RISC-V Vector Extension Webinar IV

September 29th, 2021

Thang Tran, Ph.D.
Principal Engineer

Webinar IV - Agenda

- Andes overview
- Vector technology background
  - SIMD/vector concept
  - Vector processor basic
- RISC-V V extension ISA
  - Basic
  - CSR
- RISC-V V extension ISA
  - Memory operations
  - Compute instructions
- **Sample codes**
  - **Matrix multiplication**
  - **Loads with RVV versions 0.8 and 1.0**
- **AndesCore™ NX27V introduction**
- **Summary**

Terminology

- ISA: Instruction Set Architecture
- GOPS: Giga Operations Per Second
- GFLOPS: Giga Floating-Point OPS
- **XRF**: Integer register file
- FRF: Floating-point register file
- **VRF**: Vector register file
- SIMD: Single Instruction Multiple Data
- MMX: Multi Media Extension
- SSE: Streaming SIMD Extension
- AVX: Advanced Vector Extension
- ACE: Andes Custom Extension
- CSR: Control and Status Register
- **SEW**: Element Width (8-64)
- ELEN: Largest Element Width (32 or 64)
- **XLEN**: Scalar register length in bits (64)
- FLEN: FP register length in bits (16-64)
- **VLEN**: Vector register length in bits (128-512)
- **LMUL**: Register grouping multiple (1/8-8)
- EMUL: Effective LMUL
- **VLMAX**/MVL: Vector Length Max
- AVL/**VL**: Application Vector Length
- **Configurable**: parameters are fixed at built time, i.e. cache size
- **Extensible**: added instructions to ISA includes custom instructions to be added by customer
- **Standard extension**: the reserved codes in the ISA for special purposes, i.e. FP, DSP, …
- **Programmable**: parameters can be dynamically changed in the program

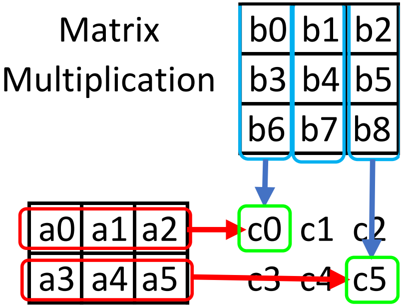
Sample Codes

Matrix Multiplication – Issue #495 in RVV Extension Work Group

```
# void matmulFloat_v(
#     size_t      n,                // a0: A columns, B rows
#     size_t      m,                // a1: A rows
#     size_t      p,                // a2: B columns
#     const float *A,               // a3: A is m x n matrix
#     const float *B,               // a4: B is n x p matrix
#     float       *C                // a5: C is m x p matrix
# ) {
#     int i, j, k;
#
#     for (i = 0; i < m; i++) {
#        for (j = 0; j < p; j++) {
#           float c = 0;
#           for (k = 0; k < n; k++) {
#              c += A[i*n+k] * B[k*p+j];
#           }
#           C[i*p+j] = c;
#        }
#     }
# }
```

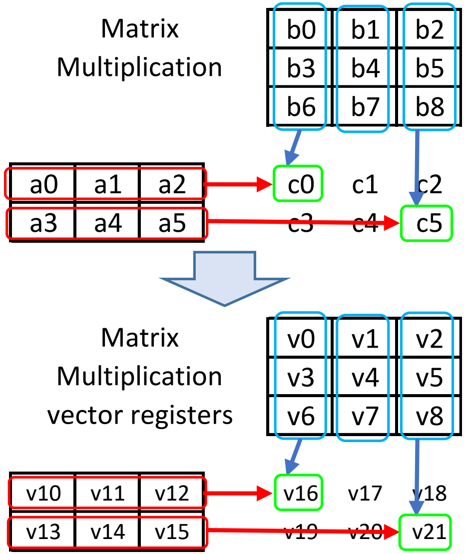
```
# void matmulFloat_v(
#     size_t      3,                // a0: A columns, B rows
#     size_t      2,                // a1: A rows
#     size_t      3,                // a2: B columns
#     const float *A,               // a3: A is 2x3 matrix
#     const float *B,               // a4: B is 3x3 matrix
#     float       *C                // a5: C is 2x3 matrix
# ) {
#     int i, j, k;
#   for (h = 0; h< COUNT; h++) {
#     for (i = 0; i < 2; i++) {
#        for (j = 0; j < 3; j++) {
#           float c = 0;
#           for (k = 0; k < 3; k++) {
#              c += A[i*n+k] * B[k*p+j];
#           }
#           C[i*p+j] = c;
#        }
#     }
#   }
# }
```

