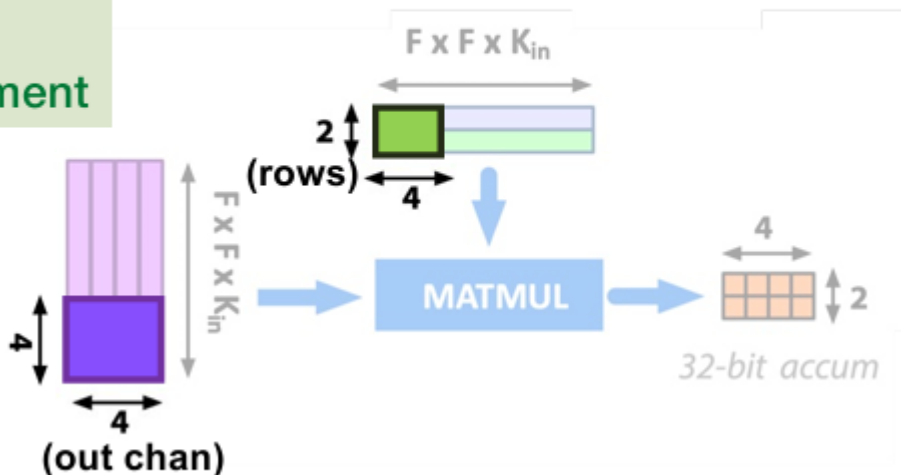
lp.setup

```
p.lw w0, 4(W0!)  
p.lw w1, 4(W1!)  
p.lw w2, 4(W2!)  
p.lw w3, 4(W3!)  
p.lw x1, 4(X0!)  
p.lw x2, 4(X1!)
```

Load 8 pixels
2 rows, 4 in chan
address post-increment

```
pv.sdotsp.b acc1, w0, x0  
pv.sdotsp.b acc2, w0, x1  
pv.sdotsp.b acc3, w1, x0  
pv.sdotsp.b acc4, w1, x1  
pv.sdotsp.b acc5, w2, x0  
pv.sdotsp.b acc6, w2, x1  
pv.sdotsp.b acc7, w3, x0  
pv.sdotsp.b acc8, w3, x1
```

end





Optimized Back-End AI Kernels

Target **int8** execution of CONV, FC, ... primitives

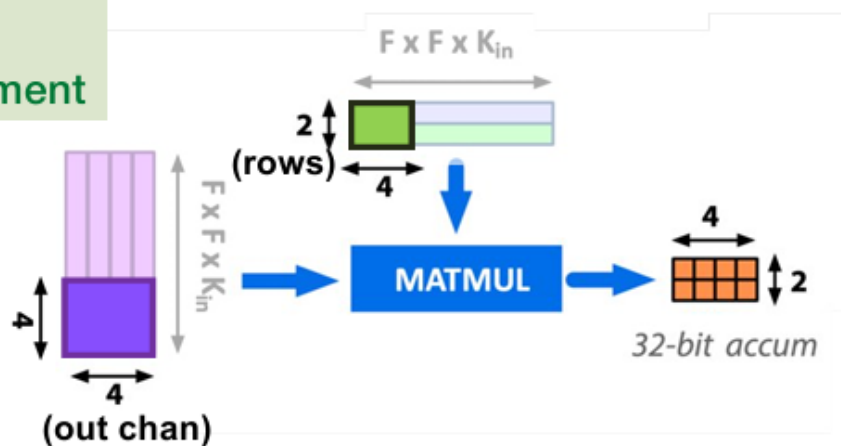
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

lp.setup

```
p.lw w0, 4(W0!)  
p.lw w1, 4(W1!)  
p.lw w2, 4(W2!)  
p.lw w3, 4(W3!)  
p.lw x1, 4(X0!)  
p.lw x2, 4(X1!)  
pv.sdosp.b acc1, w0, x0  
pv.sdosp.b acc2, w0, x1  
pv.sdosp.b acc3, w1, x0  
pv.sdosp.b acc4, w1, x1  
pv.sdosp.b acc5, w2, x0  
pv.sdosp.b acc6, w2, x1  
pv.sdosp.b acc7, w3, x0  
pv.sdosp.b acc8, w3, x1
```

end

Load 8 pixels
2 rows, 4 in chan
address post-increment



Compute 32 MAC over 8 accumulators
dot-product instructions



Optimized Back-End AI Kernels

Target **int8** execution of CONV, FC, ... primitives

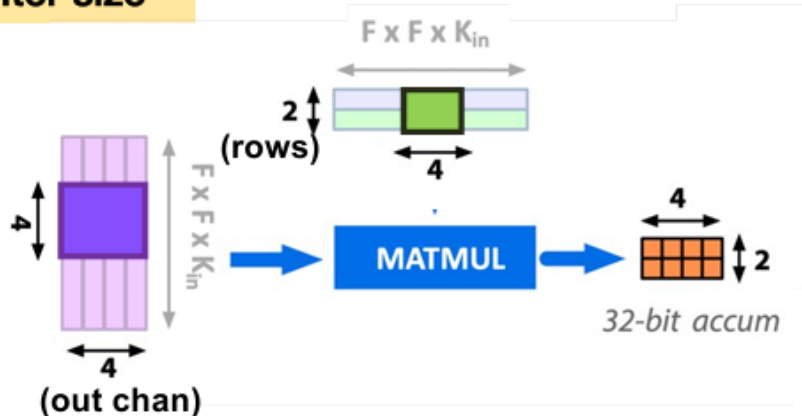
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

lp.setup

```
p.lw w0, 4(W0!)  
p.lw w1, 4(W1!)  
p.lw w2, 4(W2!)  
p.lw w3, 4(W3!)  
p.lw x1, 4(X0!)  
p.lw x2, 4(X1!)  
pv.sdotsp.b acc1, w0, x0  
pv.sdotsp.b acc2, w0, x1  
pv.sdotsp.b acc3, w1, x0  
pv.sdotsp.b acc4, w1, x1  
pv.sdotsp.b acc5, w2, x0  
pv.sdotsp.b acc6, w2, x1  
pv.sdotsp.b acc7, w3, x0  
pv.sdotsp.b acc8, w3, x1
```

end

Loop over in chan, filter size



Comp eff: 8ops/14insn = 0.57



Fused MAC-Load SIMD Instruction

- **Goal:** Increase Operational Efficiency, i.e. reducing the cost of LOADs
- **Observation 1:** Regular L1 Memory Access Pattern on MatMuls
- **Observation 2:** Create dedicated registers (NN-RF) to alleviate the pressure on the GP-RF and increase data reuse
- **Intuition:**
 - Fuse SIMD (s)dotp and LOAD ops in one single-cycle latency insns -> reduce the cost of LOADs
 - Inputs for (s)dotp/output for LOAD from/to NN-RF -> alleviate pressure on GP-RF of the processor
- **SW:** optimized routines to use mac-load, and intense data reuse strategy at RF/NN-RF level

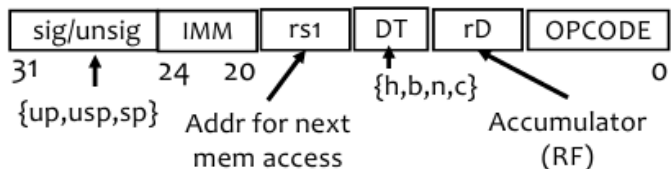


Fused MAC-Load SIMD Instruction

- The MAC-Load specifies the operations on a dedicated NN-RF
- After DSE, the NN-RF is partitioned with 2 slots (regs) for “activations” and 4 slots for “weights”

Mac-load insn encoding
(*custom I-type insn*)

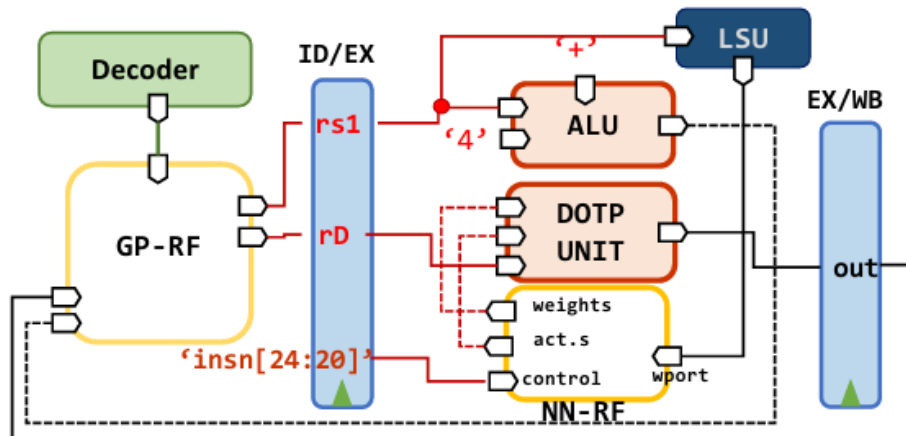
pv.mlsdot{up,usp,sp}.{h,b,n,c} rD, rs1, IMM



5-b IMMEDIATE description

- 4 : update/do not update weights NN-RF
- 3 : update/do not update activations NN-RF
- 2..1 : address of the weight NN-RF $0 \leq i \leq 3$
- 0 : address of the activation NN-RF $0 \leq j \leq 1$

Mac-load micro-archi





Performance Analysis on Assembly Code

8-bit innermost loop of the 4x2 *Matmul* kernel of the PULP-NN library

```
lp.setup    l1, l2, end
p.lw        w1, 4(aw1!)
p.lw        w2, 4(aw2!)
p.lw        w3, 4(aw3!)
p.lw        w4, 4(aw4!)
p.lw        x1, 4(ax1!)
p.lw        x2, 4(ax2!)
pv.sdotusp.b s1, x1, w1
pv.sdotusp.b s2, x1, w2
pv.sdotusp.b s3, x1, w3
pv.sdotusp.b s4, x1, w4
pv.sdotusp.b s5, x2, w1
pv.sdotusp.b s6, x2, w2
pv.sdotusp.b s7, x2, w3
end: pv.sdotusp.b s8, x2, w4
```

HW LOOP

LD/ST WITH POST INCREMENT

8-B SIMD MAC

8 SIMD MACs with 6 explicit LOADs

W_i X_i : weight/activation elements

AW_i AX_i : addresses for the MEM access

S_i : accumulators

L_i : loop setup

8-bit innermost loop of the 4x2 *MatMul* kernel of the extended PULP-NN library with *nnsdotp*

```
INIT NN-RF [ pv.mlsdotusp.h zero, aw1, 16
              pv.mlsdotusp.h zero, aw2, 18
              pv.mlsdotusp.h zero, aw3, 20
              pv.mlsdotusp.h zero, aw4, 22
              pv.mlsdotusp.h zero, ax1, 8
              lp.setup      l1, l2, end
              pv.mlsdotup.h zero, ax2, 9
              pv.mlsdotusp.b s1, aw2, 0
              pv.mlsdotusp.b s2, aw4, 2
              pv.mlsdotusp.b s3, aw3, 4
              pv.mlsdotusp.b s4, ax1, 14
              pv.mlsdotusp.b s5, aw2, 17
              pv.mlsdotusp.b s6, aw4, 19
              pv.mlsdotusp.b s7, aw3, 21
end: pv.mlsdotusp.b s8, aw1, 23
```

8 SIMD MACs with 1 explicit LOAD

$OpEff = 0.89$

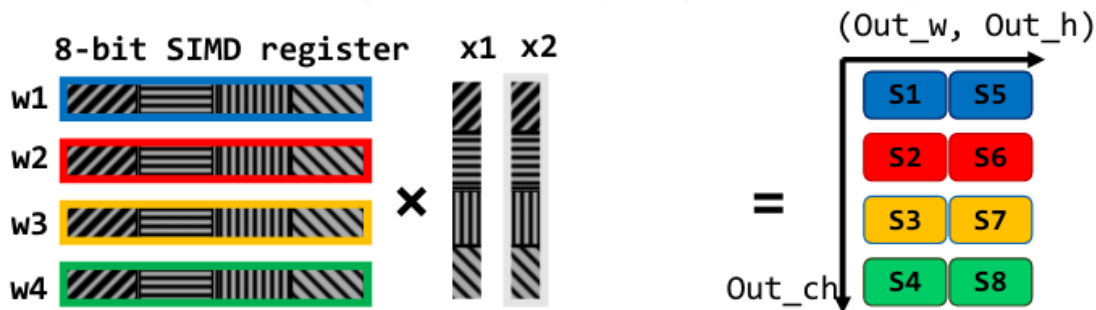
Can we still do better?



