ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA Future Computing Laboratory

EFCL Workshop

A RISC-V Vector-Processor for High-Throughput Multidimensional Sensor Data Processing

17.08.2023

Matteo Perotti mperotti@iis.ee.ethz.ch

PULP Platform

Open Source Hardware, the way it should be!

@pulp platform

pulp-platform.org

youtube.com/pulp platform

The curious case of Vector Processors

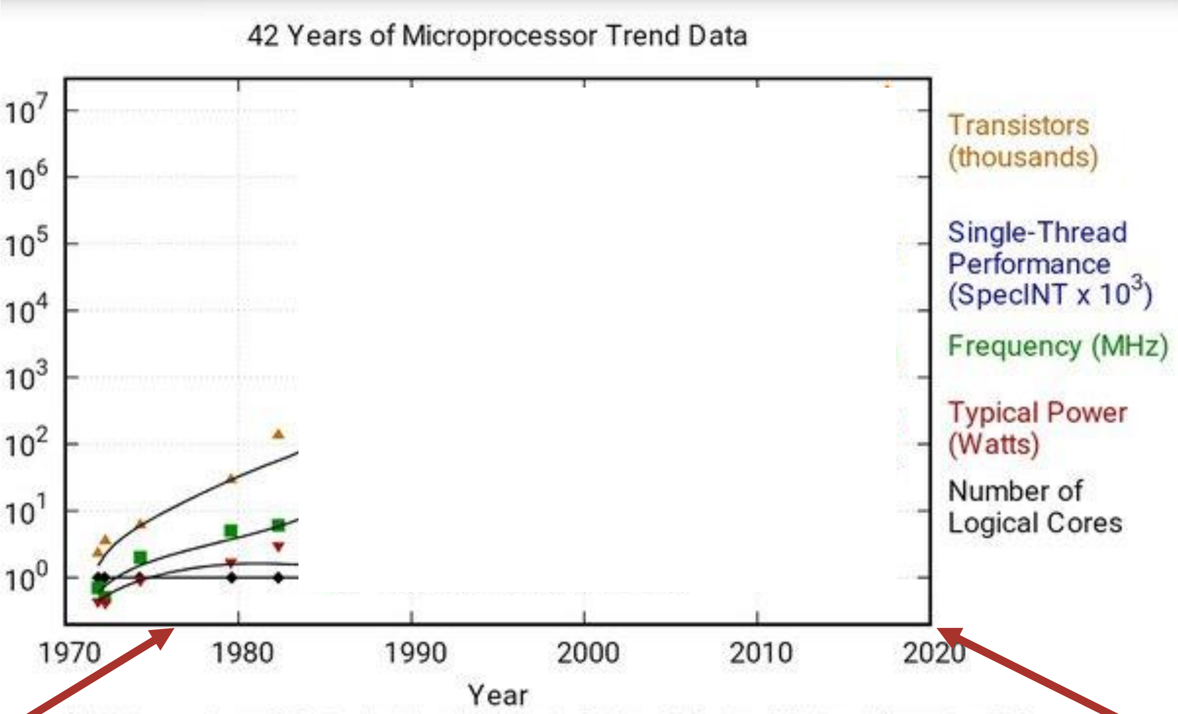
Cray X1

256 MFLOPS

Fastest supercomputer in 1976!

The Cray 1, a vector supercomputer. The first model ran at 80MHz but could retire 2 instructions/cycle for a peak of 160 MIPS. However, it could reach 250 MFLOPS using vectors.

1



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

SUPERCOMPUTER FUGAKU

500 PFLOPS

Second fastest supercomputer in 2023!

Inexpensive silicon

ILP, OoO, Super Scalar

Hype for Vector Processors

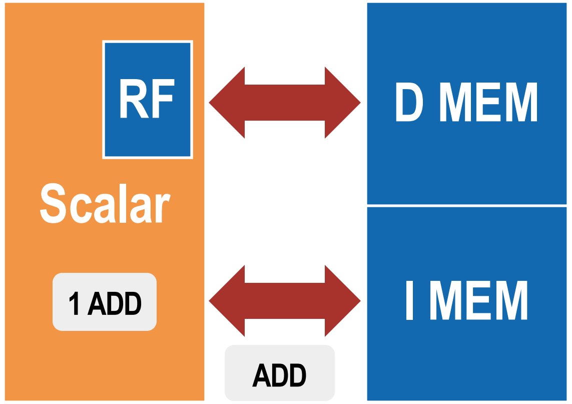
End of Dennard's Scaling

Power wall

Single-Instruction, Multiple-Data (SIMD) processor

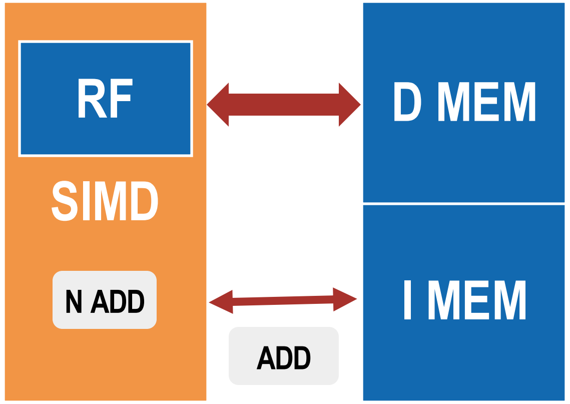
SCALAR CORE

- One instruction – One Operation
  - BW and Power on the I-MEM
- RF size is usually fixed
  - One data element per entry
  - Too small for highly-intensive WL
    - BW and Power on the D-MEM



SIMD CORE

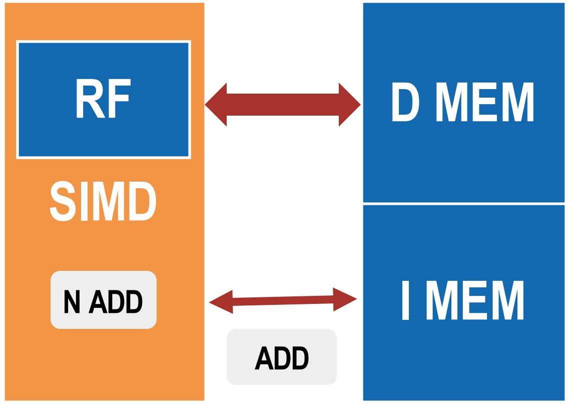
- One instruction – Multiple operations
  - Lower BW and Power on the I-MEM
- RF size is larger
  - Multiple data elements per entry
  - Better buffering, exploit locality
    - Lower BW and Power on the D-MEM



Vector Processors – A flexible and efficient choice

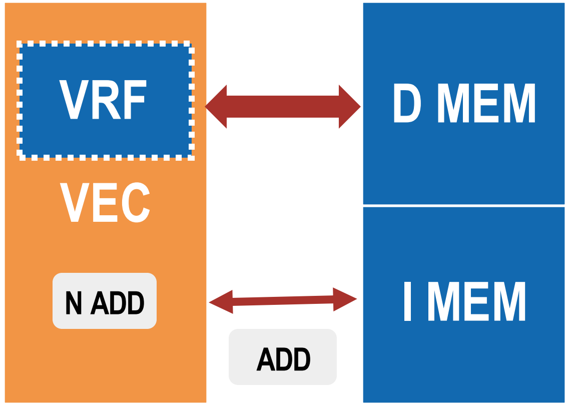
SIMD CORE

- Vector Length (VL) encoded in the instruction!
  - Support new VL? Extend the ISA!
    - e.g., add128, add256, ...
  - RF is not flexible



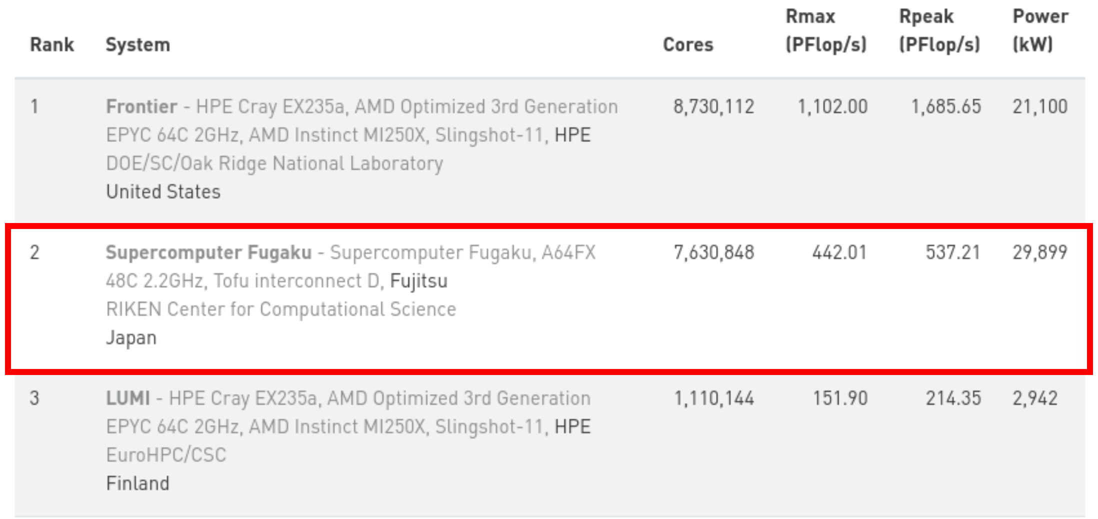
VECTOR CORE

- Programmable VL
  - VRF size decoupled from DP width



Only Fugaku?

- Intel
  - From SSE to AVX
- Arm
  - From NEON to SVE (powering FUGAKU), SVE2, Helium
- RISC-V
  - V extension



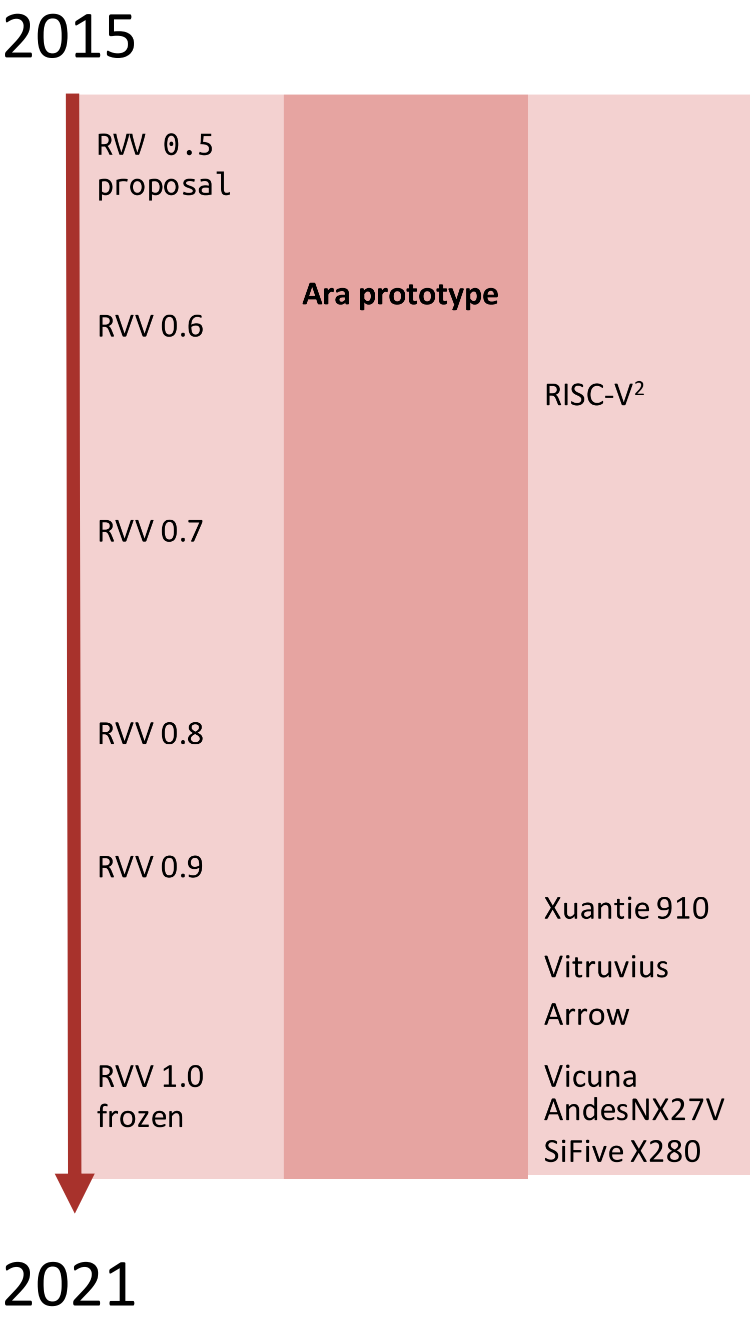
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942