Matrix Multiplication



Matrix Multiplication – Scalar Coding Style

22 vector registers are used for 22 elements of the matrices



C	A	B
0	0	0
0	1	3
0	2	6
1	0	1
1	1	4
1	2	7
2	0	2
2	1	5
2	2	8
3	3	0
3	4	3
3	5	6
4	3	1
4	4	4
4	5	7
5	3	2
5	4	5
5	5	8

```
c0=a0*b0      vfmul.vv v16, v10, v0
c1=a0*b1      vfmul.vv v17, v10, v1
c2=a0*b2      vfmul.vv v18, v10, v2
c3=a3*b0      vfmul.vv v19, v13, v0
c4=a3*b1      vfmul.vv v20, v13, v1
c5=a3*b2      vfmul.vv v21, v13, v2
c0+=a1*b3     vfmacc.vv v16, v11, v3
c1+=a1*b4     vfmacc.vv v17, v11, v4
c2+=a1*b5     vfmacc.vv v18, v11, v5
c3+=a4*b3     vfmacc.vv v19, v14, v3
c4+=a4*b4     vfmacc.vv v20, v14, v4
c5+=a4*b5     vfmacc.vv v21, v14, v5
c0+=a2*b6     vfmacc.vv v16, v12, v6
c1+=a2*b7     vfmacc.vv v17, v12, v7
c2+=a2*b8     vfmacc.vv v18, v12, v8
c3+=a5*b6     vfmacc.vv v19, v15, v6
c4+=a5*b7     vfmacc.vv v20, v15, v7
c5+=a5*b8     vfmacc.vv v21, v15, v8
```

Each vector register holds 16 elements, 16 sets of matrices are operated in parallel

Vector Scalar-Style Coding

VLEN=512b, SEW=32b, VLEN/SEW = 512b/32b = 16 elements

- Single precision floating point data
- 16 sets of matrices can be processed in parallel
- Vector multiplication and MACC (6 vfmul and 12 vfmac)

Assuming 16 matrices are contiguous in memory, prefetched in L1 data cache (data cache hit) – stride between the matrices

- 6 stride loads for 16 A matrices (2x3=6), stride=4Bx6=24
- 9 stride loads for 16 B matrices (3x3=9), stride=4Bx9=36
- 6 stride store for 16 C matrices (2x3=6), stride=4Bx6=24

Stride Loads & Stores for Scalar-Style Code

```
vlse32.v v10, x2, x5        Aref 0
addi x2, x2, 4
vlse32.v v11, x2, x5        Aref 1
addi x2, x2, 4
vlse32.v v12, x2, x5        Aref 2
addi x2, x2, 4
vlse32.v v13, x2, x5        Aref 3
addi x2, x2, 4
vlse32.v v14, x2, x5        Aref 4
addi x2, x2, 4
vlse32.v v15, x2, x5        Aref 5
addi x2, x2, 364
```

```
vlse32.v v0, x3, x6         Bref 0
addi x3, x3, 4
vlse32.v v1, x3, x6         Bref 1
addi x3, x3, 4
vlse32.v v2, x3, x6         Bref 2
addi x3, x3, 4
vlse32.v v3, x3, x6         Bref 3
addi x3, x3, 4
vlse32.v v4, x3, x6         Bref 4
addi x3, x3, 4
vlse32.v v5, x3, x6         Bref 5
addi x3, x3, 4
vlse32.v v6, x3, x6         Bref 6
addi x3, x3, 4
vlse32.v v7, x3, x6         Bref 7
addi x3, x3, 4
vlse32.v v8, x3, x6         Bref 8
addi x3, x3, 540
```

```
vsse32.v v16, x4, x5        Cref 0
addi x4, x4, 4
vsse32.v v17, x4, x5        Cref 1
addi x4, x4, 4
vsse32.v v18, x4, x5        Cref 2
addi x4, x4, 4
vsse32.v v19, x4, x5        Cref 3
addi x4, x4, 4
vsse32.v v20, x4, x5        Cref 4
addi x4, x4, 4
vsse32.v v21, x4, x5        Cref 5
addi x4, x4, 364
```

x5 is the base address
x2 is the next element of the A matrix

x6 is the base address
x3 is the next element of the B matrix

Performance of Scalar-Style Codes

- Load 6 Arefs = 12*6 = 72 cycles // access **12** 32B cache lines
- Load 9 Brefs = 17*9 = 153 cycles // access **17** 32B cache lines
- Store 6 Crefs = 12*6= 54 cycles // access **12** 32B cache lines
- Vector FP multiplications = 18 cycles // hidden by the store
- Total cycles per loop = 297 for 16 sets of matrices
- **Total cycles per set of matrix = 19 cycles**
- Number of vector registers used = 6+9+6 = 21 registers, **efficiency = 66%**
- Number of static instructions in the loop = 62 instructions
- **Average instructions per set of matrices = 4 instructions**
- It is 16X the performance of scalar code

Matrix Multiplication – Unit Load Coding Style

- VLEN=512b, SEW=32b, VLEN/SEW = 512b/32b = 16 elements
  - LMUL=4, 16x4 = 64 elements
  - LMUL=8, 16x8 = 128 elements

- Number of Cref or Aref per matrix = 6 elements
  - 10 sets of matrices are 60 elements which fit into VRF with LMUL=4
- Number of Bref per matrix = 9 elements
  - 10 sets of matrices are 90 elements which fit into VRF with LMUL=8