Goal: Increase Operational Efficiency

Can We Do Better? Analysis of Processor Resources

MatMul Innermost loop structure (example on 8-bit)



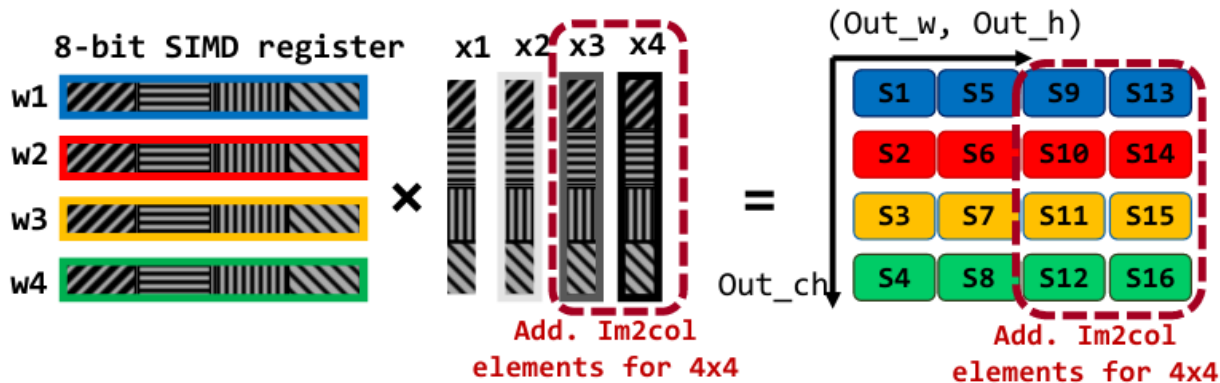
Register File Resources required	4x2 kernel
	SIMD
Loop Setup	2
Addresses for MEM access	6
MAC operands	6
MAC accumulators	8
Total	22 regs



Goal: Increase Operation Efficiency

Can We Do Better? Analysis of Processor Resources

MatMul Innermost loop structure (example on 8-bit)



Register File Resources required	4x2 kernel
	SIMD
Loop Setup	2
Addresses for MEM access	6
MAC operands	6
MAC accumulators	8
Total	22 regs

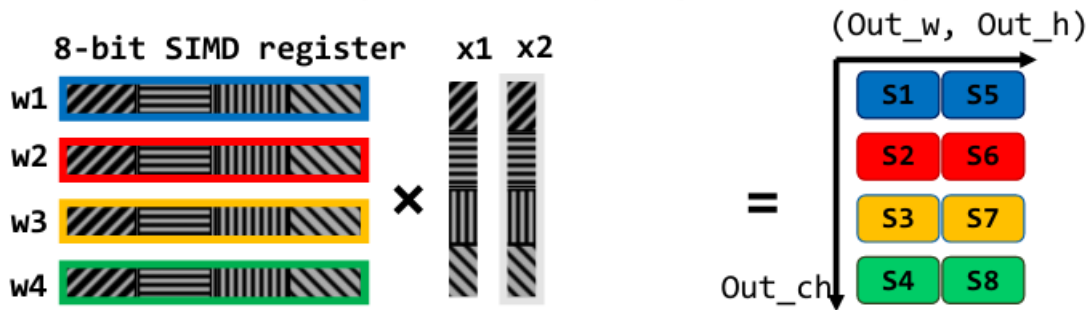
4x4 kernel
SIMD
2
8
8
16
<u>34</u> regs



Goal: Increase Operation Efficiency

Can We Do Better? Analysis of Processor Resources

MatMul Innermost loop structure (example on 8-bit)



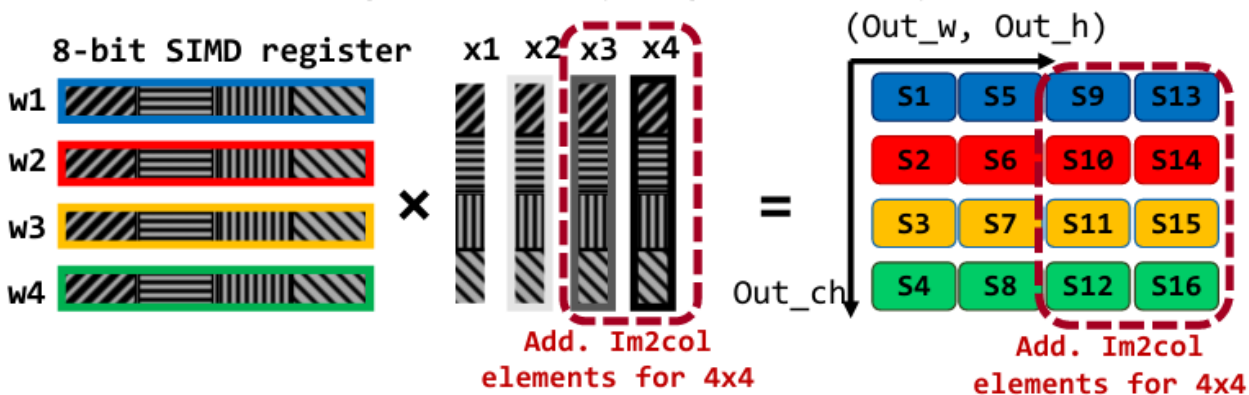
Register File Resources required	4x2 kernel		4x4 kernel
	SIMD	SIMD+m1sdotp	SIMD
Loop Setup	2	2	2
Addresses for MEM access	6	6	8
MAC operands	6	0 (op. from NN-RF)	8
MAC accumulators	8	8	16
Total	22 regs	16 regs	34 regs



Goal: Increase Operation Efficiency

Can We Do Better? Analysis of Processor Resources

MatMul Innermost loop structure (example on 8-bit)



Register File Resources required	4x2 kernel		4x4 kernel	
	SIMD	SIMD+m1sdotp	SIMD	SIMD+m1sdotp
Loop Setup	2	2	2	2
Addresses for MEM access	6	6	8	8
MAC operands	6	0 (op. from NN-RF)	8	0 (op. from NN-RF)
MAC accumulators	8	8	16	16
Total	22 regs	16 regs	34 regs	26 regs



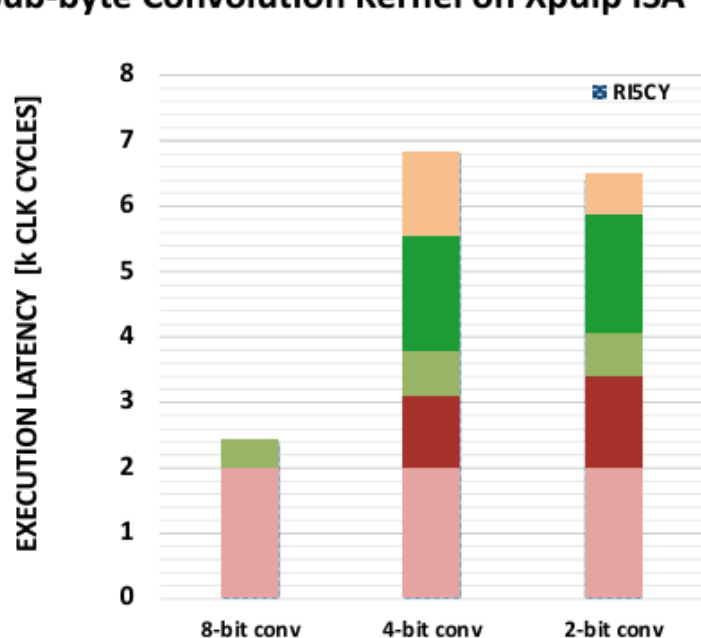
What About Low-Byte and Mixed-Precisions?

- **4-bit, 2-bit, 1-bit (or mixed-precision) common for some heavily QNNs**
 - Reduced footprint of the model -> good for memory and BW transfers
 - Computation on less bits -> Higher energy-efficiency if well supported by the processor
- **How to natively support 4-bit, 2-bit, 1-bit operations in the ISA**
 - Unpacking necessary to feed the 8-b SIMD units -> Huge Overhead !!



Analyzing Overheads to Eliminate Them

Sub-byte Convolution Kernel on Xpulp ISA



- MatMul kernel
- Overhead to extract compressed weights
- Im2col function
- Overhead to extract compressed im2col
- Re-Quantization



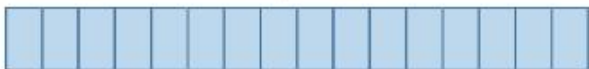
Objective: eliminate the overhead to boost the performance and the efficiency of sub-byte kernels



RISC-V Mixed-Precision ISA Extensions

Arithmetic SIMD instructions

32-bit operators



Byte SIMD operands (Xpulp) → 8-bit element

Nibble SIMD operands (XpulpNN) → 4-bit element

Crumb SIMD operands (XpulpNN) → 2-bit element

- No need to unpack sub-byte data on symmetric kernels

ALU SIMD Op.	Description for <i>nibble</i>
<code>pv.add[.sc].{n, c}</code>	$rD[i] = rs1[i] + rs2[i]$
<code>pv.sub[.sc].{n, c}</code>	$rD[i] = rs1[i] - rs2[i]$
<code>pv.avg(u)[.sc].{n, c}</code>	$rD[i] = (rs1[i] + rs2[i]) >> 1$
Vector Comparison Op.	
<code>pv.max(u)[.sc].{n, c}</code>	$rD[i] = rs1[i] > rs2[i] ? rs1[i] : rs2[i]$
<code>pv.min(u)[.sc].{n, c}</code>	$rD[i] = rs1[i] < rs2[i] ? rs1[i] : rs2[i]$
Vector Shift Op.	
<code>pv.srl[.sc].{n, c}</code>	$rD[i] = rs1[i] >> rs2[i]$ Shift is logical
<code>pv.sra[.sc].{n, c}</code>	$rD[i] = rs1[i] >> rs2[i]$ Shift is arithmetic
<code>pv.sll[.sc].{n, c}</code>	$rD[i] = rs1[i] << rs2[i]$
Vector abs Op.	
<code>pv.abs.{n, c}</code>	$rD[i] = rs1[i] < 0 ? -rs1[i] : rs1[i]$
Dot Product Op.	
<code>pv.dotup[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7]$
<code>pv.dotusp[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7]$
<code>pv.dotsp[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7]$
<code>pv.sdotup[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7] + rD$
<code>pv.sdotusp[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7] + rD$
<code>pv.sdotsp[.sc].{n, c}</code>	$rD = rs1[0]*rs2[0] + \dots + rs1[7]*rs2[7] + rD$