[1] Fugaku supercomputer at SC19. FUJITSU Supercomputer PRIMEHPC FX1000 Rack
Raysonho @ Open Grid Scheduler / Scalable Grid Engine - Eigenes Werk
CC BY-SA 4.0

RISC-V V ISA - A long journey

- Early draft in **2015**
- Years of refinment. Frozen in end 2021.
- **Known programming model**

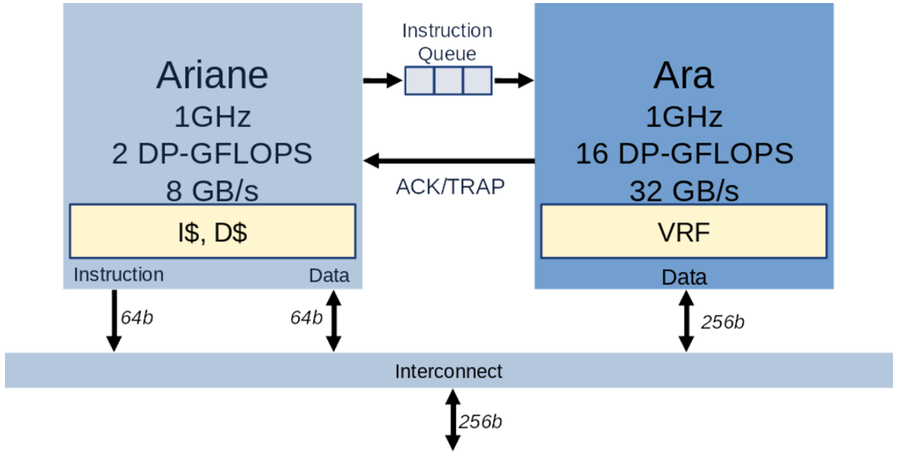
- Comprehensive ISA (+300 instructions)
  - VRF setup instructions (VL, element width, ...)
  - Arithmetic instructions (int, fixpt, fp)
  - Memory operations (unit-stride, strided, indexed)
  - Predicated execution (masks)
  - Permutation instructions (slide, reorder) **Vector length agnostic** – High reusability!



EFCL Project Overview

Duration - 24 months

- Start: 03/21
- End: 03/23



Starting point

1. **AraV1** (**RISC-V V** draft **v0.5**)
2. **Peak performance/efficiency on** matmul and conv2d for **long vectors**

Goals

1. **Expand benchmark pool** with heterogeneous applications for high-dimensional workloads
2. **Update Ara** from RISC-V V v0.5 **to** the most recent **RISC-V Vector ISA (v0.9)**
3. **Develop uArchitecture** to **improve IPC, Performance, Energy Efficiency**
4. **Address coherence problem** between Ara and Ariane
5. **HW/SW interface mixing intrinsics**, hand-**optimized macros**, **vector compilation**

EFCL Project Timeline, WPs, Milestones

Year 1

WP1: Workload analysis and benchmarking

- **Select** and **implement key** application **kernels**

WP2: uArch design and exploration of instruction extensions

- Develop **the new micro-architecture compliant with RVV 0.9 (then RVV 1.0)**
- **Evaluate HW** cost of **ISA extensions** to **speed up key** application **kernels**

Milestones: workload and benchmark analysis, version 0.1 of Ara2 with performance profiling

Year 2

WP3: uArch opt and implementation

- **Optimize and implement microarchitecture**
- **Target frequency in the GHz range**
- Emphasize **scalability** and high **energy efficiency**
- **PPA** of the resulting design

WP4: design space exploration, sw environment, open-source release

- **Verify** the **design**
- Provide **Software** support
- **Open-source release** of **HW** and **SW**

Milestones: version 1.0 of Arav2 with instruction extensions and PPA analysis, open source release of HW and SW

EFCL Project Execution

Bi-weekly meetings

Rationale for **SW selection**

- Discussed with **companies, as representative** of the target computation
- **Not too complex** to ease vectorization, analysis, and insights
- **From RiVEC RVV** benchmark suite too, to have **comparable results**
- **Stress** different **datapath units of Ara**

Project WPs were mainly **overlapped**:

- **Specs update** (to 0.9, then to 1.0) and new features **require iterations** on the physical **implementation**
- **Benchmarks** required **HW fixes** and **modifications**, which **require** new **iterations** on the implementation
- **Verification** done **throughout the project**, when adding new instructions, new benchmarks

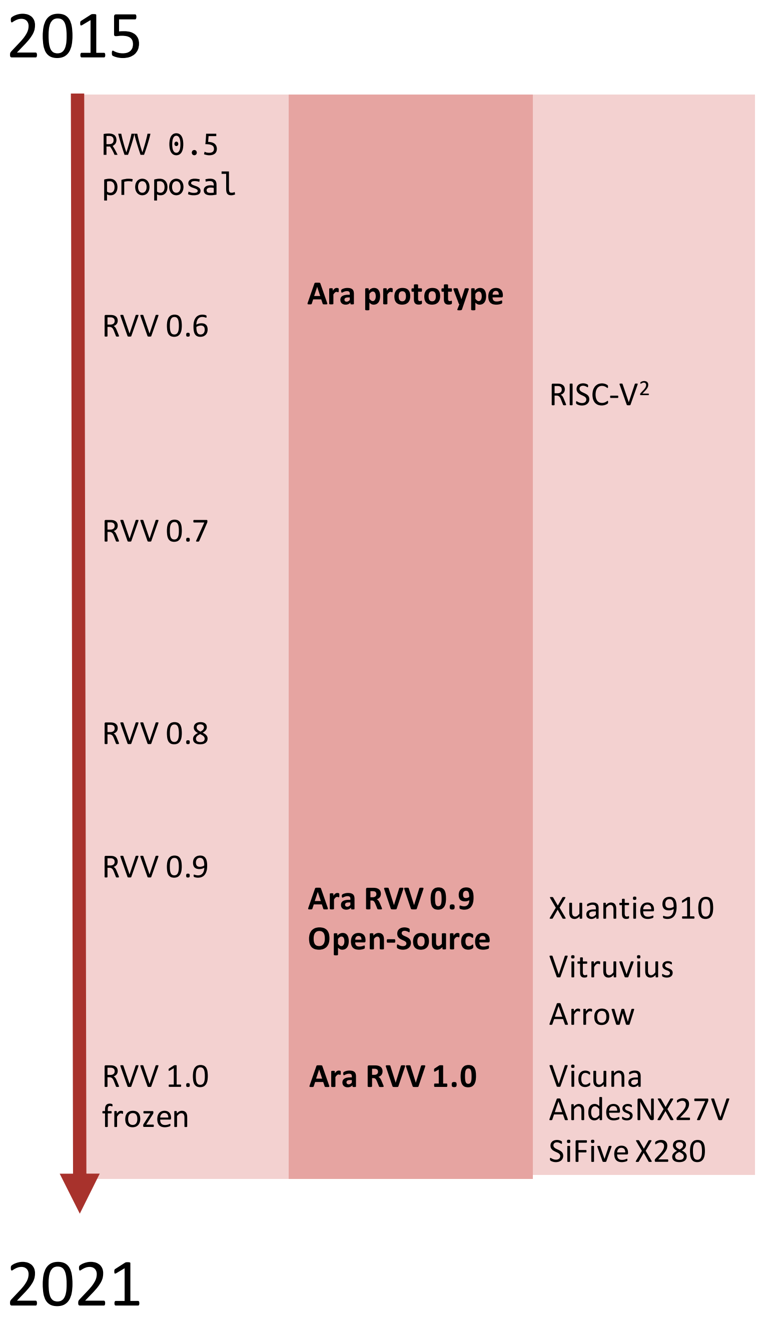
The **software ecosystem** evolved throughout the project (compilers, simulator, V intrinsics, ...)

- **Updates needed** to guarantee a high-quality open-source distribution

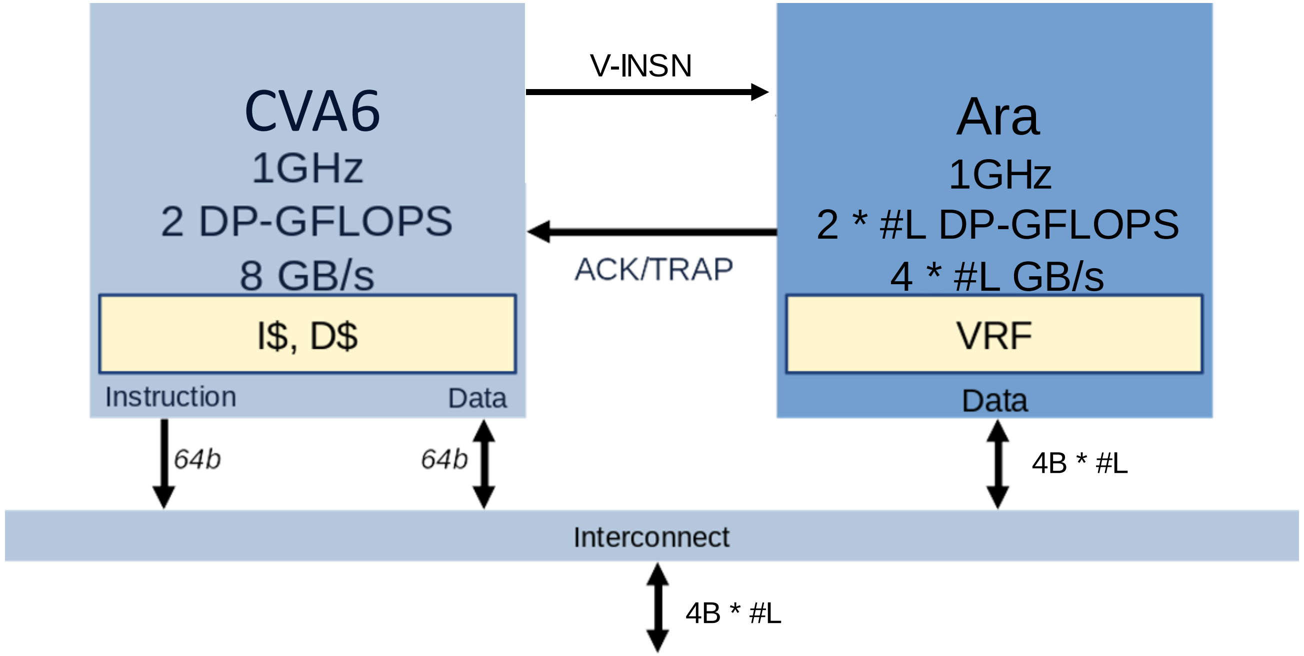
RISC-V V ISA - A long journey

- Early draft in **2015**
- Years of refinment. Frozen in end 2021.
- **Known programming model**

- Comprehensive ISA (+300 instructions)
  - VRF setup instructions (VL, element width, ...)
  - Arithmetic instructions (int, fixpt, fp)
  - Memory operations (unit-stride, strided, indexed)
  - Predicated execution (masks)
  - Permutation instructions (slide, reorder) **Vector length agnostic** – High reusability!