Unit Load of B elements

- Unit load of 10 consecutive B matrices (10x9=90 elements)
  - vrgather to set up 10 sets of 6 B elements = 2, 1, 0, 2, 1, 0 (yellow) in v16
  - vslideup v16 to v20
  - vrgather to set up 10 sets of 6 B elements = 5, 4, 3, 5, 4, 3 (blue) in v16
  - vrgather to set up 10 sets of 6 B elements = 8, 7, 6, 8, 7, 6 (white) in v24

setvli t1, x5, e32, m8	vl=90, LMUL=8
vle32.v v8, x3	unit load - Bref, 90 elements
addi x3, x3, 360	
setvli t1, x6, e32, m8	vl=60, LMUL=8
vsub.vi v0, v0, 6	
vrgather.vv v16, v8, v0	60 elements of first Bref in v16
vadd.vi v0, v0, 3	
vslideup.vi v16, v16, 64	Move first Bref to v20
vrgather.vv v16, v8, v0	60 elements of second Bref in v16
vadd.vi v0, v0, 3	
vrgather.vv v24, v8, v0	60 elements of third Bref in v24

C	A	B
0	0	0
0	1	3
0	2	6
1	0	1
1	1	4
1	2	7
2	0	2
2	1	5
2	2	8
3	3	0
3	4	3
3	5	6
4	3	1
4	4	4
4	5	7
5	3	2
5	4	5
5	5	8

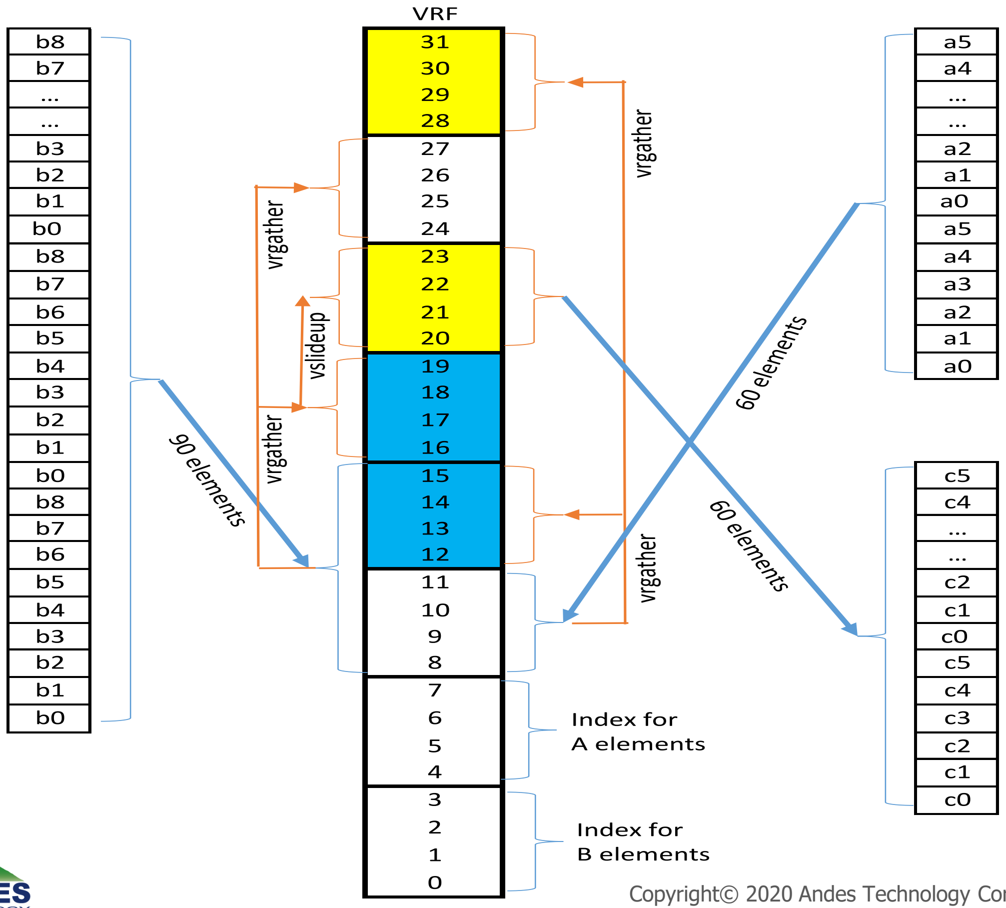
Unit Load of A elements

- Unit load of 10 consecutive A matrices (10x6=60 elements)
  - vrgather to set up 10 sets of 6 A elements = 3, 3, 3, 0, 0, 0 (yellow) in v16
  - vrgather to set up 10 sets of 6 A elements = 4, 4, 4, 1, 1, 1 (blue)
  - vrgather to set up 10 sets of 6 A elements = 5, 5, 5, 2, 2, 2 (white)
- Multiplication and accumulate and unit store C elements to memory

```
setvli t1, x7, e32, m4          vl=60, LMUL=4
vle32.v v8, x2                  unit load - Aref, 60 elements
addi x2, x2, 240
vrgather.vv v28, v8, v4         set up 60 elements of first Aref
vadd.vi v4, v4, 1
vfmul.vv v20, v20, v28
vrgather.vv v12, v8, v4         set up 60 elements, of second Aref
vadd.vi v4, v4, 1
vfmacc.vv v20, v16, v12
vrgather.vv v28, v8, v4         set up 60 elements of third Aref
vsub.vi v8, v8, 2
vfmacc.vv v20, v24, v28
vse32.v v20, x4                 unit store - Cref, 60 elements
addi x4, x4, 240
bne
```