Mixed Precision

- **High Performance Overhead !!**
 - Many extra instructions for bit manipulation

8x4 SIMD sdotp operation

RI5CY Assembly

```
p.lw  x10,4(x4!)  
p.lw  x11,4(x5!)  
p.extract x5, x11, 4, 0  
p.extract x6, x11, 4, 4  
p.extract x7, x11, 4, 8  
p.extract x8, x11, 4, 12  
pv.packlo.b x15, x5, x6  
pv.packhi.b x15, x7, x8  
pv.sdotsp.b x20, x15, x10
```



Mixed Precision

- More than 100 dotp instructions needed to support all various precision permutations
 - High encoding space and high decoding stage complexity

8x4 SIMD sdotp operation

RI5CY Assembly

```
p.lw x10,4(x4!)
p.lw x11,4(x5!)
p.extract x5, x11, 4, 0
p.extract x6, x11, 4, 4
p.extract x7, x11, 4, 8
p.extract x8, x11, 4, 12
pv.packlo.b x15, x5, x6
pv.packhi.b x15, x7, x8
pv.sdotsp.b x20, x15, x10
```

ISA extensions

```
p.lw x10,4(x4!)
p.lw x11,4(x5!)
pv.sdotsp.MIX8x4
x20,x11,x10
```

□ One instruction per format and type

Standard Instructions

pv.sdotsp.h	pv.sdotsp.h	pv.sdotsp.h	pv.sdotsp.h	pv.sdotsp.h	pv.sdotsp.h
pv.sdotsp.b	pv.sdotsp.b	pv.sdotsp.b	pv.sdotsp.b	pv.sdotsp.b	pv.sdotsp.b
pv.sdotsp.n	pv.sdotsp.n	pv.sdotsp.n	pv.sdotsp.n	pv.sdotsp.n	pv.sdotsp.n
pv.sdotsp.c	pv.sdotsp.c	pv.sdotsp.c	pv.sdotsp.c	pv.sdotsp.c	pv.sdotsp.c
pv.sdotsp.m4x2	pv.sdotsp.m4x2	pv.sdotsp.m4x2	pv.sdotsp.m4x2	pv.sdotsp.m4x2	pv.sdotsp.m4x2
pv.sdotsp.m8x2	pv.sdotsp.m8x2	pv.sdotsp.m8x2	pv.sdotsp.m8x2	pv.sdotsp.m8x2	pv.sdotsp.m8x2
pv.sdotsp.m8x4	pv.sdotsp.m8x4	pv.sdotsp.m8x4	pv.sdotsp.m8x4	pv.sdotsp.m8x4	pv.sdotsp.m8x4
pv.sdotsp.m16x8	pv.sdotsp.m16x8	pv.sdotsp.m16x8	pv.sdotsp.m16x8	pv.sdotsp.m16x8	pv.sdotsp.m16x8
pv.sdotsp.m16x4	pv.sdotsp.m16x4	pv.sdotsp.m16x4	pv.sdotsp.m16x4	pv.sdotsp.m16x4	pv.sdotsp.m16x4
pv.sdotsp.m16x2	pv.sdotsp.m16x2	pv.sdotsp.m16x2	pv.sdotsp.m16x2	pv.sdotsp.m16x2	pv.sdotsp.m16x2
pv.sdotsp.sc.h	pv.sdotsp.sc.h	pv.sdotsp.sc.h	pv.sdotsp.sc.h	pv.sdotsp.sc.h	pv.sdotsp.sc.h
pv.sdotsp.sc.b	pv.sdotsp.sc.b	pv.sdotsp.sc.b	pv.sdotsp.sc.b	pv.sdotsp.sc.b	pv.sdotsp.sc.b
pv.sdotsp.sc.c	pv.sdotsp.sc.c	pv.sdotsp.sc.c	pv.sdotsp.sc.c	pv.sdotsp.sc.c	pv.sdotsp.sc.c
pv.sdotsp.sc.n	pv.sdotsp.sc.n	pv.sdotsp.sc.n	pv.sdotsp.sc.n	pv.sdotsp.sc.n	pv.sdotsp.sc.n
pv.sdotsp.sc.m4x2	pv.sdotsp.sc.m4x2	pv.sdotsp.sc.m4x2	pv.sdotsp.sc.m4x2	pv.sdotsp.sc.m4x2	pv.sdotsp.sc.m4x2
pv.sdotsp.sc.m8x2	pv.sdotsp.sc.m8x2	pv.sdotsp.sc.m8x2	pv.sdotsp.sc.m8x2	pv.sdotsp.sc.m8x2	pv.sdotsp.sc.m8x2
pv.sdotsp.sc.m8x4	pv.sdotsp.sc.m8x4	pv.sdotsp.sc.m8x4	pv.sdotsp.sc.m8x4	pv.sdotsp.sc.m8x4	pv.sdotsp.sc.m8x4
pv.sdotsp.sc.m16x8	pv.sdotsp.sc.m16x8	pv.sdotsp.sc.m16x8	pv.sdotsp.sc.m16x8	pv.sdotsp.sc.m16x8	pv.sdotsp.sc.m16x8
pv.sdotsp.sc.m16x4	pv.sdotsp.sc.m16x4	pv.sdotsp.sc.m16x4	pv.sdotsp.sc.m16x4	pv.sdotsp.sc.m16x4	pv.sdotsp.sc.m16x4
pv.sdotsp.sc.m16x2	pv.sdotsp.sc.m16x2	pv.sdotsp.sc.m16x2	pv.sdotsp.sc.m16x2	pv.sdotsp.sc.m16x2	pv.sdotsp.sc.m16x2
pv.sdotsp.sci.h	pv.sdotsp.sci.h	pv.sdotsp.sci.h	pv.sdotsp.sci.h	pv.sdotsp.sci.h	pv.sdotsp.sci.h
pv.sdotsp.sci.b	pv.sdotsp.sci.b	pv.sdotsp.sci.b	pv.sdotsp.sci.b	pv.sdotsp.sci.b	pv.sdotsp.sci.b
pv.sdotsp.sci.c	pv.sdotsp.sci.c	pv.sdotsp.sci.c	pv.sdotsp.sci.c	pv.sdotsp.sci.c	pv.sdotsp.sci.c
pv.sdotsp.sci.n	pv.sdotsp.sci.n	pv.sdotsp.sci.n	pv.sdotsp.sci.n	pv.sdotsp.sci.n	pv.sdotsp.sci.n
pv.sdotsp.sci.m4x2	pv.sdotsp.sci.m4x2	pv.sdotsp.sci.m4x2	pv.sdotsp.sci.m4x2	pv.sdotsp.sci.m4x2	pv.sdotsp.sci.m4x2
pv.sdotsp.sci.m8x2	pv.sdotsp.sci.m8x2	pv.sdotsp.sci.m8x2	pv.sdotsp.sci.m8x2	pv.sdotsp.sci.m8x2	pv.sdotsp.sci.m8x2
pv.sdotsp.sci.m8x4	pv.sdotsp.sci.m8x4	pv.sdotsp.sci.m8x4	pv.sdotsp.sci.m8x4	pv.sdotsp.sci.m8x4	pv.sdotsp.sci.m8x4
pv.sdotsp.sci.m16x8	pv.sdotsp.sci.m16x8	pv.sdotsp.sci.m16x8	pv.sdotsp.sci.m16x8	pv.sdotsp.sci.m16x8	pv.sdotsp.sci.m16x8
pv.sdotsp.sci.m16x4	pv.sdotsp.sci.m16x4	pv.sdotsp.sci.m16x4	pv.sdotsp.sci.m16x4	pv.sdotsp.sci.m16x4	pv.sdotsp.sci.m16x4
pv.sdotsp.sci.m16x2	pv.sdotsp.sci.m16x2	pv.sdotsp.sci.m16x2	pv.sdotsp.sci.m16x2	pv.sdotsp.sci.m16x2	pv.sdotsp.sci.m16x2



Mixed Precision: Dynamic Bit-Scalable Execution

8x4 SIMD sdotp operation

RISCV Assembly

```
p.lw x10,4(x4!)  
p.lw x11,4(x5!)  
p.extract x5, x11, 4, 0  
p.extract x6, x11, 4, 4  
p.extract x7, x11, 4, 8  
p.extract x8, x11, 4, 12  
pv.packlo.b x15, x5, x6  
pv.packhi.b x15, x7, x8  
pv.sdotsp.b x20, x15, x10
```

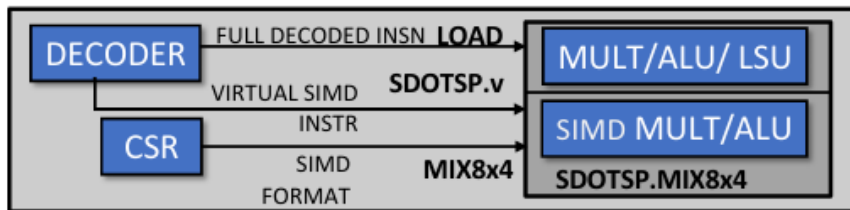
ISA extensions

```
p.lw x10,4(x4!)  
p.lw x11,4(x5!)  
pv.sdotsp.MIX8x4  
x20,x11,x10
```

- One instruction per format and type

Dynamic Bit-Scalable Execution

```
p.lw x10,4(x4!)  
p.lw x11,4(x5!)  
pv.sdotsp.v x20,x11,x10
```



Virtual Instructions

```
pv.sdotsp.v  
pv.sdotsp.sc.v  
pv.sdotsp.sci.v  
pv.sdotup.v  
pv.sdotup.sc.v  
pv.sdotup.sci.v  
pv.sdotusp.v  
pv.sdotusp.sc.v  
pv.sdotusp.sci.v  
pv.sdotusp.v  
pv.sdotusp.sc.v  
pv.sdotusp.sci.v
```



Mixed Precision: Dynamic Bit-Scalable Execution

8x4 SIMD sdotp operation

RI5CY Assembly

```
p.lw x10,4(x4!)
p.lw x11,4(x5!)
p.extract x5, x11, 4, 0
p.extract x6, x11, 4, 4
p.extract x7, x11, 4, 8
p.extract x8, x11, 4, 12
pv.packlo.b x15, x5, x6
pv.packhi.b x15, x7, x8
pv.sdotsp.b x20, x15, x10
```

ISA extensions

```
p.lw x10,4(x4!)
p.lw x11,4(x5!)
pv.sdotsp.MIX8x4
x20,x11,x10
```

- One instruction per format and type

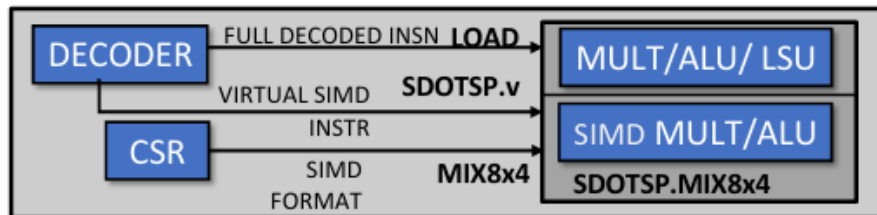
Dynamic Bit-Scalable Execution

```
p.lw x10,4(x4!)
p.lw x11,4(x5!)
pv.sdotsp.v x20,x11,x10
```