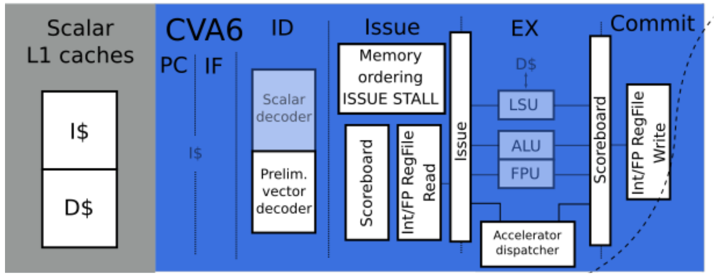


Ara System – High-Level View



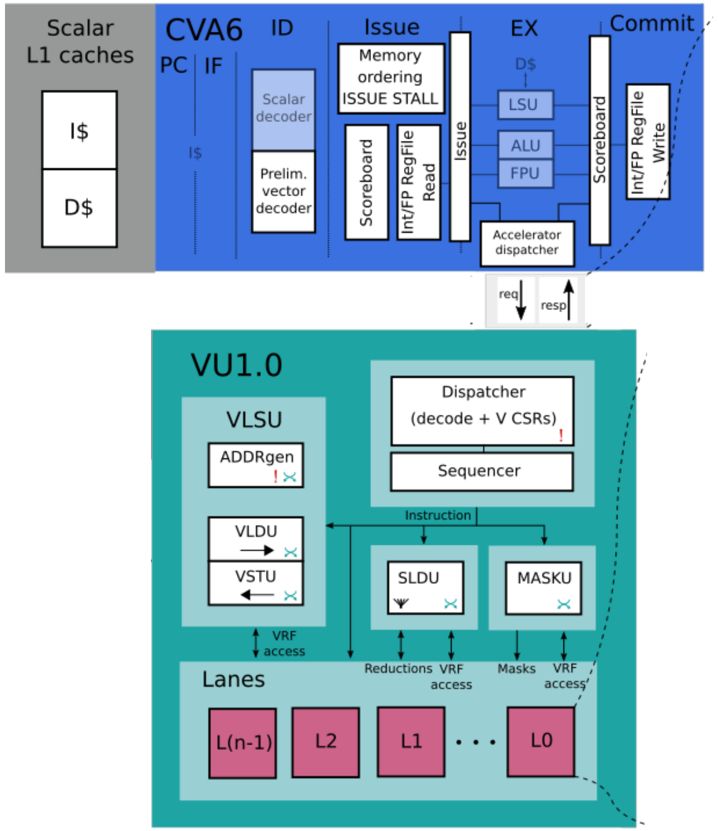
Ara System – One step below

- Scalar core: **CVA6**



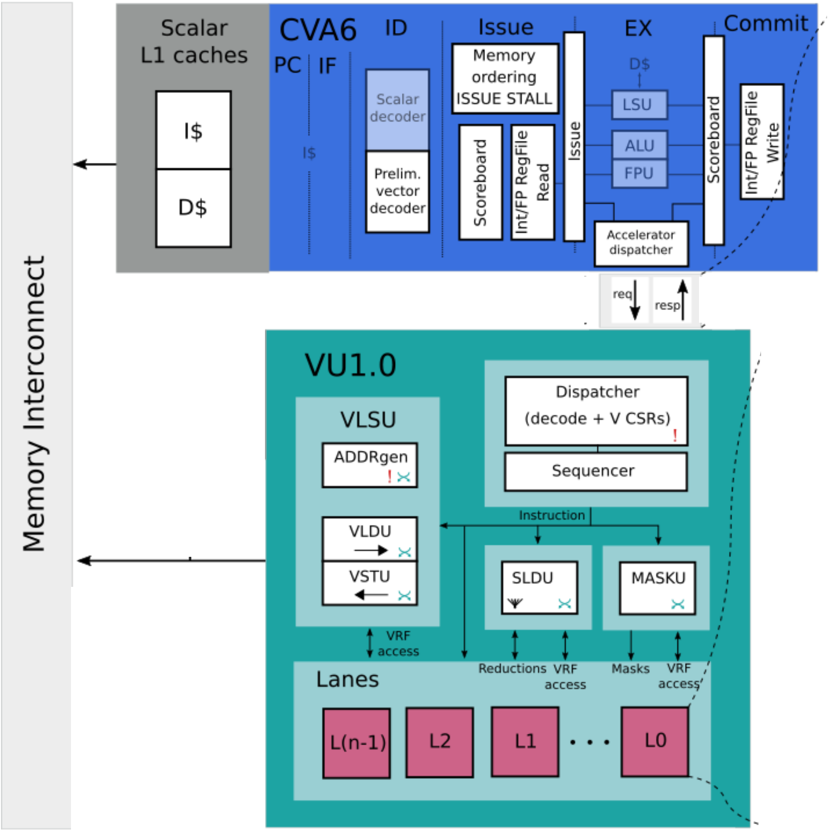
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**



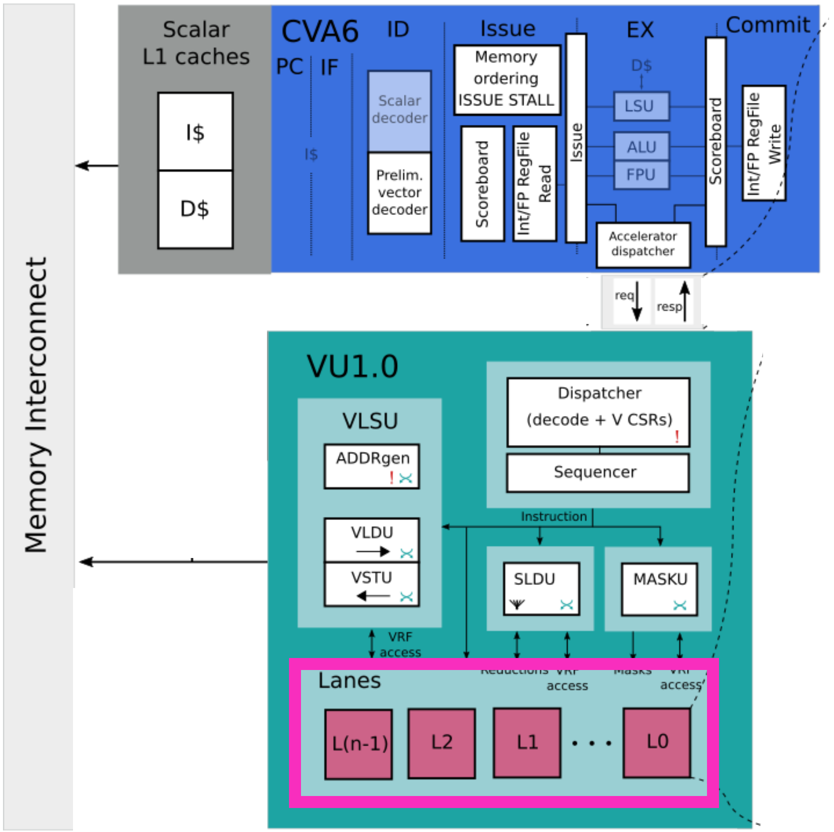
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect



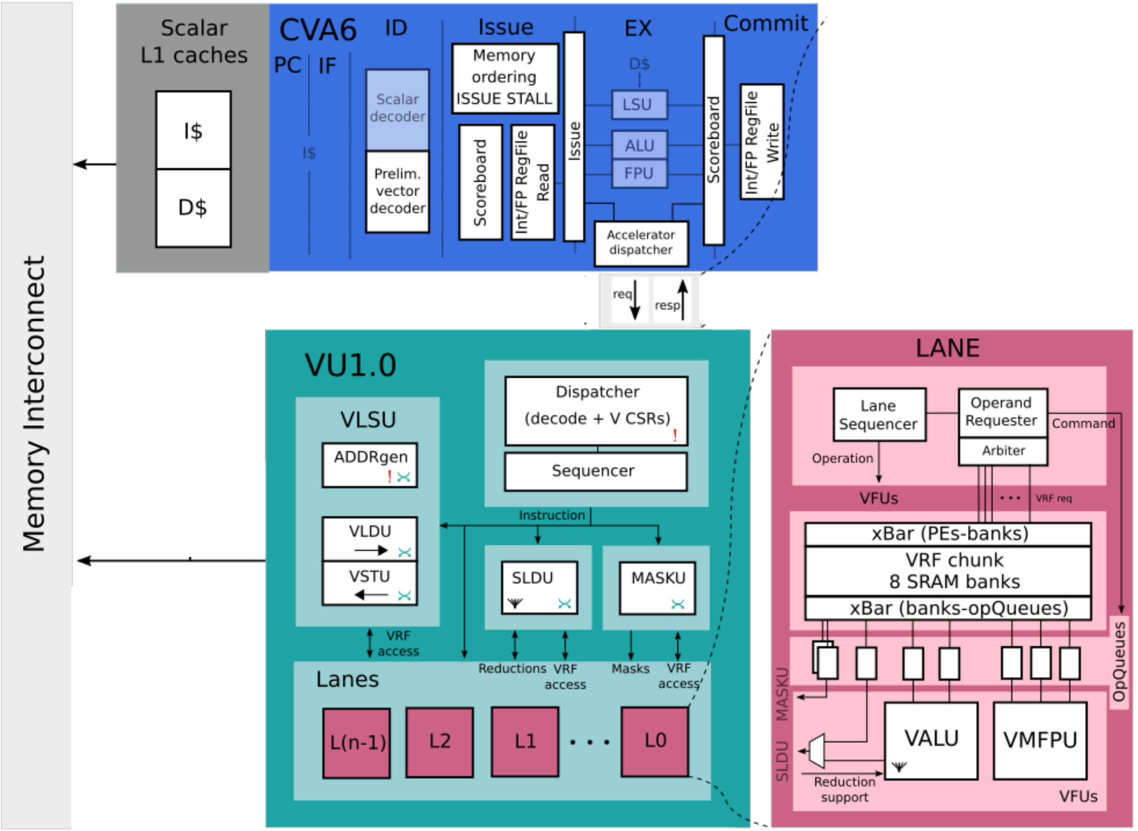
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