C	A	B
0	0	0
0	1	3
0	2	6
1	0	1
1	1	4
1	2	7
2	0	2
2	1	5
2	2	8
3	3	0
3	4	3
3	5	6
4	3	1
4	4	4
4	5	7
5	3	2
5	4	5
5	5	8

Graphical View of Unit Load Codes



C	A	B
0	0	0
0	1	3
0	2	6
1	0	1
1	1	4
1	2	7
2	0	2
2	1	5
2	2	8
3	3	0
3	4	3
3	5	6
4	3	1
4	4	4
4	5	7
5	3	2
5	4	5
5	5	8

Performance of Unit Load Codes

- Load 60 Aref and 90 Bref elements, store 60 Cref elements = 28 cycles of 32B cache lines
  - Vrgather for Aref = 8 cycles + 4 + 4 = 16 cycles
  - Vrgather/vslideup for Bref = 13 cycles + 8 + 13 + 13 = 47 cycles
- Vector FP multiplications = 12 cycles
- Total cycles per loop = 103 for 10 sets of matrices
- **Total cycles per set of matrix = 10 cycles** (19 cycles for straight code)
- Number of static instructions in the loop = 27 instructions
- **Average instructions per set of matrices = 2.7 instructions** (4 instructions for straight code)
- Number of elements used in VLMAX, 60/64 and 90/128, **efficiency is about 90%**
- Unit load/store are much more efficient memory operation than stride load: accessing 28 instead of 41 cache lines. Shifting the complexity of stride load/store to vrgather instructions

Larger Matrices – Scalar Style Coding

- Same VLEN, SEW=32b, VLMAX=16
  - A matrix 6x3, 18 elements – requires **18** vector registers
  - B matrix 3x6, 18 elements – requires **18** vector registers
  - C matrix 6x6, 36 elements – requires **36** vector registers
  - Compute 1 row of A and C elements at a time, 6 iterations to finish 16 sets of matrices

Matrix Multiplication

b0	b1	b2	b3	b4	b5
b6	b7	b8	b9	b10	b11
b12	b13	b14	b15	b16	b17

a0	a1	a2
a3	a4	a5
a6	a7	a8
a9	a10	a11
a12	a13	a14
a15	a16	a17

c0	c1	c2	c3	c4	c5
c6	c7	c8	c9	c10	c11
c12	c13	c14	c15	c16	c17
c18	c19	c20	c21	c22	c23
c24	c25	c26	c27	c28	c29
c30	c31	c32	c33	c34	c35