Reuse of Code

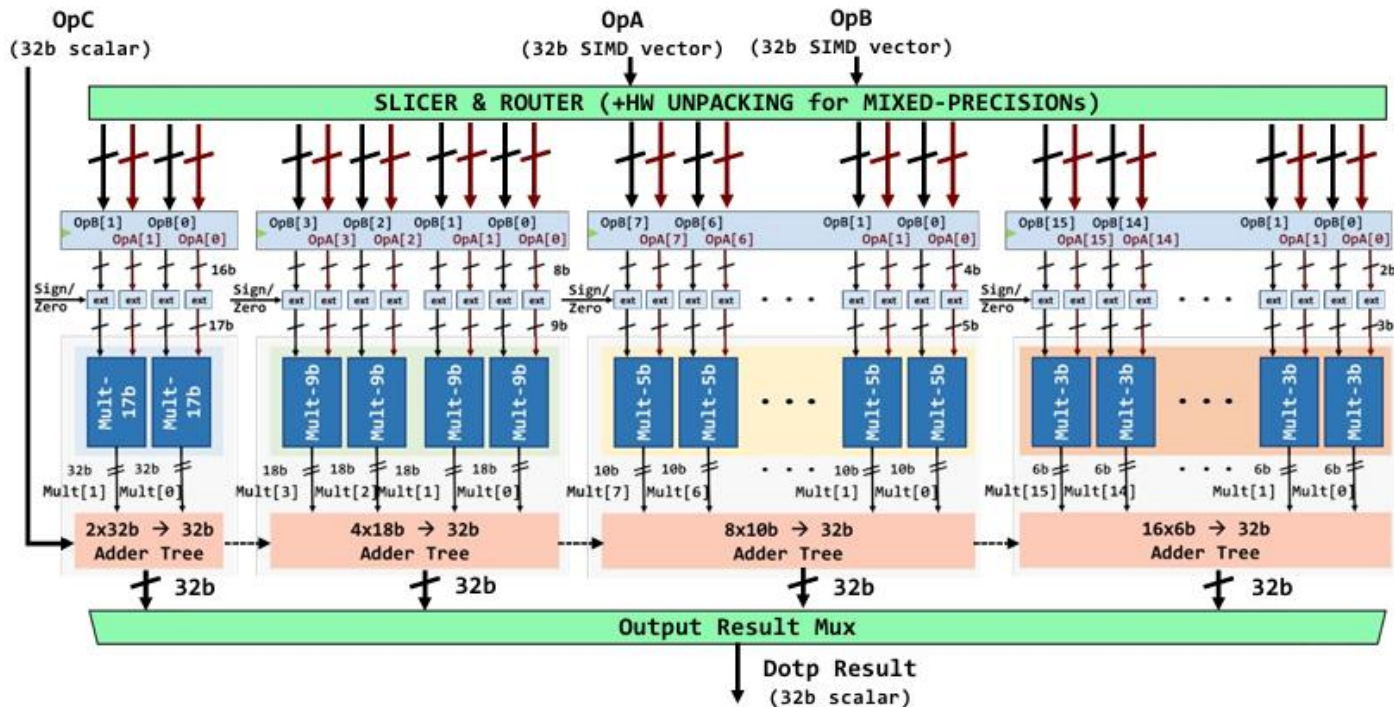
```
int main()
{
....
SIMD_FMT(M8x4);
convolution(A, W, Res);
....
SIMD_FMT(M8x2);
convolution(A, W, Res);
....}
```

- Virtual Instruction
 - Format of SIMD instructions specified at runtime by a CSR
 - CSR written by the core before SIMD insn exec
 - Tiny overhead in QNN computation
- No effort in adding future formats
- Reusable code





Multi-Precision Integer SIMD sdotp Unit

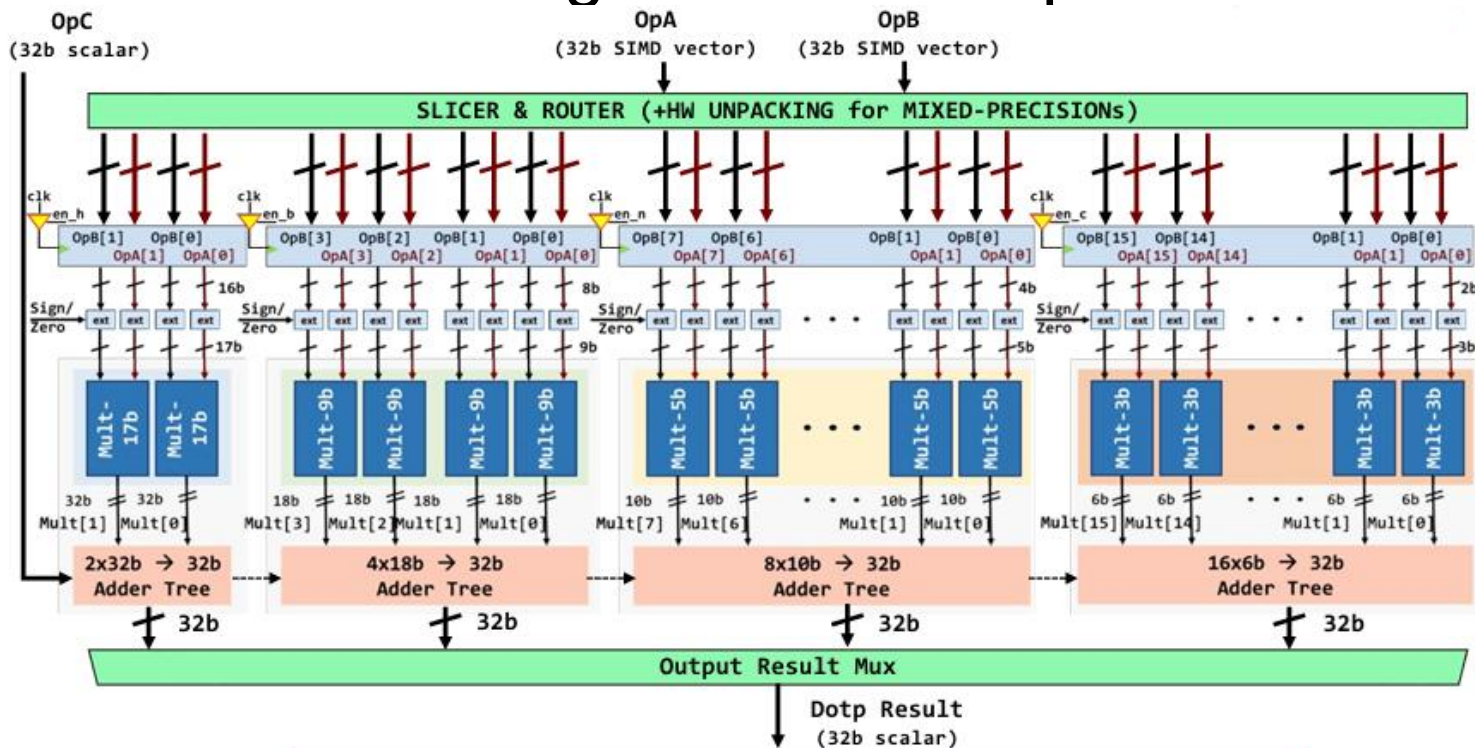


- RISCY already features 16-b/8-b SIMD datapath
- We add different multiplier regions dedicated to each "precision" (4-b/2-b)
- **Pros:** do not increase the critical path of the core
- **Cons:** higher power consumption due to additional multipliers

Can we reduce switching activities of unused "regions"?



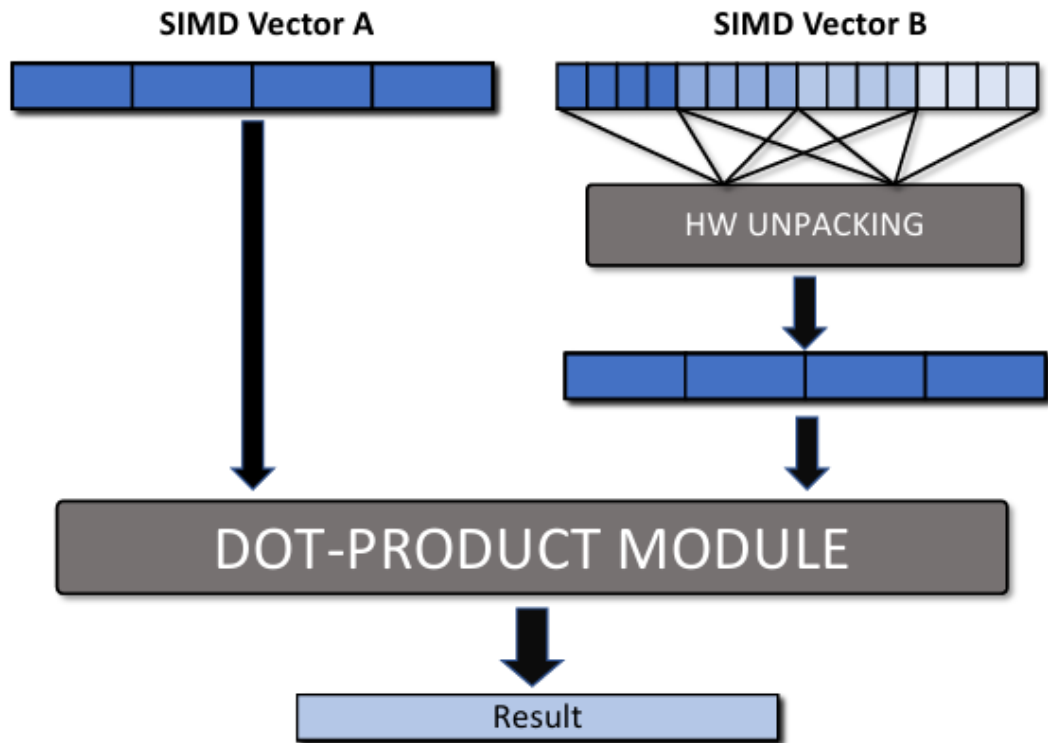
Multi-Precision Integer SIMD sdotp Unit



Selective Clock Gating of Input/Output Registers + Operand Isolation



Mixed-Precision Multipliers

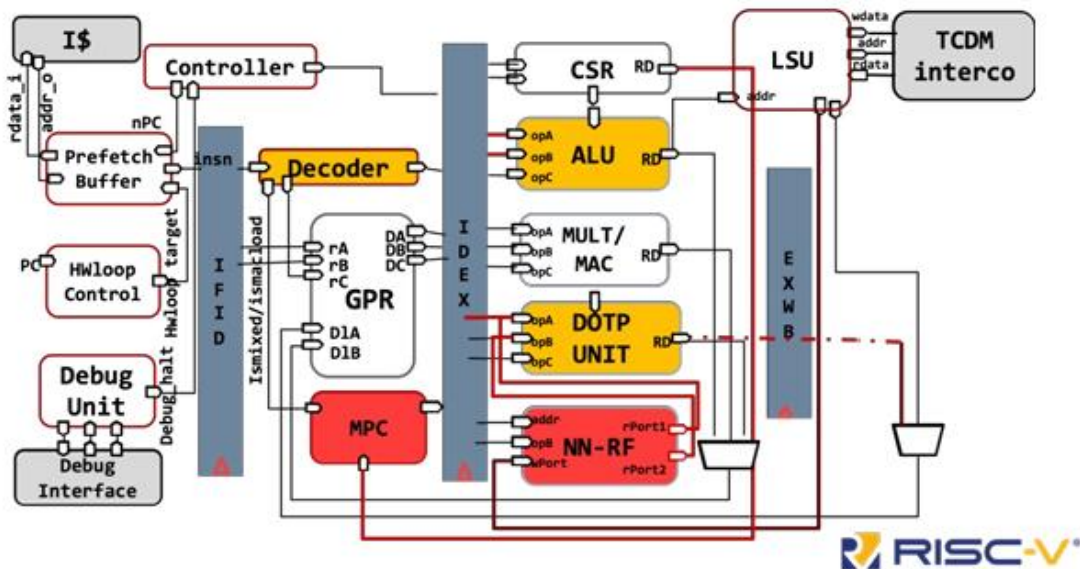


- Mixed-Precision operations require a controller: selection of the correct sub-word of the lowest precision operand (Vector B) to be used in current SIMD op.
- ☐ The controller is programmed by control status registers (CSRs).