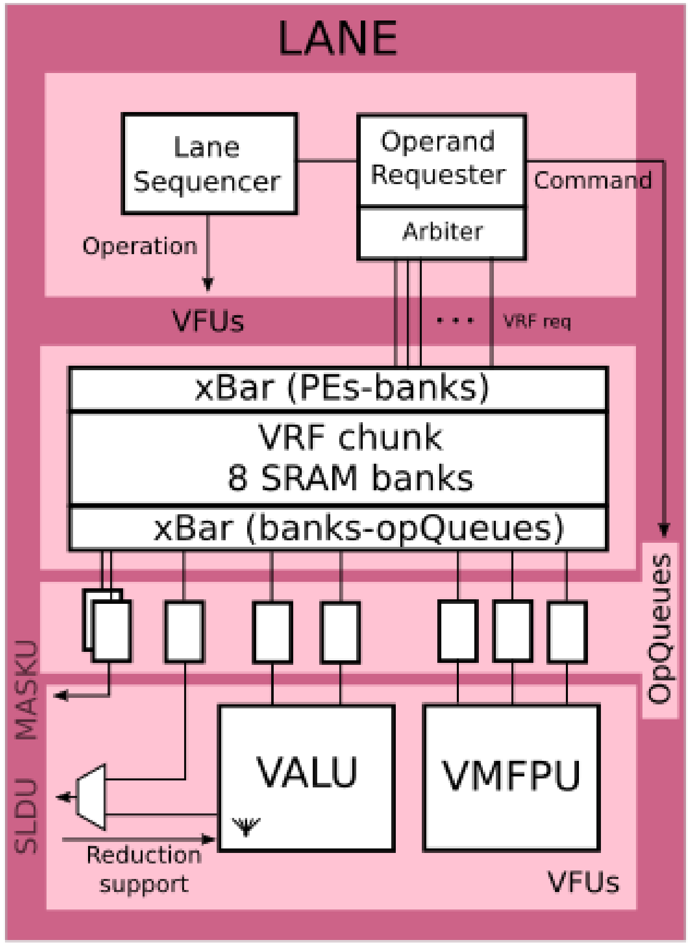
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



Ara System – The Lane

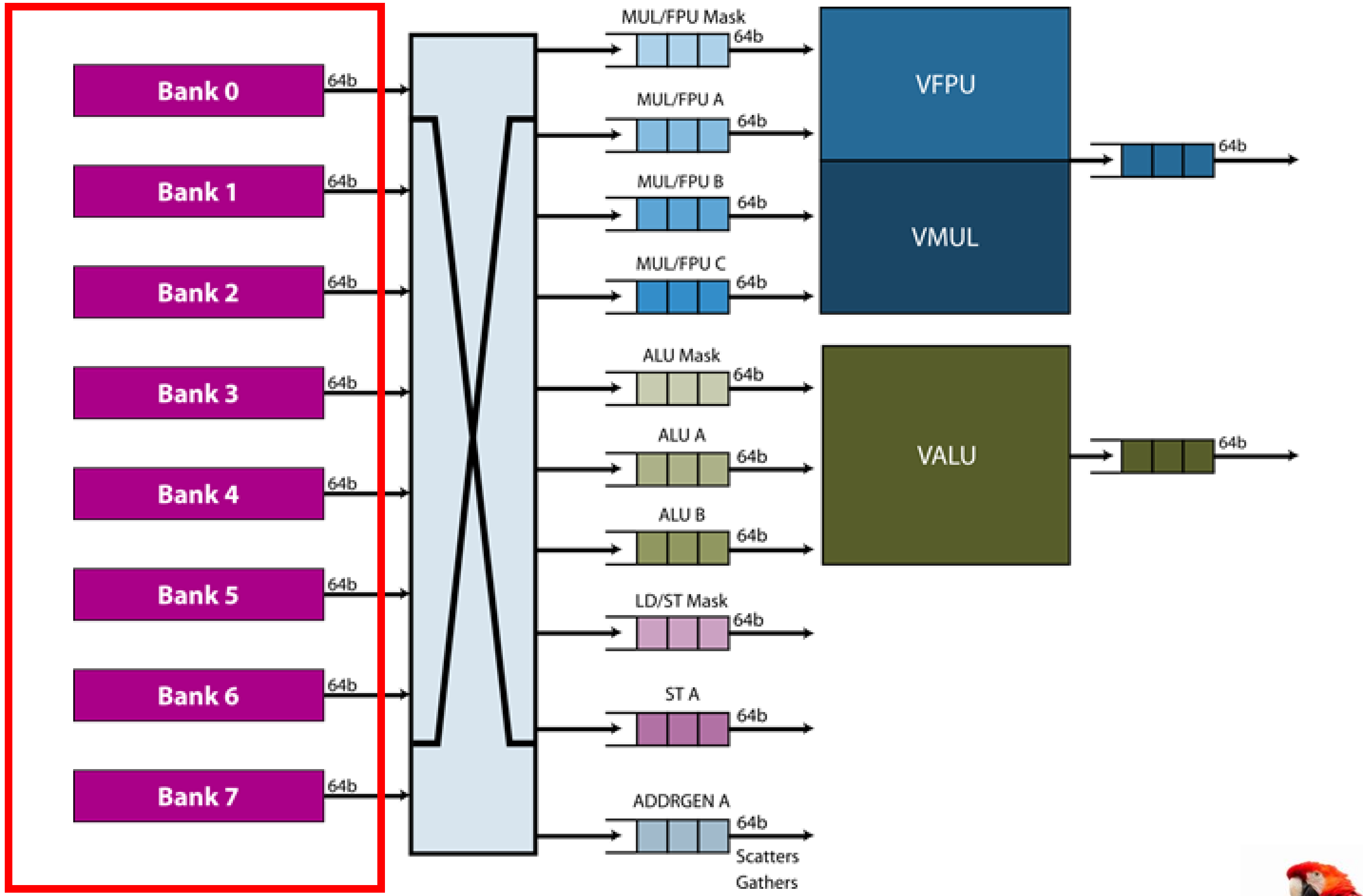
- **Computational** heart
  - One 64-bit SIMD FPU/lane
  - One 64-bit SIMD ALU/lane
- Keep a **chunk** of the **VRF**
- **Max**imize **locality**
  - Element 0 remains in lane 0
  - Minimize inter-lane communication
- VRF in-lane reads/writes



16 Lanes: 32 DP-FLOP/Cycle – 128 HP-FLOP/Cycle!

Ara System - The Lane

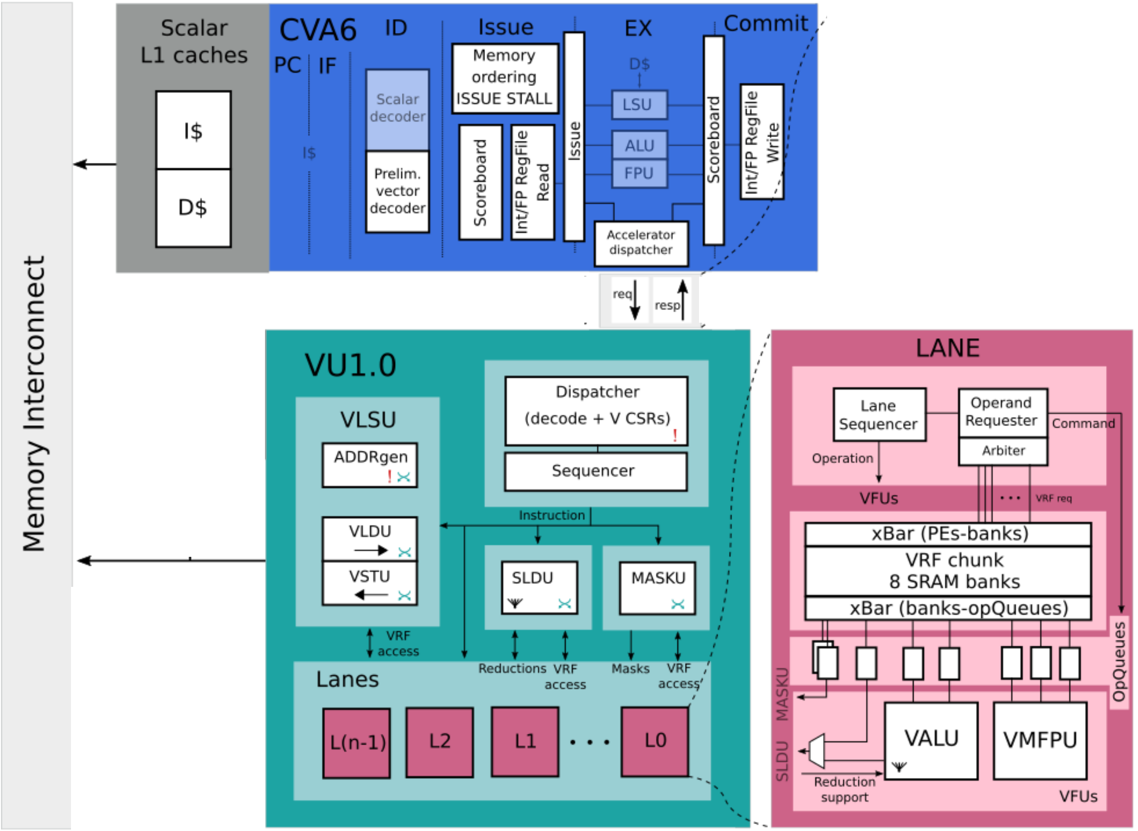
- **Modular VRF**
- **4 KiB/lane**
- **8 SRAM banks/lane**
- **Operand queues**
  - Amortize VRF bank conflicts



VRF Chunk (4 KiB)

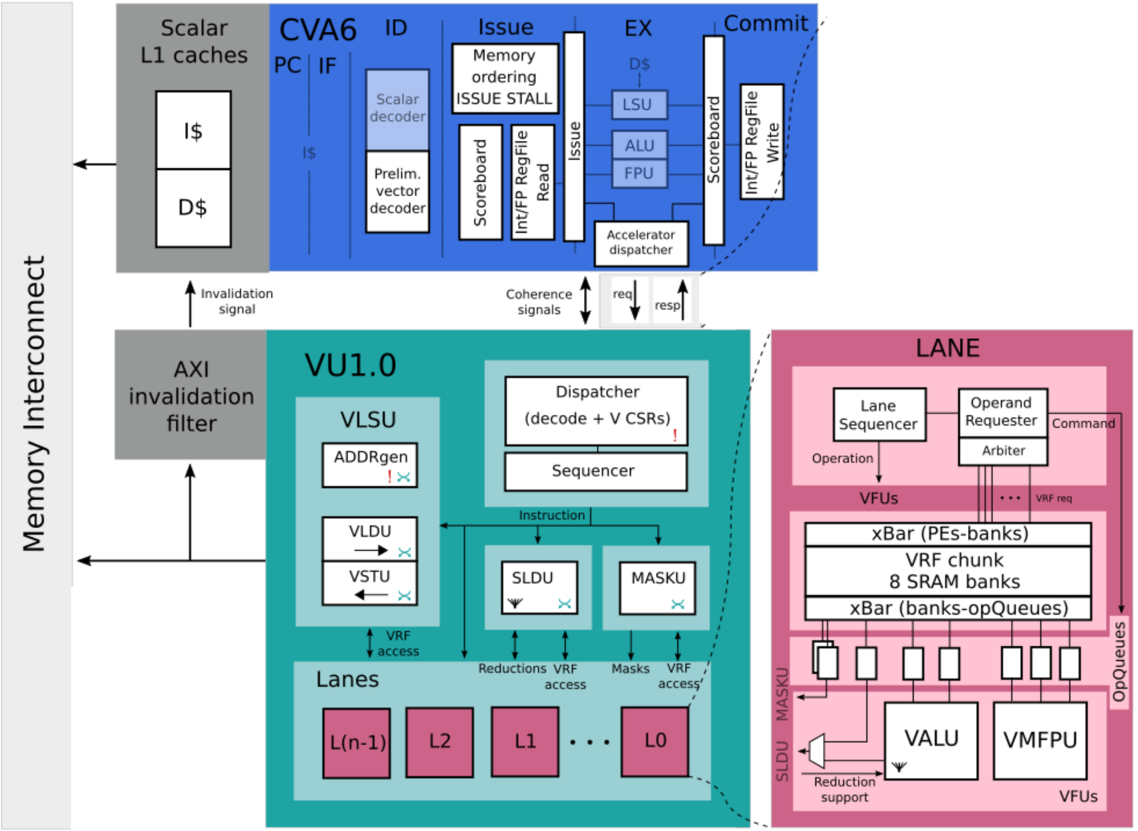
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