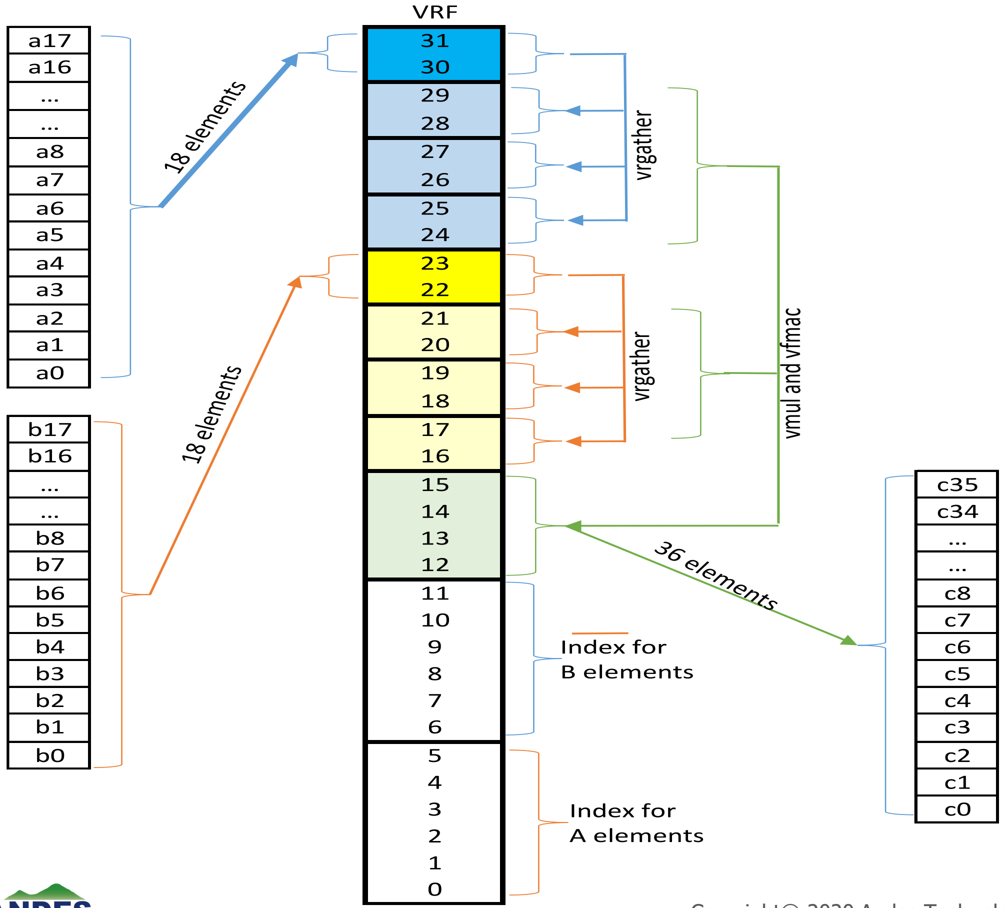
Matrix Multiplication vector registers

v0	v1	v2	v3	v4	v5
v6	v7	v8	v9	v10	v11
v12	v13	v14	v15	v16	v17

v18	v19	v20
v18	v19	v20
v18	v19	v20
v18	v19	v20
v18	v19	v20
v18	v19	v20

v21	v22	v23	v24	v25	v26
v21	v22	v23	v24	v25	v26
v21	v22	v23	v24	v25	v26
v21	v22	v23	v24	v25	v26
v21	v22	v23	v24	v25	v26
v21	v22	v23	v24	v25	v26

Larger Matrices – Unit Load Codes



- VLEN=512b, SEW=32b, VLMAX=16 elements
  - LMUL=2, 32 elements, 2 vector registers are used for each 18 elements of Aref and Bref
  - LMUL=4, 64 elements, 4 vector registers are used for 36 elements of Cref
  - 1 set of matrices per loop iteration

Performances of Larger Matrices

	Scalar style	Unit load
Number of static instructions per loop	252	30
Total number of cycles	1259	22*
Number of cycles per set of matrices	79	22
Efficiency of register usages	81%	56%**

*Vector loads/stores are pipelined between iterations

**6x4 or 6x5 matrices are more efficient. 6x3 has 18 active elements with VLMAX=32

Unit loads/stores are much more efficient memory operation than stride load

RVV Work Group, Issue #362, Codes Based on RVV 0.8

```
vsetvli t0, zero, e32, m8     // VLEN=512b, LMUL=8, SEW=32b, ELEN=4096b
vlb.v v0, t0;   (RVV 0.8)
add t0, t0, imm;
vadd.vx v0, v0, t1;
lfw f0, t2;
add t2, t2, imm;
vfcvt.f.x.v v0, v0;
vfmacc.vf v8, v0, f0;
bne t0, t3, imm
```

Loop of 8 instructions,
Load element = 8b INT, extension to 32b
Operation on FP 32b

- Performance requirement: 16B of 8b INT every clock cycle

Vector Processor Pipeline Example

VLEN=64B ← 8 cycles → ← 8 cycles → ← 8 cycles →

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
vadd v0, v0, t1	va								va								va							
vadd v1, v1, t1		va								va								va						
vadd v2, v2, t1			va								va								va					
vadd v3, v3, t1				va								va								va				
vadd v4, v4, t1					va								va								va			
vadd v5, v5, t1						va								va								va		
vadd v6, v6, t1							va								va								va	
vadd v7, v7, t1								va								va								va
vfcvt v0, v0				fc	fc							fc	fc							fc	fc			
vfcvt v1, v1					fc	fc							fc	fc							fc	fc		
vfcvt v2, v2						fc	fc							fc	fc							fc	fc	
vfcvt v3, v3							fc	fc							fc	fc							fc	fc
vfcvt v4, v4								fc	fc							fc	fc							fc
vfcvt v5, v5									fc	fc							fc	fc						
vfcvt v6, v6										fc	fc							fc	fc					
vfcvt v7, v7											fc	fc							fc	fc				
vfmacc v8, v0, f0						fm	fm	fm						fm	fm	fm						fm	fm	fm
vfmacc v9, v1, f0							fm	fm	fm						fm	fm	fm						fm	fm
vfmacc v10, v2, f0								fm	fm	fm						fm	fm	fm						fm
vfmacc v11, v3, f0									fm	fm	fm						fm	fm	fm					
vfmacc v12, v4, f0										fm	fm	fm						fm	fm	fm				
vfmacc v13, v5, f0											fm	fm	fm						fm	fm	fm			
vfmacc v14, v6, f0												fm	fm	fm						fm	fm	fm		
vfmacc v15, v7, f0													fm	fm	fm						fm	fm	fm	
vlb v0, t0	v0	v1	v2	v3	v4	v5	v6	v7	v0	v1	v2	v3	v4	v5	v6	v7	v0	v1	v2	v3	v4	v5	v6	v7
lfw f0, t2								f0								f0								f0

chaining

chaining

Vector Processor Performance

- Number of instructions and micro-ops in 8 cycles
  - 3 vector arithmetic instructions = 24 micro-ops (vadd, vfcvt, vfmac)
  - 1 FP load
  - 3 scalar instructions + 1 vector load to data cache
  - Performance at 1 GHz = 64 GOPS or 48 GFLOPS for SEW=32b
  - Performance at 1 GHz = 128 GOPS or 96 GFLOPS for SEW=16b
- Unroll the loop twice in order to pipeline vector load instructions
  - All 32 vector registers are used
  - Performance improvement can be achieved with VLEN=1024b