Extended uArchi to support Mixed-Precision

4-stage, single in-order issue pipeline



- ❑ [Baseline] RV32IMC: not good for machine learning workload (~44kGE)
- ❑ [DSP] Xpulp: 16/8-bit SIMD, HW loops, post-modified LD/ST & bit-manip. insns (~55kGE)
- ❑ [AI] XpulpNN: 4-bit, 2-bit SIMD (and mixed-precisions) & fused MAC-Load ops (~65kGE)



DNN Acceleration Through ISA Extensions

8-bit Convolution Kernel

RV32IMC

N

```
addi a0,a0,1
addi t1,t1,1
addi t3,t3,1
addi t4,t4,1
lbu a7,-1(a0)
lbu a6,-1(t4)
lbu a5,-1(t3)
lbu t5,-1(t1)
mul s1,a7,a6
mul a7,a7,a5
add s0,s0,s1
mul a6,a6,t5
add t0,t0,a7
mul a5,a5,t5
add t2,t2,a6
add t6,t6,a5
bne s5,a0,1c000bc
```

RV32IMCXpulp [DSP]

N/4

```
lp.setup
p.lw w1, 4(a0!)
p.lw w2, 4(a1!)
p.lw x1, 4(a2!)
p.lw x2, 4(a3!)
pv.sdotsp.b s1, w1, x1
pv.sdotsp.b s2, w1, x2
pv.sdotsp.b s3, w2, x1
pv.sdotsp.b s4, w2, x2
End: ...
```

HW Loop

LD/ST with post increment

8-bit SIMD sdotp

9x less insns than RV32IMC
22% area & 9% power overhead over RV32IMC

RV32IMCXpulpNN [AI]

N/4

```
Init NN-RF (outside of the loop)
lp.setup
pv.nnsdotup.h s0,ax1,9
pv.nnsdotsp.b s1,aw2,0
pv.nnsdotsp.b s2,aw4,2
pv.nnsdotsp.b s3,aw3,4
pv.nnsdotsp.b s4,ax1,14
end
```

8-bit SIMD Fused MAC-Load

Up to 1.7x less insns than Xpulp
Up to 94% Dotp-Unit utilization
18% & 2.5% extra-area & power than Xpulp

- Power ↑ (slightly); Latency ↓ ↓ ↓ (>10x) on QNN kernels → Energy ↓ ↓
- Parallelization of the workload (multi-core) for high throughput



How Much Does it Cost?

- Power, Performance, Area Analysis conducted by implementing the core in GF 22nm FDX

	Area[um2] Overhead vs baseline RI5CY [%]	Power @ 400 MHz [mW] Overhead vs baseline RI5CY [%] GP workload	Power @ 400 MHz [mW] Overhead vs baseline RI5CY [%] 8-b MatMul	Max Operating Frequency [MHz] in SS 0.75V 125C Overhead vs baseline RI5CY [%]
RI5CY (Xpulp)	24300	0.71 (-1.7%)	1.18	420
XpulpNN	26008 (10%)	0.70 (0.4%)	1.10 (-1.7%)	420
XpulpNN + M&L	27821 (17.9%)	0.72 (1.2%)	1.21 (7.7%)	415 (-1.1%)
XpulpNN + M&L + Mixed-Precision	31300 (32.7%)	0.72 (1.2%) 	1.27 (8.4%) 	412 (-1.9%)

- However, a core alone can not do much. It needs a system around it (mem hierarchy, DMA, ..)



Summary

- **SW & ISA**

- Assembly analysis gives a clear understanding of acceleration opportunities through ISA extensions
- SW needs to exploit the new/custom/extended ISA to be efficient at application level

- **Microarchitecture**

- Careful microarchitecture design to not introduce operational frequency regressions
- Fine-grained clock gating reduces drastically dynamic power consumption
- New ISA support do not jeopardize power figure of the processor on general-purpose applications



Data (or Vector) Level Parallelism

- **The least flexible form of parallelism**

- Among instruction level parallelism (ILP) and Thread level parallelism (TLP)
- Anything that can be expressed to exploit Data Level Parallelism (DLP) can also be written to exploit ILP or TLP
- A machine that exploits DLP only needs to fetch and decode a single instruction to describe a whole array of vector operations
- Very energy efficient !!
- Who is responsible for finding parallelism in the instruction flow?



From Packed-SIMD to Vector Processors

- **Vector seen as: one-dimensional array of numbers**
- **Many programs use vectors**
 - for ($i = 0; i \leq 49; i++$)
 $C[i] = (A[i] + B[i]) / 2$
- **Vector Processor's instructions operate on vector rather than scalar values**
- **Basic Requirements**
 - **Vector register file** -> Need to store/load the vectors
 - **Vector Length Register (VLEN -> 'vl' in RVV)** -> to set (with an instruction) the lengths of vectors on which we operate on



From Packed-SIMD to Vector Processors

- **Basic Requirements**

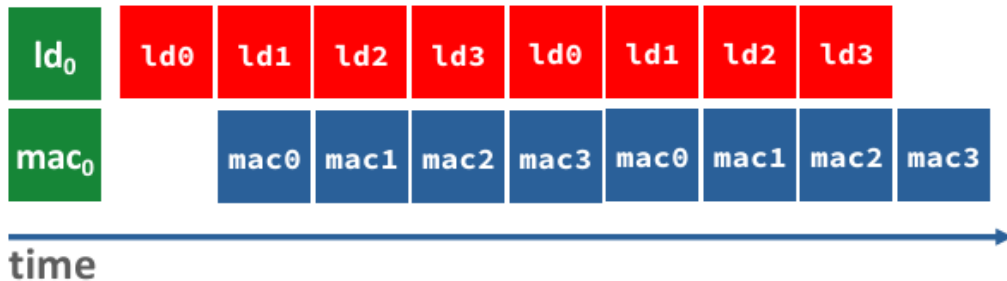
- **Vector Stride Register (VSTR)** -> to locate where vector elements are stored in memory -> in RISC-V encoded into the instruction
 - Vector elements might be stored apart from each other in memory
 - Stride -> distance in memory between two elements of the same vector



Vector Processors

- **Vector instruction performs an operation on each element in consecutive cycles**
 - Vector Functional Units can be pipelined
 - Each pipeline stage operates on a different data element
- **Vector instructions allow deeper pipelines in functional units because ..**

Cray-style Vector, 1 lane

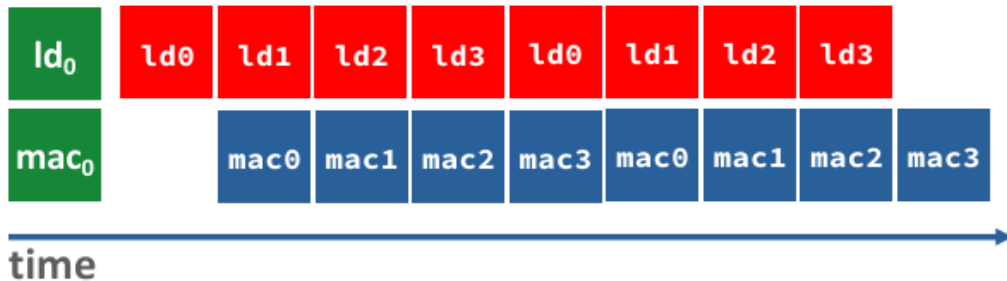




Vector Processors

- **Vector instructions allow deeper pipelines in functional units because ...**
 - **No intra-vector dependencies** -> No hardware interlocking needed within a vector
 - **No control flow** within a vector
 - **Known stride allow easy address calculation** for all vector elements

Cray-style Vector, 1 lane

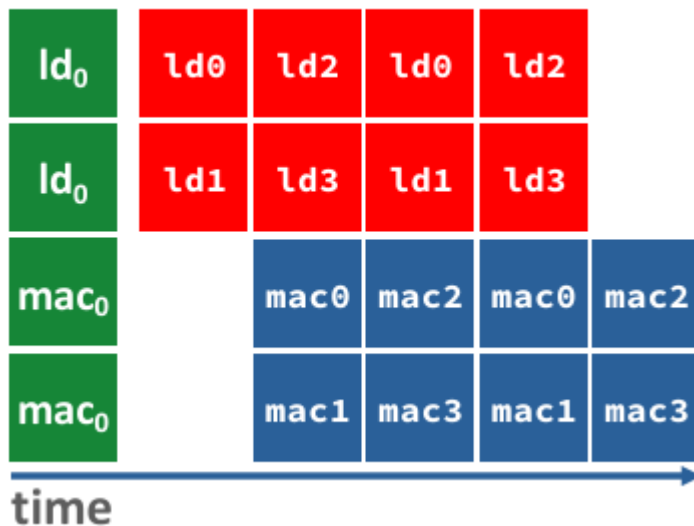




Vector Processors

- **We can exploit parallelism with multiple vector lanes**
 - This parallelism (multiple lanes) is transparent to the software

Cray-style Vector, 2 lanes





Vector Processor Advantages

- **No dependencies within a vector**
 - Pipelining and parallelization work really well
 - Can have very deep pipelines, no dependencies
- **Each instruction generates a lot of work**
 - Reduce instruction fetch bandwidth requirements
- **Highly regular memory access pattern**
- **No need to explicitly code loops**
 - Fewer branches in the instruction sequence

Works well only if parallelism is regular (data parallelism!)

If parallelism is irregular →
Very inefficient!



Vector Code Example

C code

```
for (i=0; i<64; i++)
```

```
  C[i] = A[i] + B[i]
```

Scalar Code

```
LI R4, 64
```

```
loop:
```

```
FLD F0, 0(R1)
```

```
FLD F2, 0(R2)
```

```
FADD F4, F2, F0
```

```
FSW F4, 0(R3)
```

```
ADDI R1, 8
```

```
ADDI R2, 8
```

```
ADDI R3, 8
```

```
ADDI R4, -1
```

```
BNEZ R4, loop
```

Vector Code

```
VSETVL 64
```

```
VLE V1, R1
```

```
VLE V2, R2
```

```
VADD V3, V1, V2
```

```
VSE V3, R3
```

RVV Code

```
li          t0, 64
```

```
vsetvli     t1, t0, e64
```

```
vle64.v     v1, (R1)
```

```
vle64.v     v2, (R2)
```

```
vfadd.vv    v3, v1, v2
```

```
vse64.v     v3, (R3)
```



Vector Instruction Set Advantages

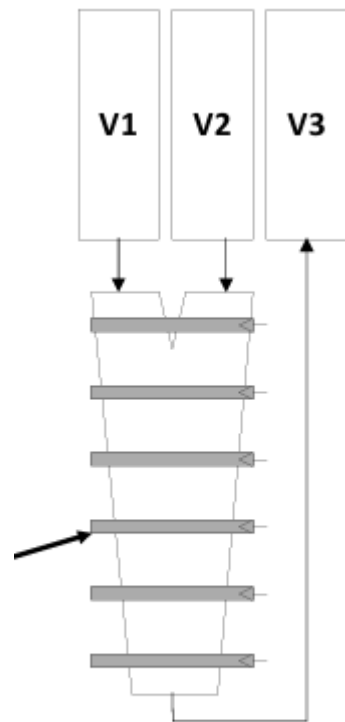
- **Compact**
 - One instruction (short) encodes N operations
- **Expressive, highly powerful semantic, tells the hardware that these N operations**
 - Are **INDEPENDENT**
 - Use the **same functional units**
 - Access **disjoint registers**
 - Access registers **with the same pattern as previous instructions**
 - Access either
 - A contiguous block of memory (unit-strided load/store)
 - A known memory pattern (strided load/store)



Vector Arithmetic Execution

- **Use deep pipelined functional units**
 - Fast clock to execute element-wise operations
- **Control of deep pipelines in VAU is simplified**
 - There are no hazards (elements in a vector are inherently independent!)