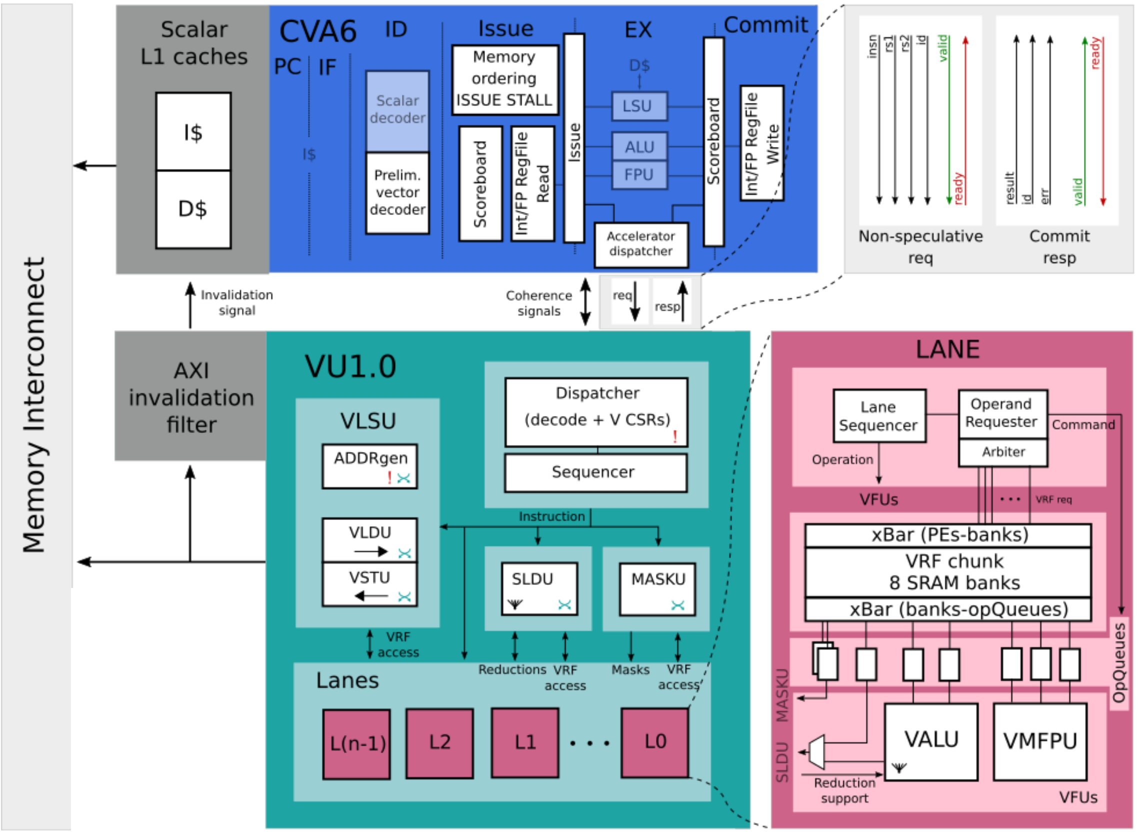
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)
- Memory coherence



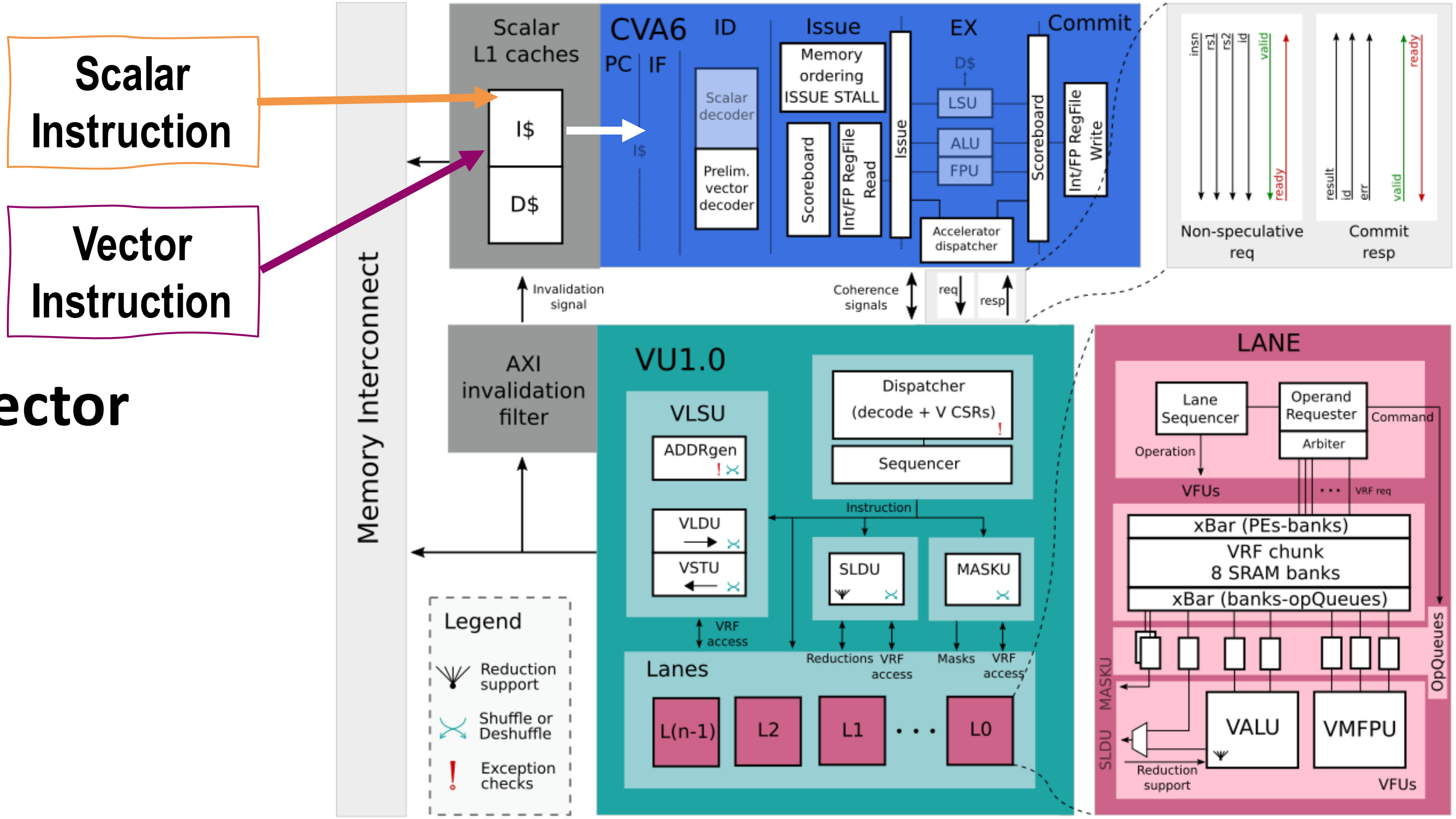
Ara System – One step below

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)
- Memory coherence
- Non-speculative interface