RVV Work Group, Issue #362, Codes Based on RVV 0.9

```
vle8.v v6, t0;
add t0, t0, imm;
vsext.v4  v0, v6
vadd.vx v0, v0, t1;
lfw f0, t2;
add t2, t2, imm;
vfcvt.f.x.v v0, v0;
vfmacc.vf v8, v0, f0;
bne t0, t3, imm
```

Loop of 9 instructions,
Load element = 8b INT,
Vector extension to 32b,
Operation on FP 32b

- An extra instruction in a tight loop can degrade the performance by 12.5%
- Andes provides custom load instruction in order to achieve the required performance
- In addition, the vsext.v4 instruction needs an extra write port since all the write ports are used

AndesCore™ NX27V introduction

Andes NX27V Design Objectives

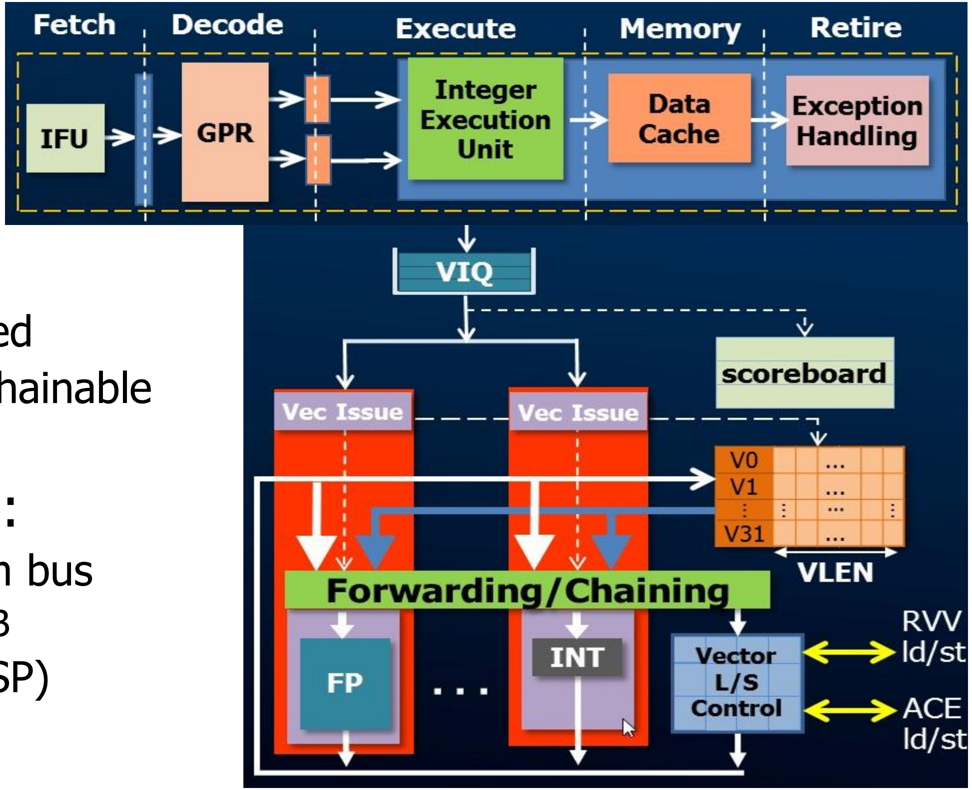
- **Vector Processor is implemented as part of the RISC-V CPU**

- **Configurability:** necessary for IP to attract wide range of customers, 128b to 512b
- **Scalability:** necessary for next generation
- **Low power and area:** necessary for embedded market
- **Performance:** out-of-order execution and design
- **Simplicity:** necessary for time to market, reducing verification time
- **Modular Design:** decoupling for independent design, verification, timing, and clock gating

Love this quote: “**Simplicity** is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.” — Edsger W. Dijkstra