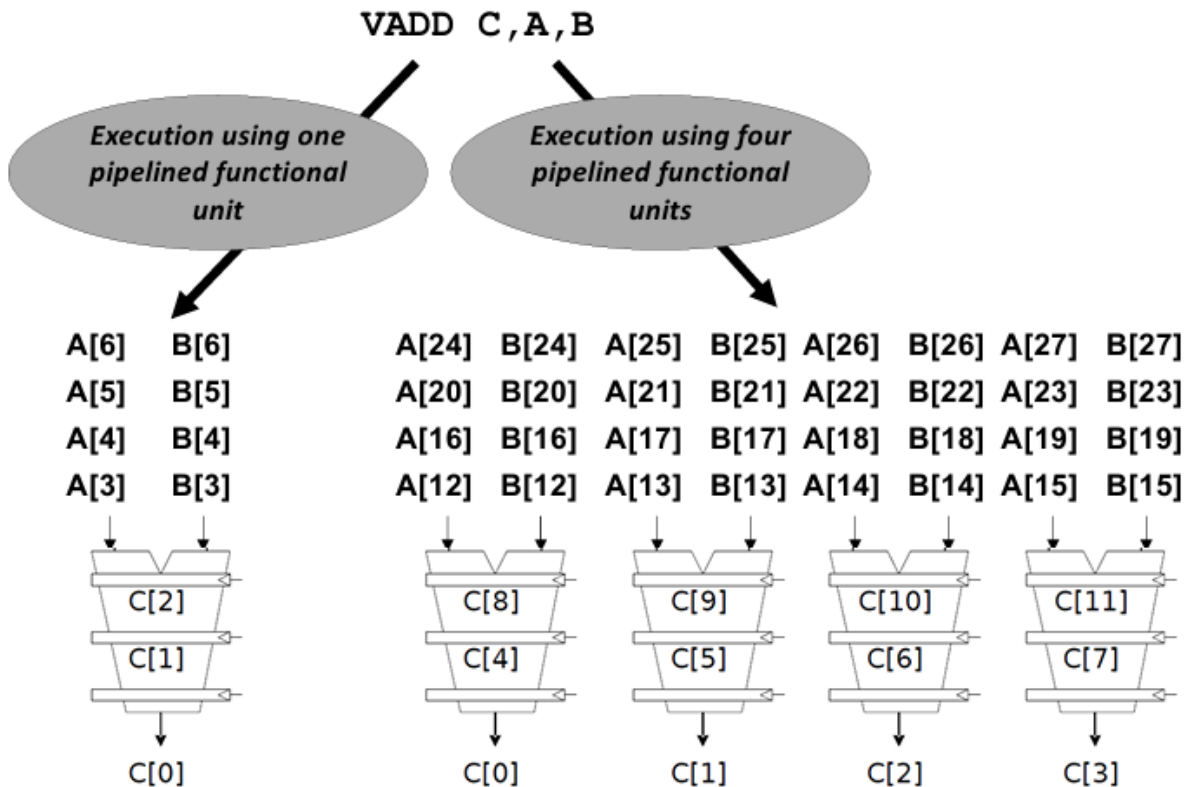
Six stage multiply pipeline



$v3 \leftarrow v1 * v2$

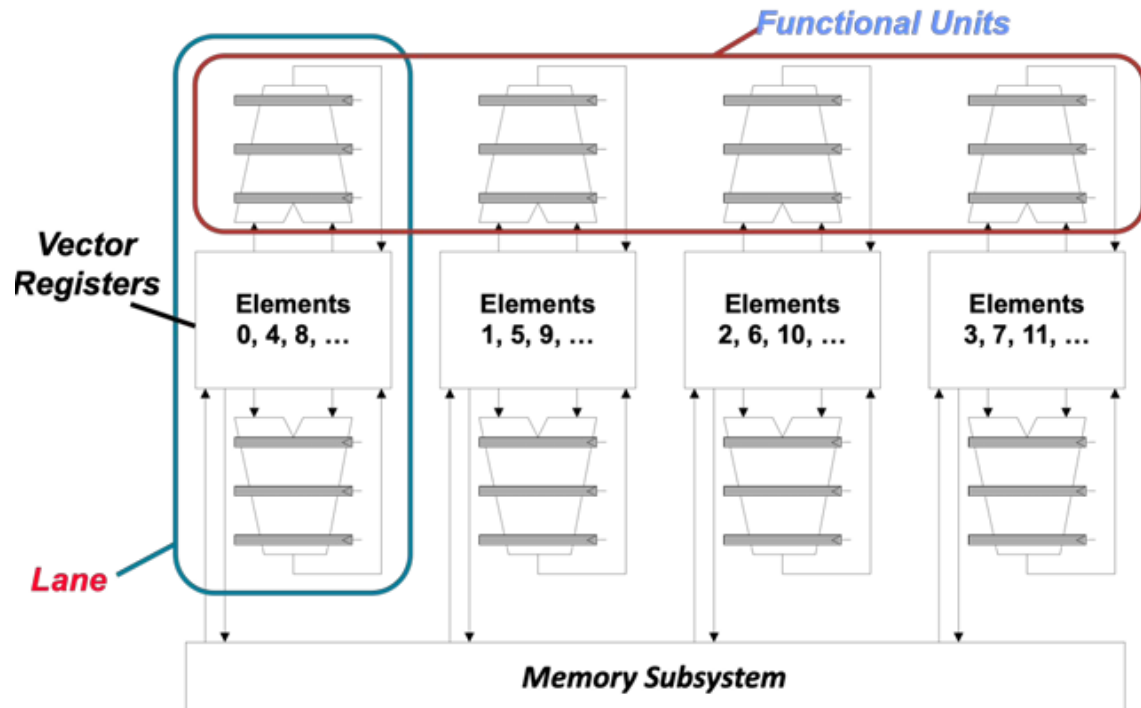


Vector Instruction Execution





Vector Unit Structure



- Each lane sees one portion of the VRF and one set of FUs
- Each lane computes in parallel independently from others (no communication)



Vector Unit Structure

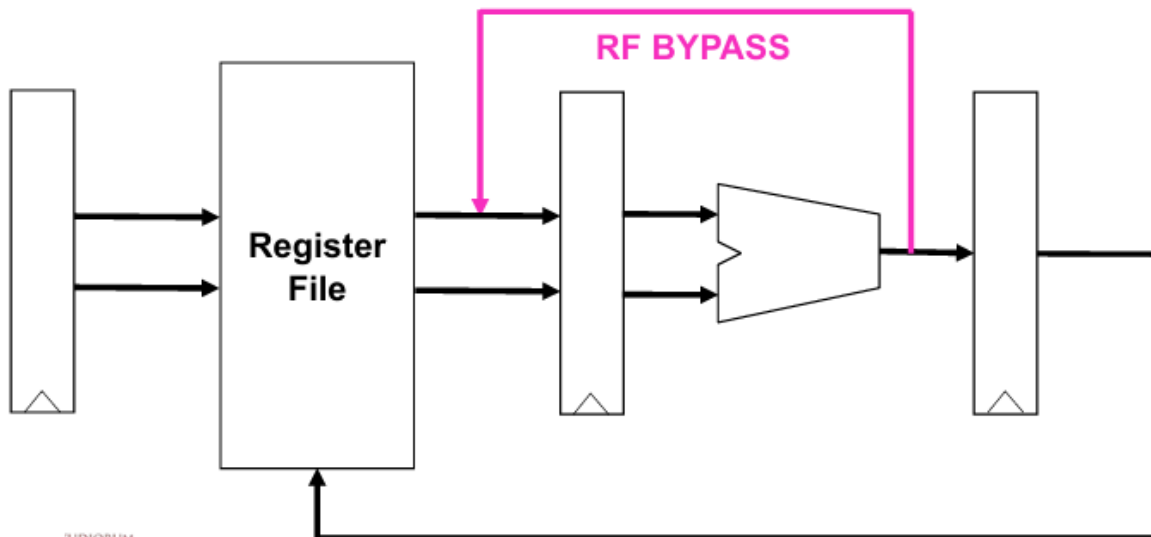
- Operator forwarding (register bypassing)

ADD **r1**, r2, r2

MUL r4, **r1**, r3

RAW dependency on **r1**!

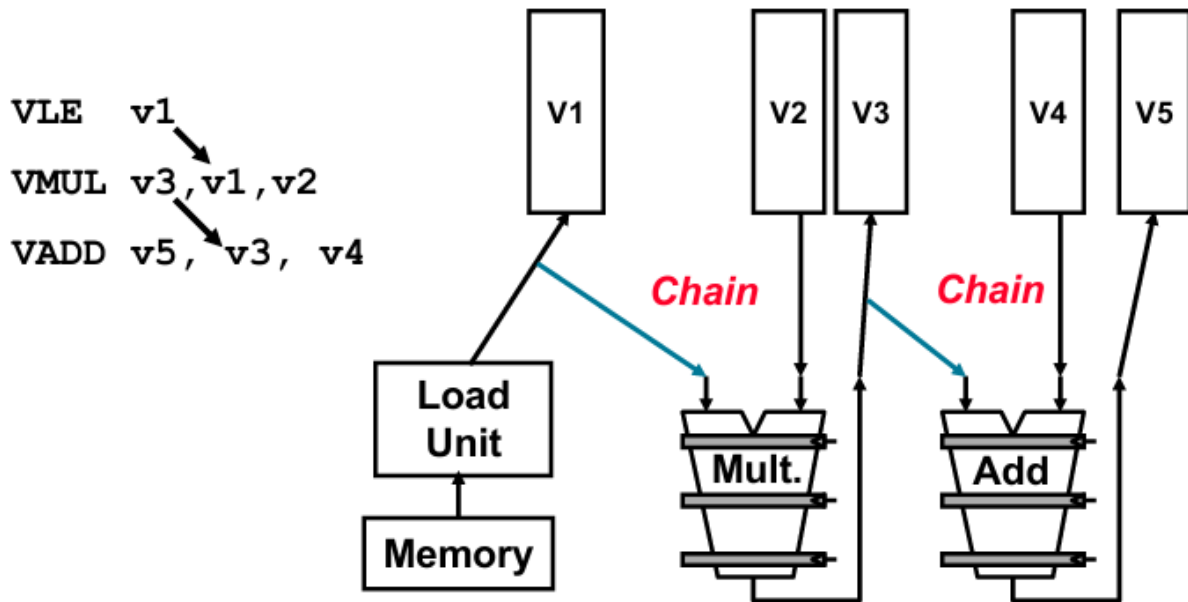
We should wait until LD has written
back before reading operands for MUL