Ara System – Working principles

CVA6 and **Ara** work **in tandem!**

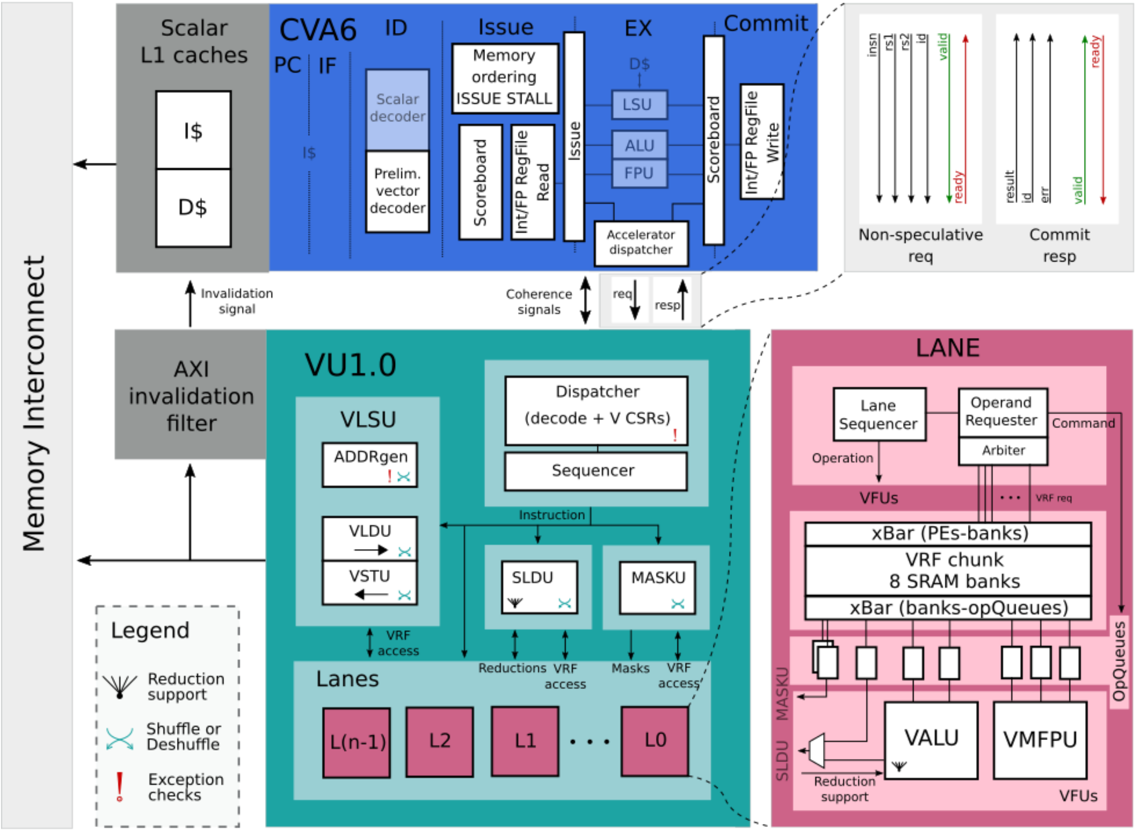

Scalar Instruction

Vector Instruction

CVA6 fetches **scalar** and **vector** instructions from memory

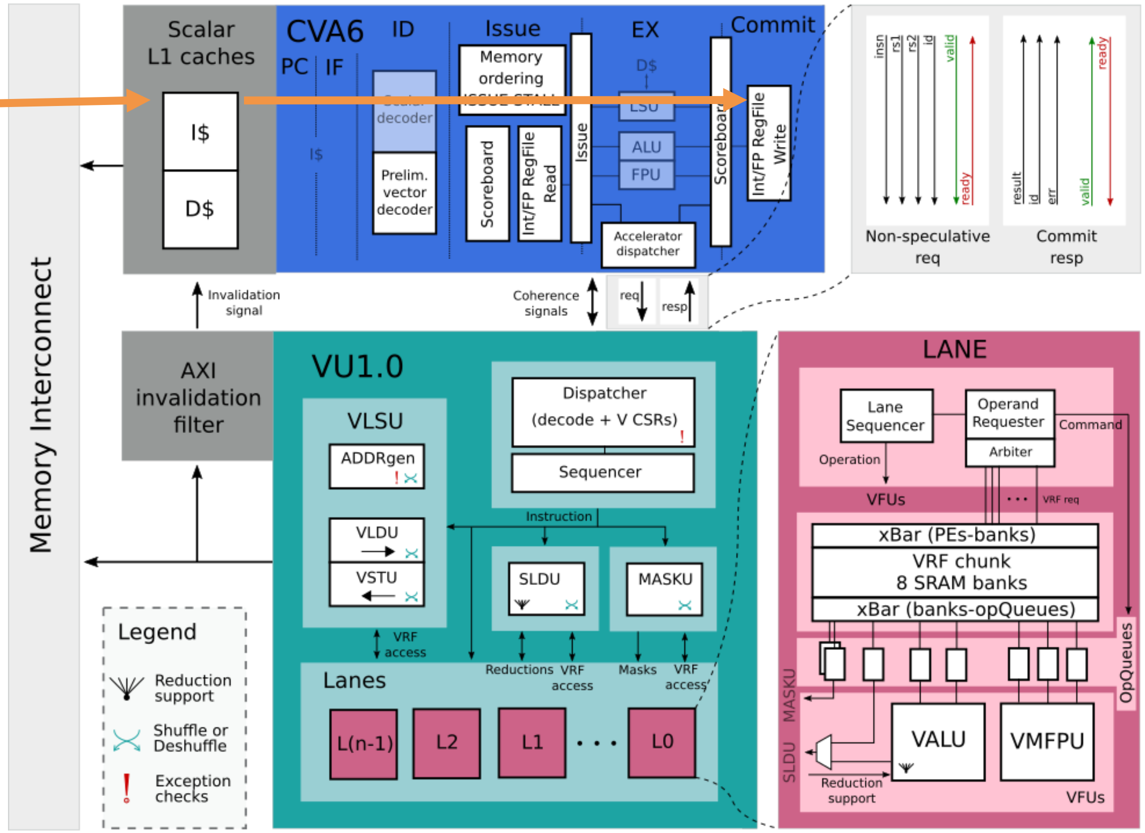
Ara System – Scalar instructions

Scalar Instruction



Ara System – Scalar instructions

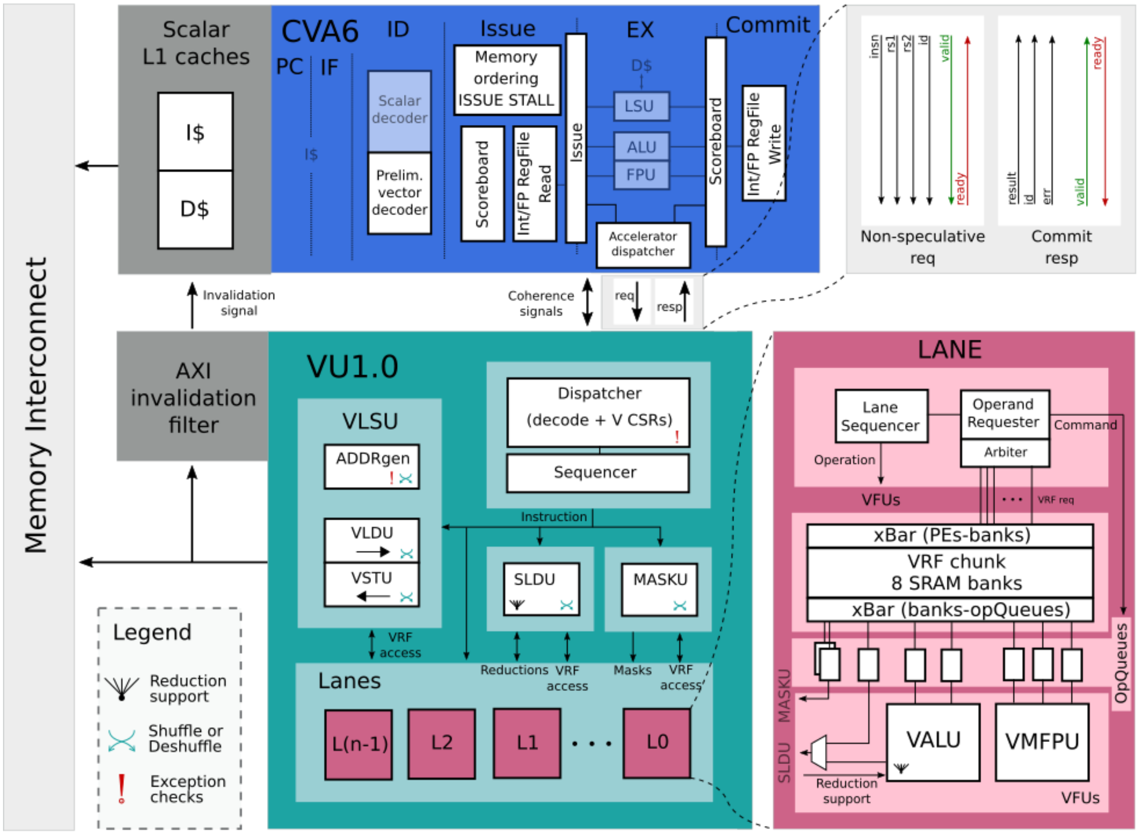
Scalar Instruction



Scalar execution – **CVA6**

Ara System – Vector instructions

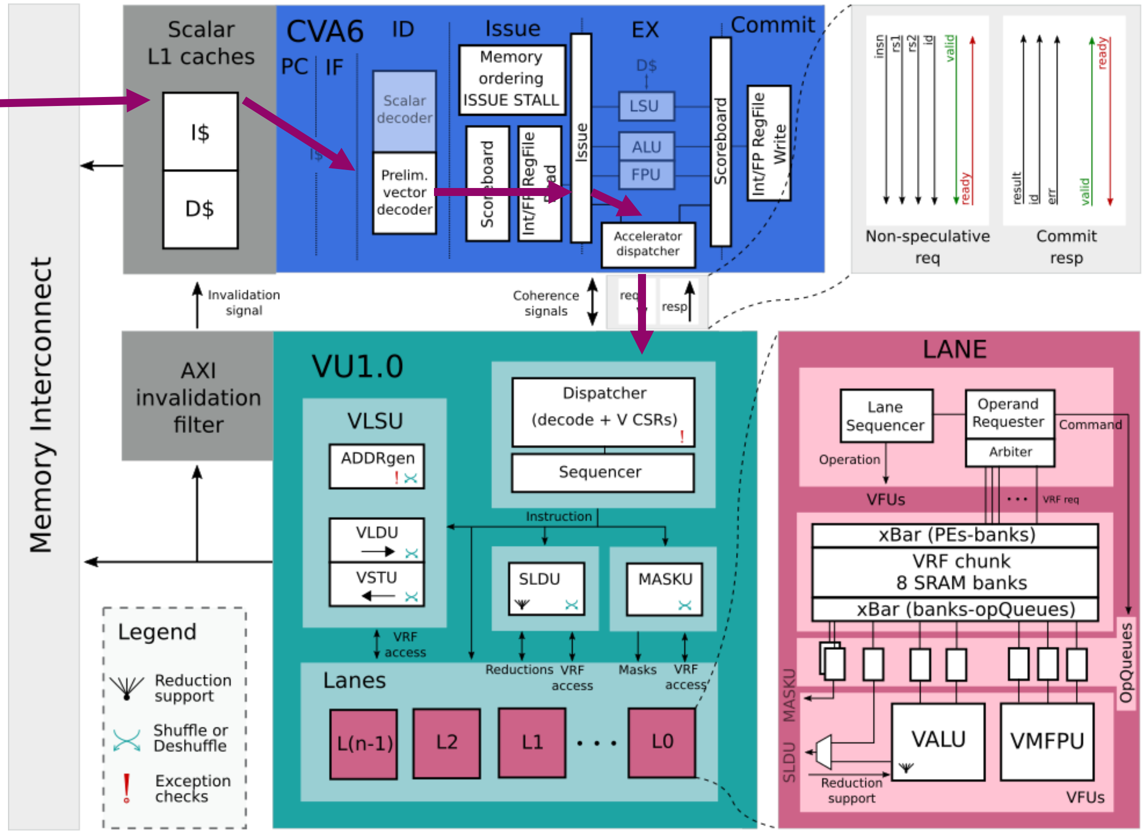
Vector Instruction



Ara System – Vector instructions

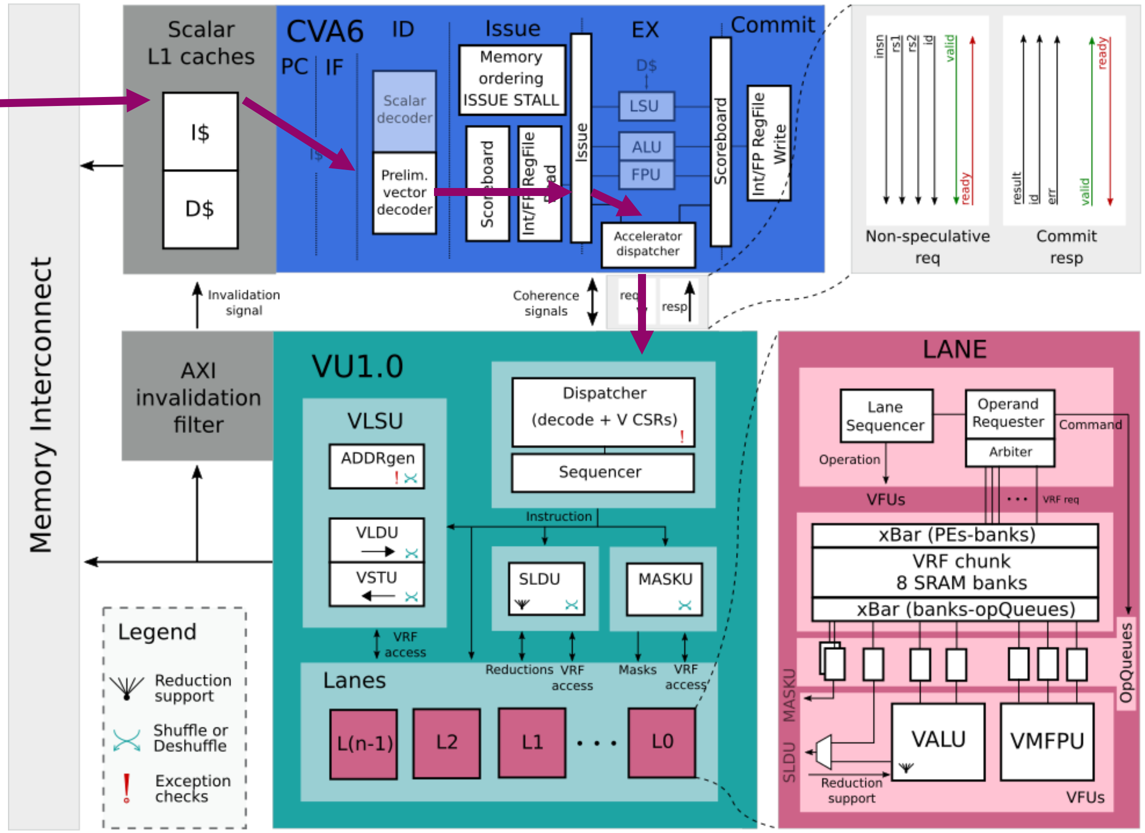
Vector Instruction

- Vector instruction **dispatch**



Ara System – Vector instructions