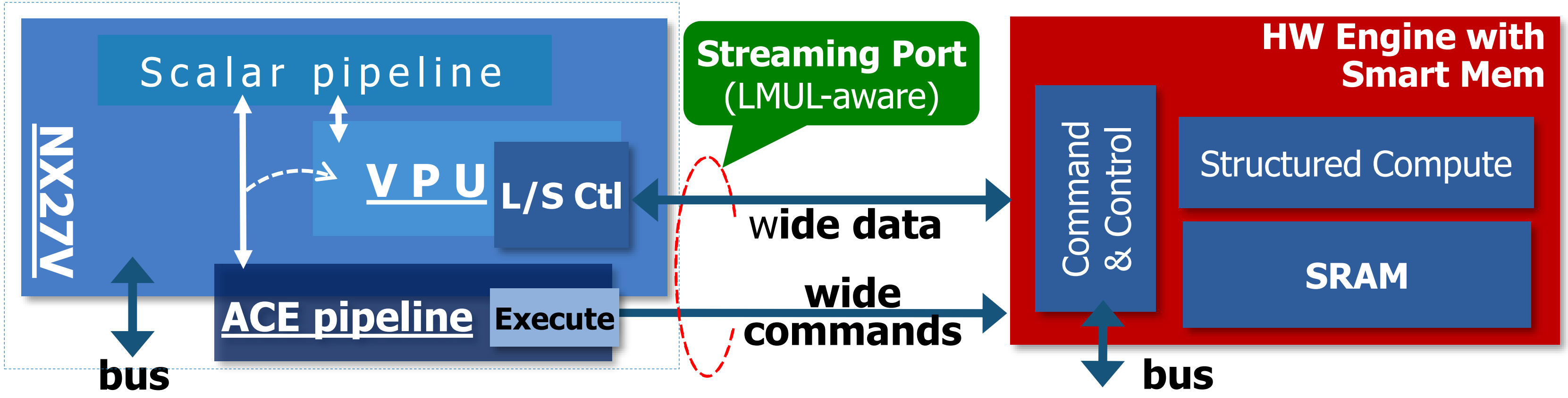
NX27V: Powerful Vector Processor

- An efficient Scalar Unit
- A powerful decoupled OOO VPU
  - Data formats or SEW
    - Standard: int8/16/32/64 and fp16/32/64
    - Andes extended: bfloat16, and int4
  - RVV instructions start execution after retired
  - Functional units: parallel, most pipelined/chainable
  - VLEN & SIMD_MEM width: 128, 256, 512
- Independent memory access paths:
  - RVV load/store through dcache and system bus
    - Configurable L1 cache memory 32KB-512KB
  - ACE load/store through Streaming Port (ASP)



NX27V: ACE Streaming Port (ASP)

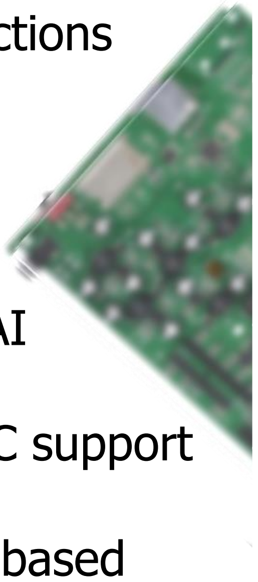
- **A common scenario in SoC with HW accelerators:**
  - DMA-equipped HW engine accelerates regular/structured computations (e.g. CNN, ME, FFT)
  - Efficient & programmable pre/post-processing is needed → algorithm innovation/evolution



- Solution: ASP to tightly couple NX27V and HW engine as a PE
  - Wide ASP buses allow efficient command dispatch and data transfer to/from HW engine
  - ASP load/store instructions to/from V/F/X registers with custom addressing mode
  - Leverage NX27V vector processing for fast and flexible data pre/post-processing

RVV Tools and Performance

- Standard tools in AndeSight™
  - AndeSim™: Cycle simulator
  - Compiler: intrinsic functions
  - Assembler
  - Debugger and ICE
  - Computation libraries
- Advanced tools:
  - **LLVM** IR support for AI compilers
  - **AndeSysC**™: SystemC support for AndeSim™
  - **AndesClarity**™: GUI-based pipeline visualizer and analyzer



RVV Functions	NX27V Speedup[1]
F32 basic function	**23x**
F32 32x32 GEMM	**43x**
Q7 CNN HWC[2] (33,33,51)	**35x**
Q7 Relu CNN	**81x**

Note 1: Compared to pure C scalar code compiled with high optimization; both vector and scalar code ran on the NX27V FPGA with 512/256-bit VLEN/SIMD, 256-bit bus.
Note 2: HWC(Height, Width, Channel)

NX27V Performance Data

Features	NX27V		
VLEN/SIMD	256/256	512/256	512/512
ELEN (bit)	32 (int+fp)		
Max Frequency (worst case)[1]	All frequencies >=1.2G		
Gate Count (gates)[1]	From 1.6M to 2.6M		
Dynamic Power (uW/MHz)[1]	< 17		
F32 GEMM 32X32 (cycle)	Around 6,200	Around 4,800	Around 3,400

Note 1: 7nm **mixed VT** 240H library, **V-extension, 128-entry BTB, 16-entry PMP/PMA and 32KB I/D$, AXI BUS** with I/O constraint, power are core only. Frequency condition: 0.675v/-40°c, SS; Dynamic power condition: 0.75v/25°c, TT, Dhrystone program, logic synthesis

Andes NX27V vs. Competitions

	Andes NX27V	CAxx	CMxx
Architecture	RVV/Andes VPU	Popular SIMD	Hxxx
Vector registers	32	32	8
Vector Length	Up to 512b	128b	128b
SIMD width	Up to 512b/cycle	128b/cycle	64b/cycle
LMUL	Yes	None	None
Chaining	Yes	Not applicable	Yes
Custom extension	ACE	No	No
Streaming Ports	Yes	No	No

Core/System Performance Comparison

CPU	A64FX*	NX27V**
Vector ISA	ARM SVE	RISC-V RVV
Technology (nm)	7	7
Core sustain Perf 16b (GFLOPS)	~56	96
Core Peak Perf 16b (GOPS)	230	320
# of cores	48	48**
System (TFLOPS)	~2.7	~4.6
Memory BW (GB/s)	1024	3072