Vector Chaining

- Vector version of register bypassing
- The VRF acts as a controller point



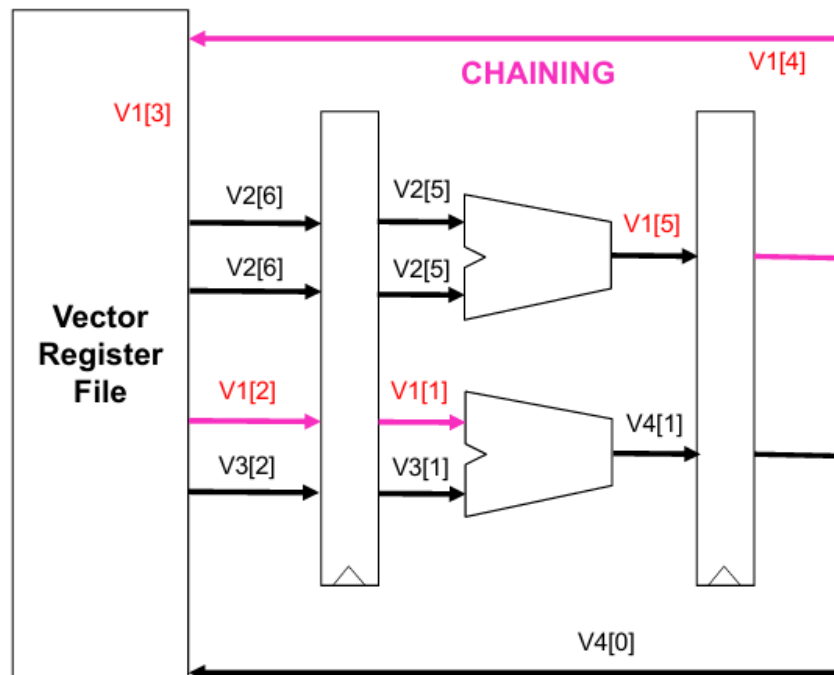


Vector Flexible Chaining

- We can use VRF as chaining node

VADD **v1**, v2, v2

VMUL v4, **v1**, v3



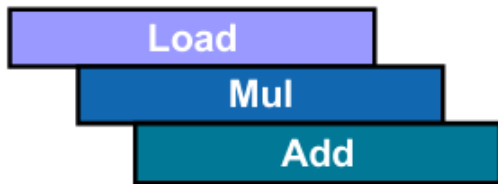


Vector Chaining

- Without chaining, must wait for last element of result to be written before starting dependent instruction



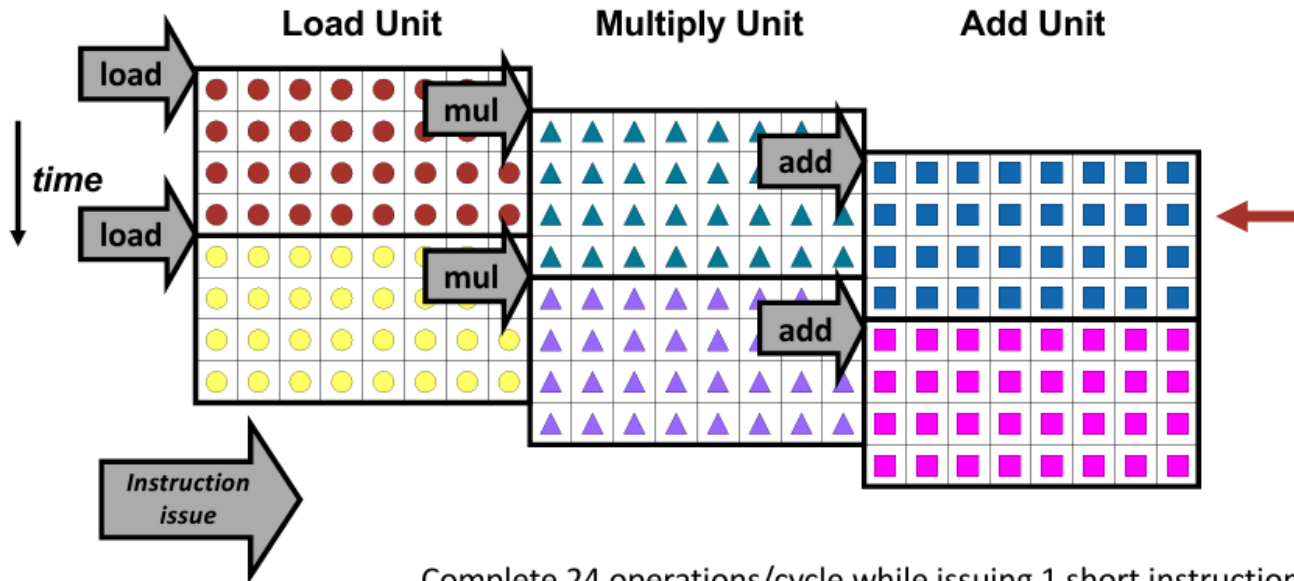
- With chaining, can start dependent instruction as soon as first result appears





Vector Instruction Parallelism

- **Can overlap execution of multiple vector instructions**
 - Example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle



A brief overview of vector execution

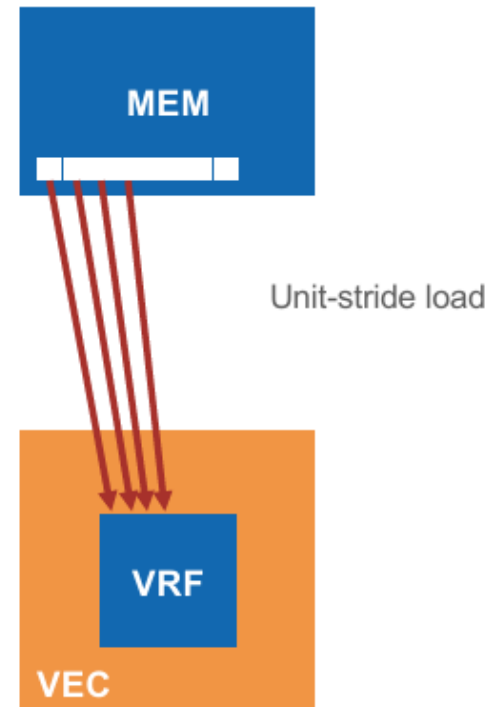
- **The vector elements to process are in memory**
- **The vector architecture contains a vector register file (VRF)**
- **Load vector elements from memory to the VRF**
 - The VRF buffers our vector close to the processing units
 - The VRF is the bottom of the memory hierarchy
 - The VRF increases data reuse by exploiting data locality
- **If the vector registers are too small to host a full vector?**
 - Only load and process the first chunk of the vector
 - Continue with the second chunk
 - Until to process the whole vector in memory





Vector Memory Operations

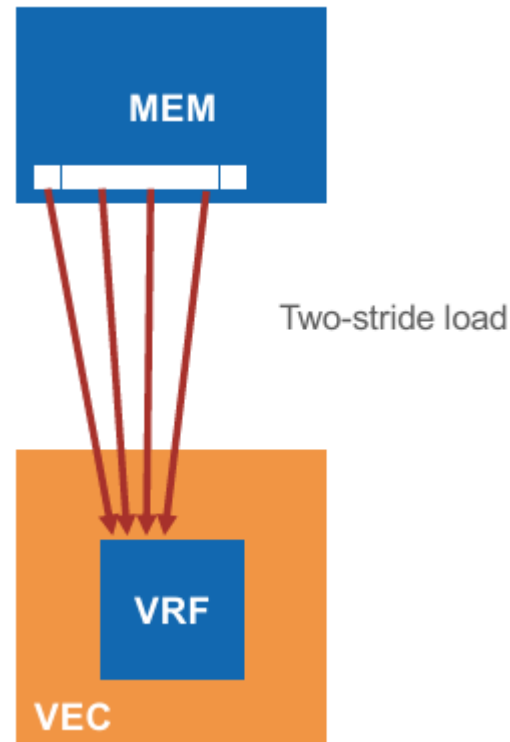
- **Load/store operations**
 - move group of data between registers and memory
- **Three types of addressing**
 - **Unit stride**
 - Contiguous block of information in memory
 - Fastest: always possible to optimize this





Vector Memory Operations

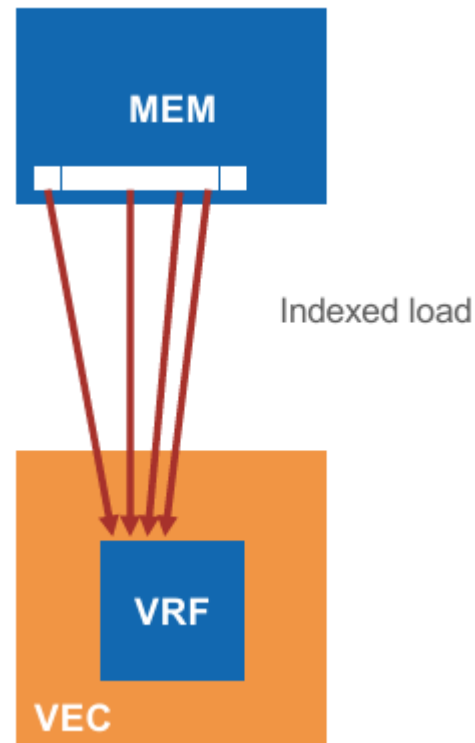
- **Three types of addressing**
 - **Non-unit (constant) stride**
 - Hard to optimize memory system for all possible stride
 - Prime number of data banks makes it easier to support different strides at full bandwidth





Vector Memory Operations

- **Three types of addressing**
 - **Indexed (gather-scatter)**
 - Vector equivalent of register indirect
 - Good for sparse arrays of data
 - Increases number of programs that vectorize





Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

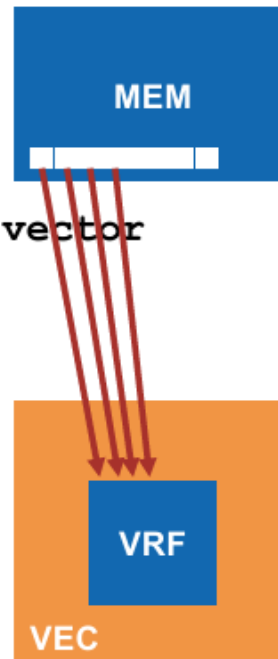
```
for (i=0; i<N; i++)  
    A[i] = B[i] + C[D[i]]
```

`vle32.v v4, (a1)`

Unit-stride load

Load indices in D vector

`a1` is a scalar register and contains
the memory address of `D`





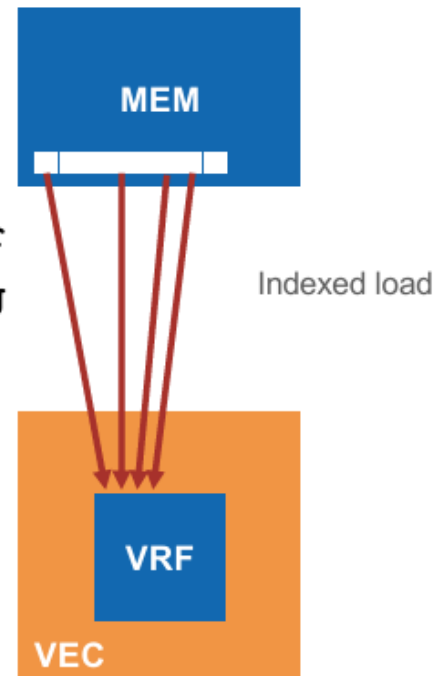
Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)  
    A[i] = B[i] + C[D[i]]
```

- Indexed load instruction (Gather) [RVV]

```
vle32.v  v4, (a1)    # Load indices in D vector  
vsll.vi  v4, v4, 2    # index to byte addressing  
vluxei32.v v3, (a2), v4 # v3[i] = C[index[i]]
```





Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
```

```
    A[i] = B[i] + C[D[i]]
```