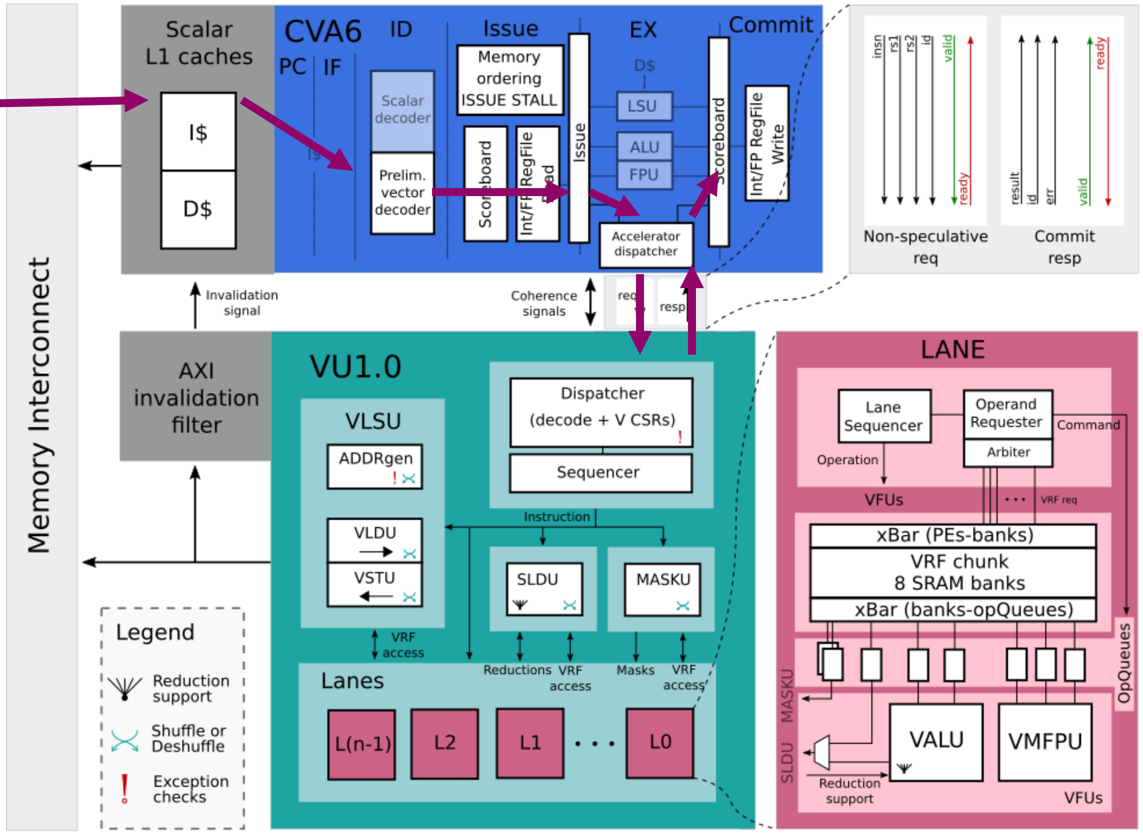
Vector Instruction



- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**

Ara System – Vector instructions

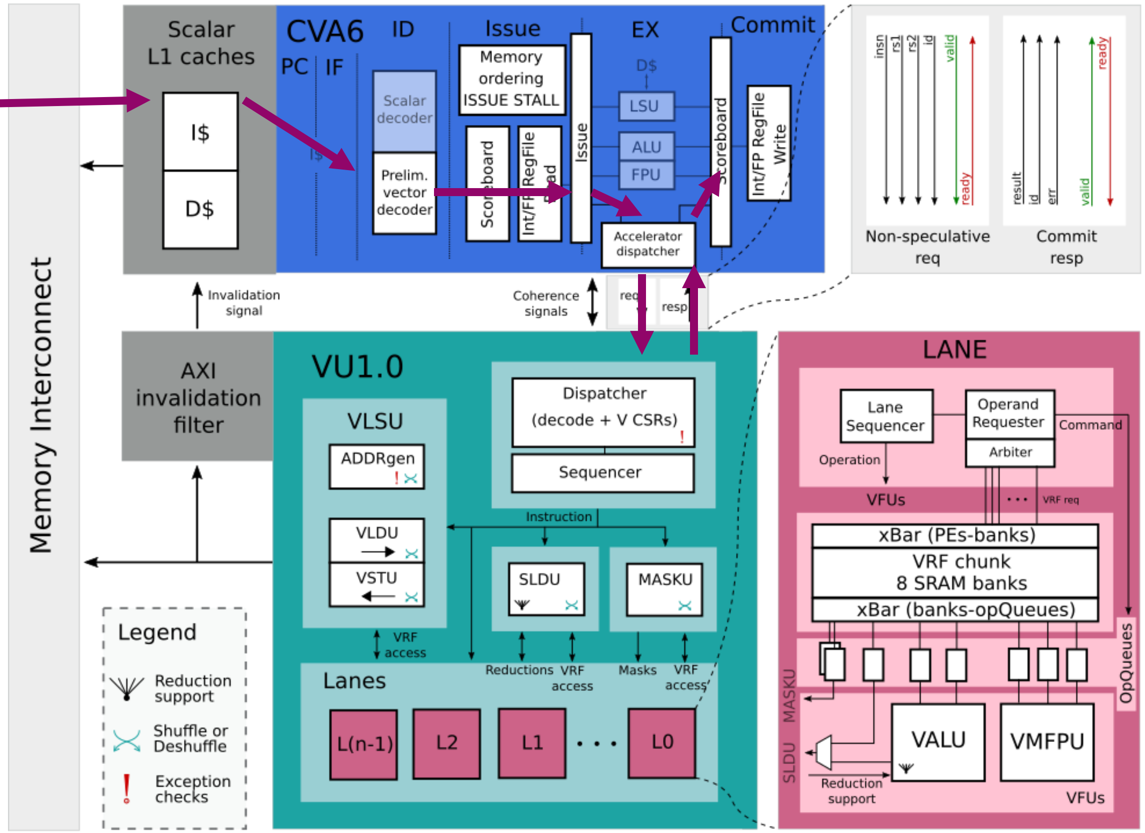
Vector Instruction



- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back

Ara System – Vector instructions

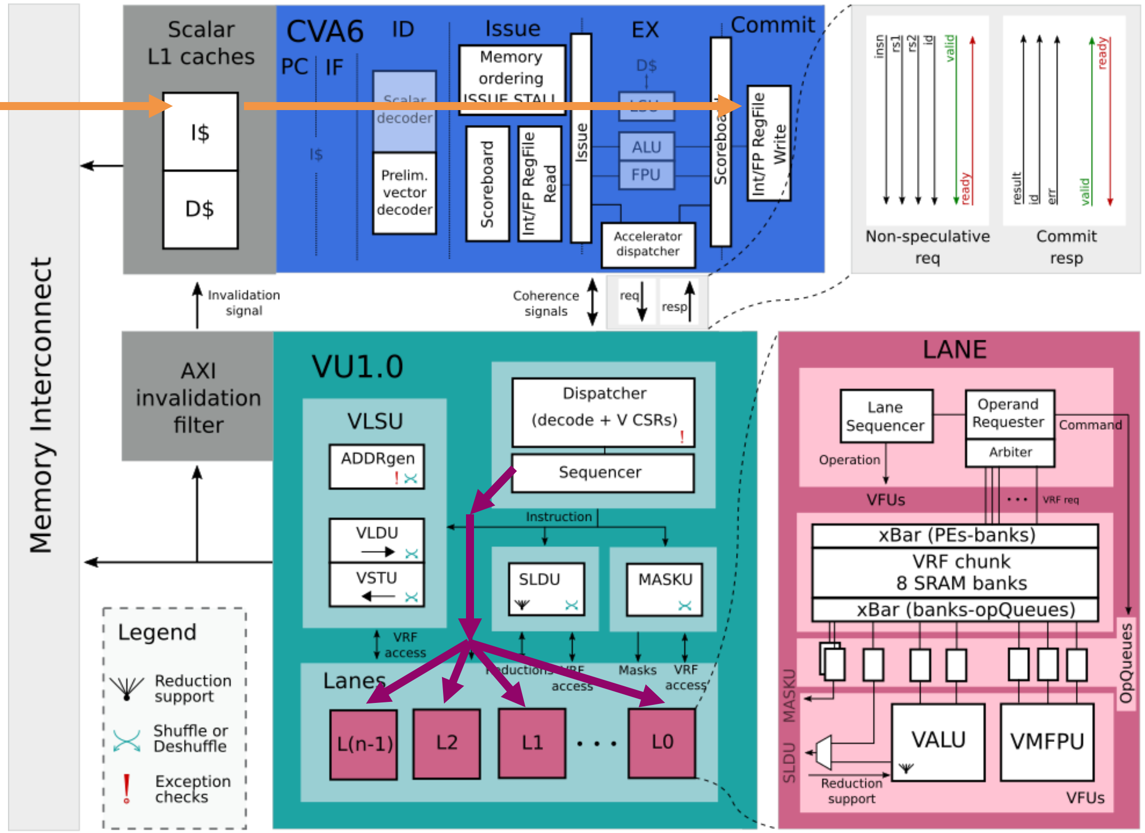
Vector Instruction



- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back
- **No exceptions**? CVA6 **commits**

Ara System – Vector instructions

Scalar Instruction

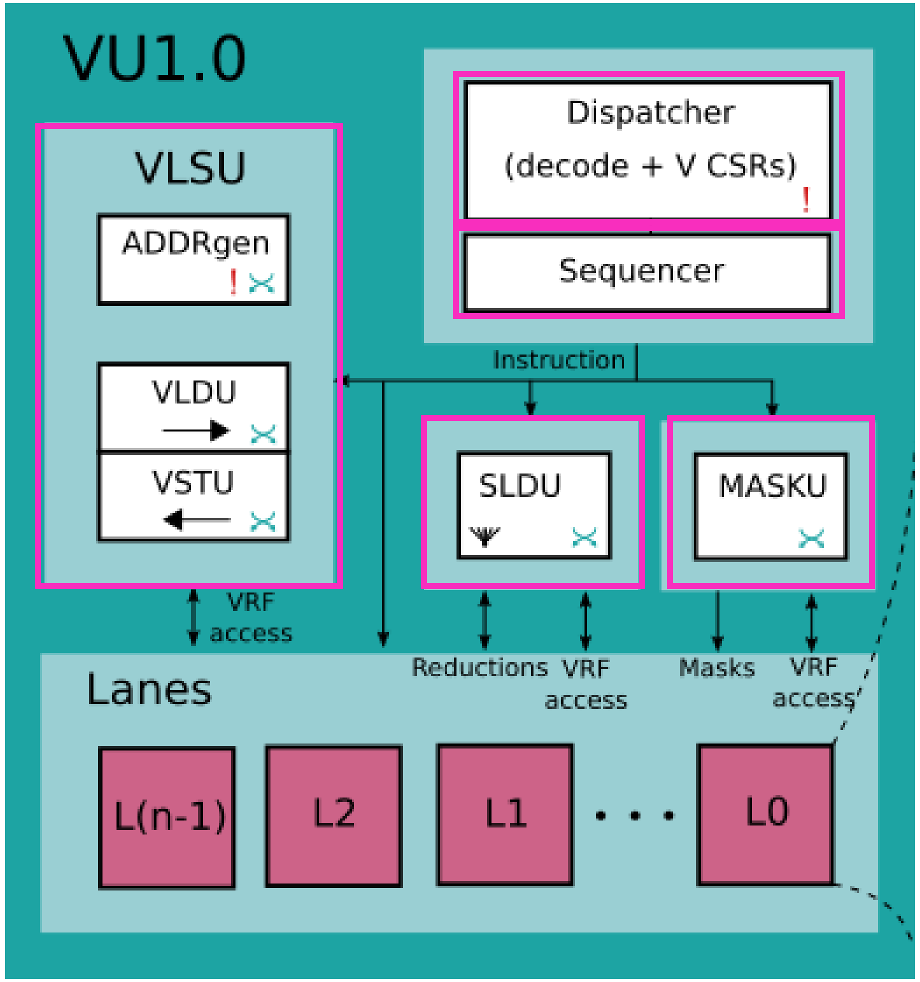


Vector Instruction

- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back
- **No exceptions**? CVA6 **commits**
- **Ara executes** the **instruction**