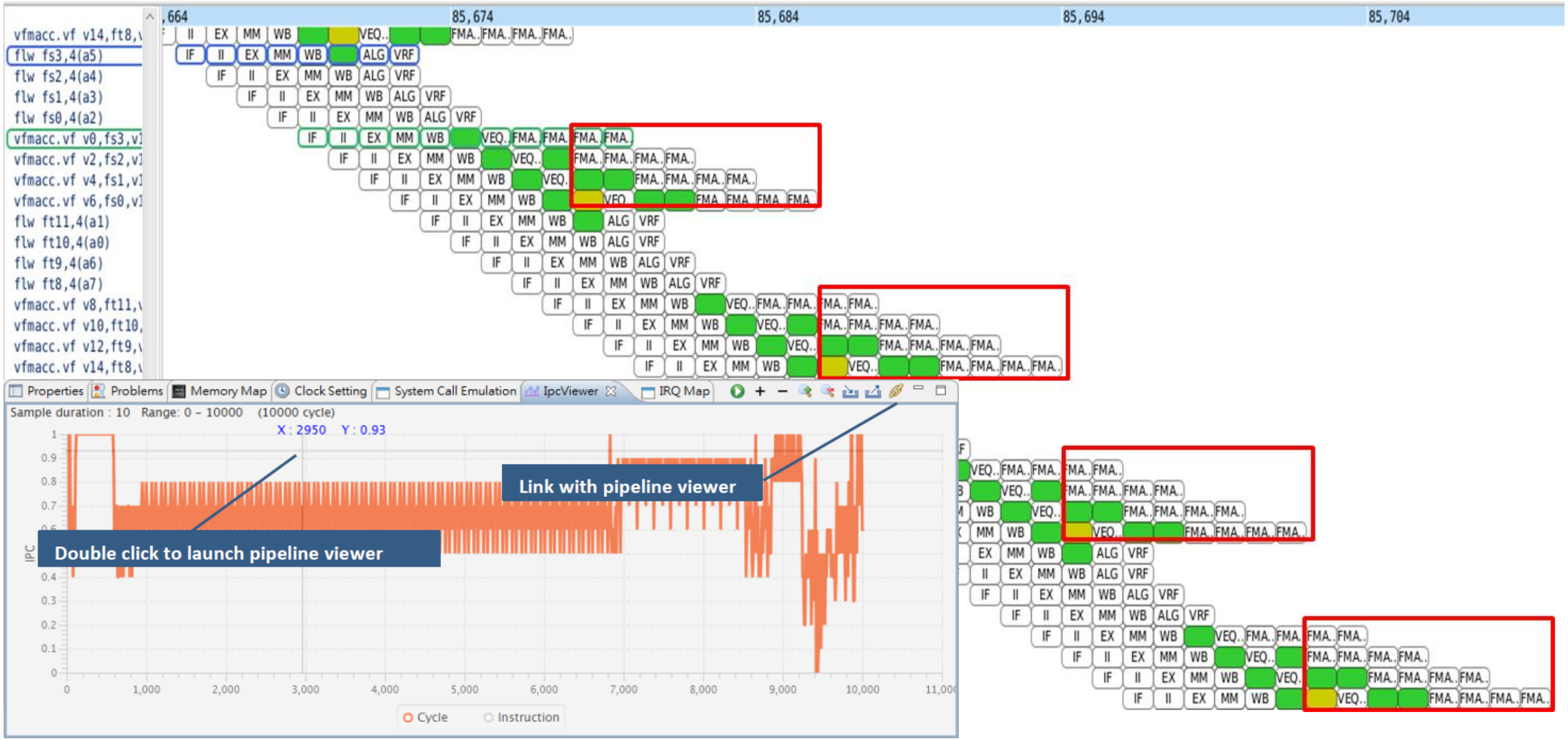
*Fujitsu presentation at Hot Chip 2018.

**Assume the same number of cores for comparison. 512-bit SIMD, 512-bit bus.

Development Tools

- Standard tools in AndeSight™ IDE:
  - AndeSim: Near-cycle accurate simulator
  - Compiler (intrinsic functions and inline assembly)
  - Assembler
  - Debugger and ICE
  - Computation library
- Advanced tools:
  - AI compiler support thru LLVM
  - AndesClarity pipeline visualizer and analyzer
    - Pipeline view of instructions and functional units
    - Resource view corresponding to instruction usage
    - Stall bubbles with different colors for data dependency

Clarity: NX27V Pipeline View



Summary

- Andes NX27V vector processor
  - The world first commercial RISC-V vector processor, VLEN = 512b/256b/128b are available (announced in December 2019)
  - Five customer projects …
  - High performance in HPC and AI applications
  - Flexible VPU configurations to enable a wide range of applications
  - AndesClarity for performance optimization
  - Work with customers on Andes Custom Extension (ACE) to provide proprietary, hardware acceleration, and marketing competitive
- Andes Technology will continue to expand vector product families to meet the requirements of edge to cloud applications

Follow Andes, Find Latest Trends

Thank you!