- Indexed load instruction (Gather) [RVV]

```
vle32.v  v4, (a1)      # Load indices in D vector
vsll.vi  v4, v4, 2      # index to byte addressing (conversion)
vluxei32.v v3, (a2), v4 # v3[i] = C[index[i]]
vle32.v  v2, (a3)      # v2 = B[i]
vadd.vv  v1, v2, v3     # v1 = B[i] + C[index[i]]
vse32.v  v1, (a4)      # store A[i]
```



Vector Conditional Execution

- **Problem:** Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)  
    if (A[i]>B[i]) then  
        C[i] = A[i] + B[i];
```

- **Solution:** Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

- ...and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

vA	10	34	52	8	29	72	94	35
vB	9	66	37	4	43	83	6	76
vM	1	0	1	1	0	0	1	0
vC	19		89	12			100	

- **Code example (RVV):**

- `vle32.v v1, (a0)` # Load entire vector A
- `vle32.v v2, (a1)` # load entire vector B
- `vmslt.vv v0, v2, v1` # Set bits in mask register where A>B
- # Note: In RVV, mask register is v0
- `vadd.vv v3, v1, v2, v0.t` # Add A and B under mask
- `vse32.v v3, (a2), v0.t.` # Store C back to memory under mask



Vector Reductions

- **Problem: Loop-carried dependence on reduction variables**

```
sum = 0;
```

```
for (i=0; i<N; i++)
```

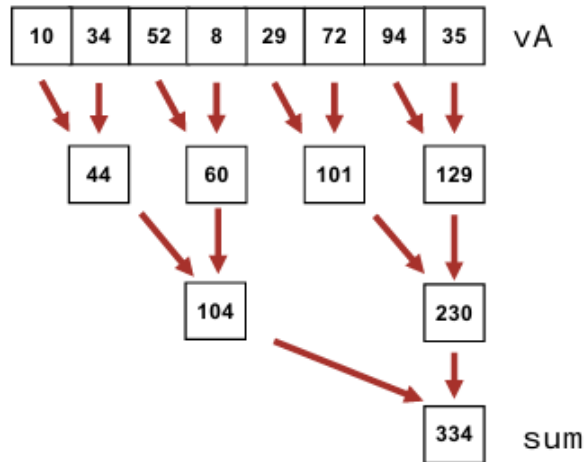
```
    sum += A[i];  # Loop-carried dependence on sum
```

- **Solution: vector reduction**

```
VLE      vA, rA
```

```
VREDSUM sum, vA
```

- **Reductions are intra-vector operations, whereas most of the other vector instructions are element-wise between two vectors**





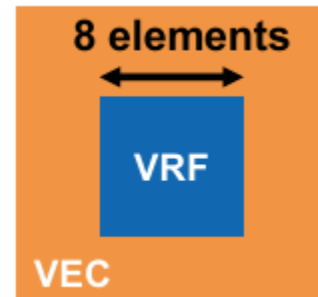
How to know the vector register length

- We can tell the vector architecture which is the length of vector (AVL) in our application/task
- If AVL is too high, the architecture sets up the max vector length
- The architecture returns back the vector length it will actually handle!
- THIS IS VISIBLE IN SOFTWARE!

```
// We have a vector with 17 elements  
avl = 17;
```

```
// We inform the vector architecture and get back the  
// length of the vector that will be processed (vl)  
vl = vsetvl(avl)
```

(vl will be set to 8)





A Vector Length Problem

- **Problem:**

- Want to add two 143-element vectors
- Don't know the vector architecture and its VRF size
- Not able to process the whole vector in one iteration

```
// We have a vector with 143 elements  
avl = 143;
```

```
// We can query the architecture
```

```
vl = vsetvl(0)    // vl == 0
```

```
vl = vsetvl(8)    // vl == 8
```

```
vl = vsetvl(35)   // vl == 35
```

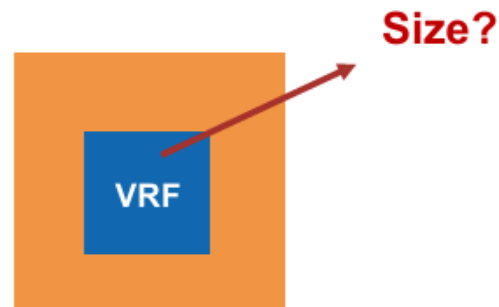
```
vl = vsetvl(63)   // vl == 63
```

```
vl = vsetvl(64)   // vl == 64 ←
```

```
vl = vsetvl(65)   // vl == 64
```

```
vl = vsetvl(143)  // vl == 64
```

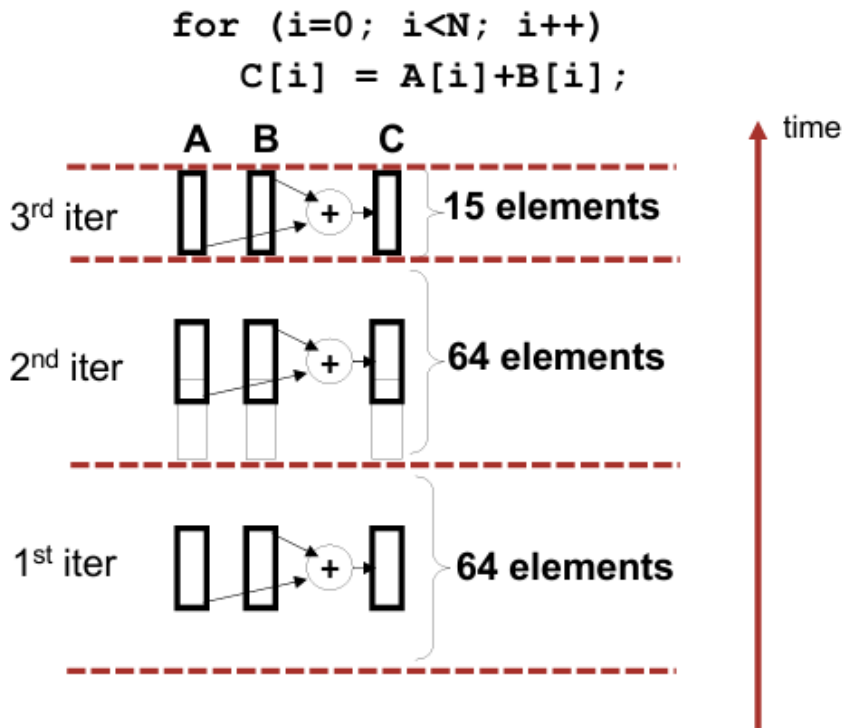
```
// We can process up to 64 elements with a vector instruction!
```





Vector Stripmining

- What if # data elements > # elements in a vector register?
- Idea: Break loops so that each iteration operates on # elements in a vector register
- E.g., 143 data elements, 64-element VRF
 - 2 iterations where VLEN = 64
 - 1 iteration where VLEN = 15
- Called vector stripmining





Vector Code is Portable

- Vector code is agnostic to the vector length ..
- ONLY IF YOUR CODE SUPPORT STRIPMINING!
- Proper vector code can work on different architecture with different VRF sizes

```
void vadd(&a, &b, &c, avl) {  
    // Strip-mine loop  
    while (avl > 0) {  
        // Set up the vector length for the current chunks  
        vl = vsetvl(avl);  
        // Load the two vector chunks  
        // Add the two vector chunks  
        // Store the vector chunk  
        // Bump pointers  
        ...  
        // Account for the chunks we have already processed!  
        avl -= vl;  
    }  
}
```

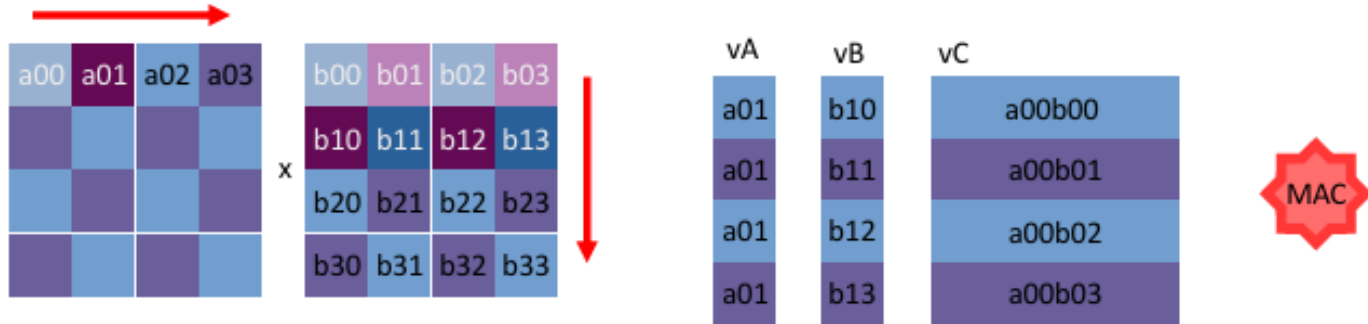
The vector length
is treated as a variable,
but should appear in
software nevertheless!





Matrix Multiplication by “Scalar Splatting”

- **Standard algorithm (row times column + reduction) is slow**
 - Highly sequential
- **Use a vector of reductions instead**





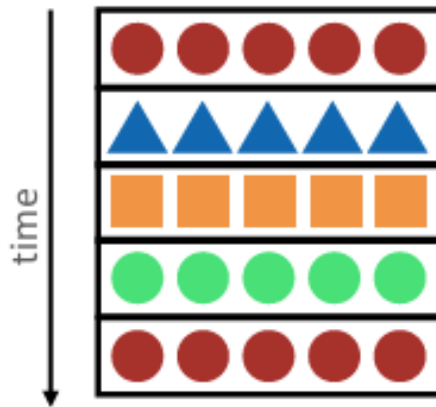
Matrix Multiplication by “Scalar Splatting”

```
# Inputs a0: pointer to matrix A (row), a1: pointer to matrix B (column)
# a2: pointer to matrix C (accumulator), a3: Matrix dimension (k)
# Set vector length to 32-bit floats, LMUL=1
vsetvli t0, a3, e32, m1, ta, ma    # Set vl to k, 32-bit, m1
# Load vector from A (row)
vle32.v v1, (a0)                  # v1 = A[i, 0...vl-1]
# Load scalar from B (single element)
flw ft0, 0(a1)                    # Load B[0, j] to floating point register
# Vector-Scalar Multiply-Accumulate: v0 = v0 + (v1 * ft0)
# (Here v0 is initialized to 0 and acts as accumulator)
vfmacc.vf v0, ft0, v1              # v0 += B[0,j] * A[i, 0...vl-1]
# Store results
vse32.v v0, (a2)
```



Vector Processors

- **Exploiting the DLP of modern workloads**
 - Amortizing the instruction fetch and decode energy cost over many cycles of computation
- **Embedded vector machine for DLP**
 - Arm Helium/MVE (M-Profile Vector Extension) on the Cortex-M cores
 - RISC-V Zve32x and Zve64x subsets of the Vector Extension (RVV)





Takeaway Questions

- What is the purpose of im2col ?
 - (A) Convert CONV to GEMM
 - (B) Increase the cache hit rate
 - (C) Reduce the data movement between remote memory
- What are benefits of Fused MAC-LOAD SIMD Instructions?
 - (A) Reduce the cost of LOADs
 - (B) Improve the L1 cache hit rate
 - (C) Mitigate the pressure of GP-RF on the processor



Takeaway Questions

- How to realize register bypass on the vector processor?
 - (A) Vector scatter-gather
 - (B) Vector chaining
 - (C) Vector stripmining