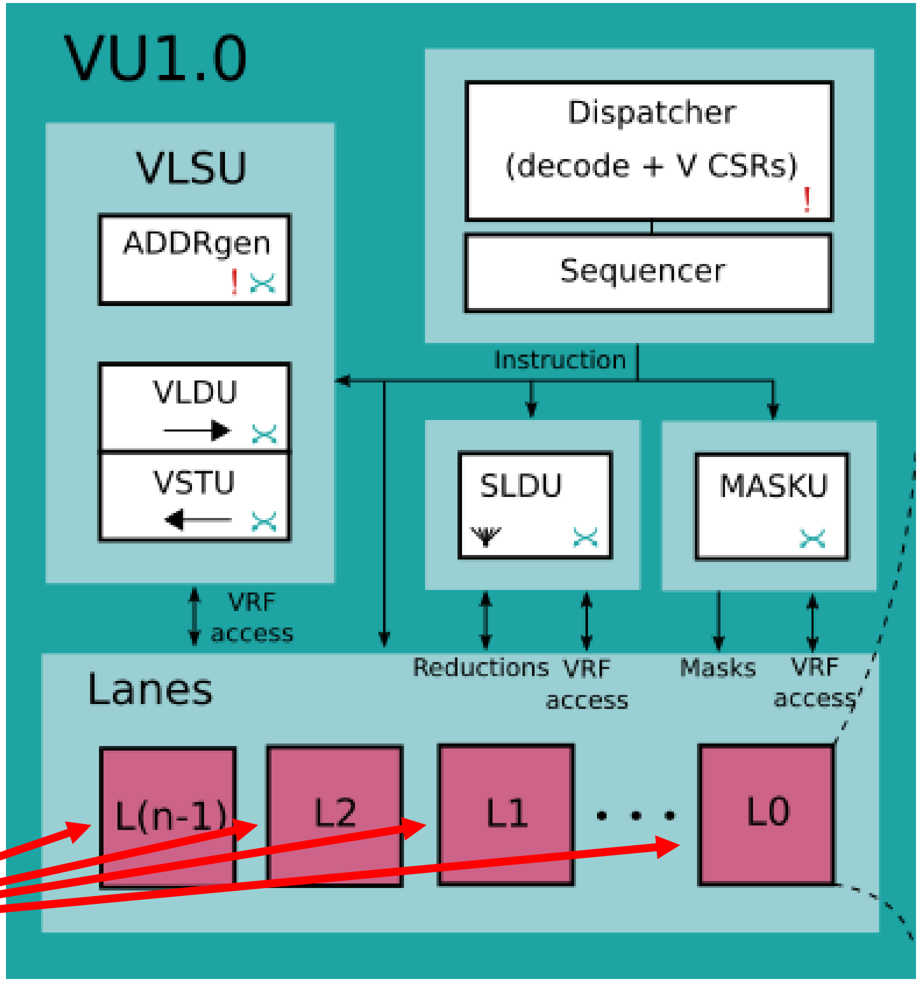
Ara

- Dispatcher (decode + CSRs)
- Sequencer (issue + hazards)
- Private VLSU (vload + vstore)
- Slide Unit (permutations)
- Mask Unit (predication)



Ara

- Dispatcher (decode + CSRs)
- Sequencer (issue + hazards)
- Private VLSU (vload + vstore)
- Slide Unit (permutations)
- Mask Unit (predication)
- Lanes (computation + VRF)



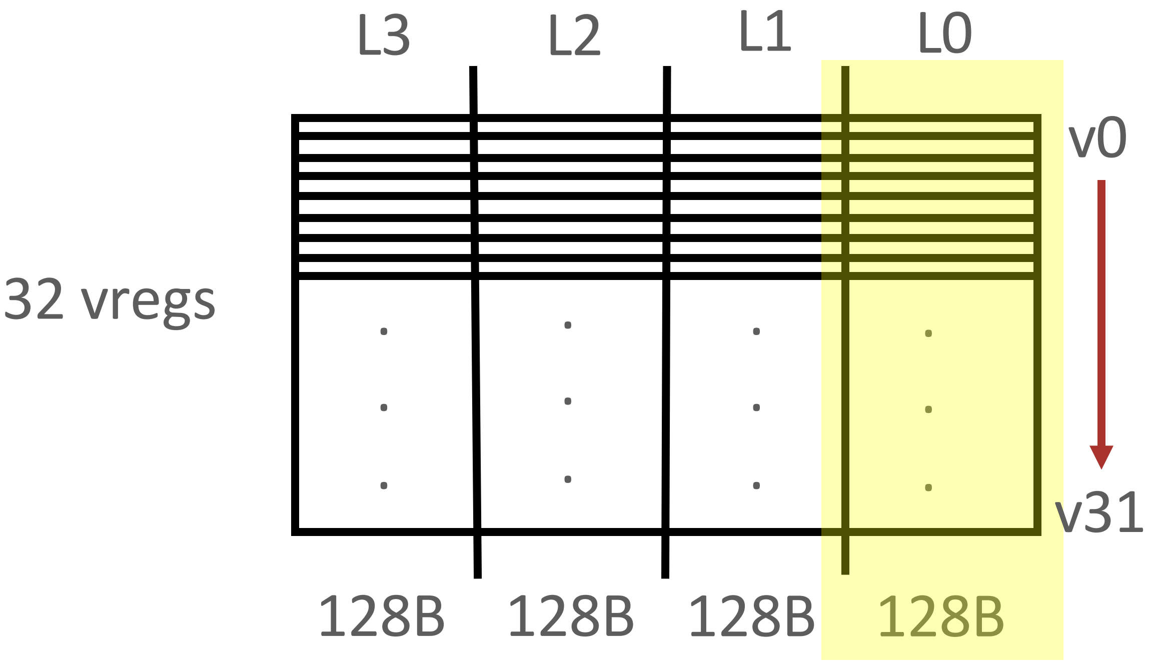
Parametric number of lanes!
From 2 to 16

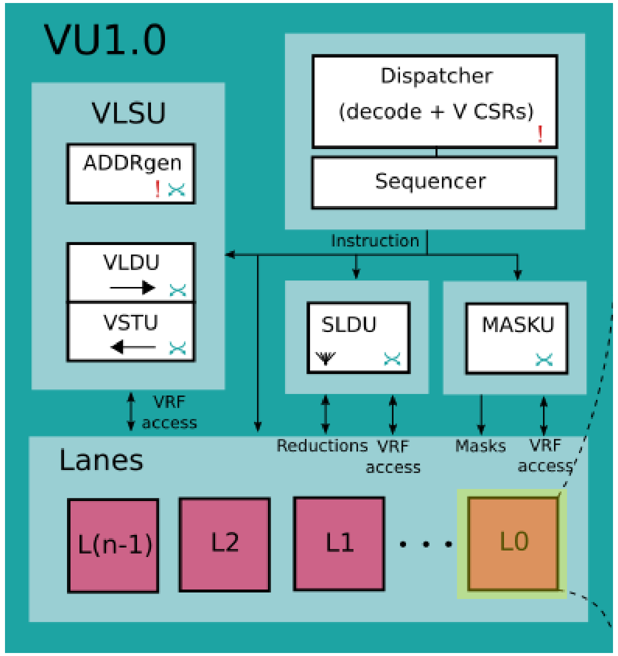
Ara VRF

Ara 4 Lanes VRF

Size: 16 KiB

Split among the lanes!

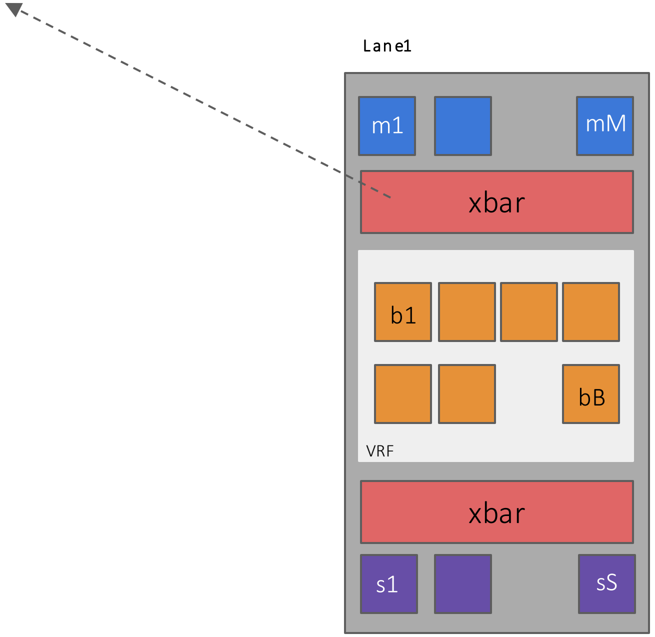




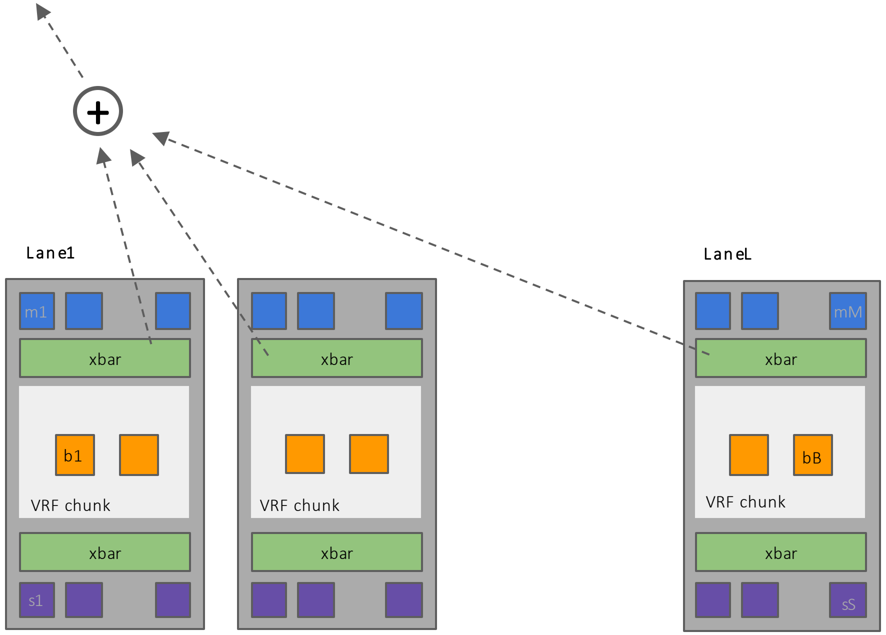
VRF – Monolithic vs. Split

```
#B memory banks
#M masters
#S slaves
#L lanes
```

$$A_{\text{xbar}}^{\text{mono}} \propto (M_{\text{lane}} \times \ell) \times B_{\text{tot}} = \boxed{M_{\text{lane}} \times 8 \times \ell^2}$$

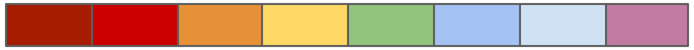


$$A_{\text{xbar}}^{\text{split}} \propto M_{\text{lane}} \times \mathrm{B}_{\text{lane}} \times \ell = \boxed{M_{\text{lane}} \times 8 \times \ell}$$



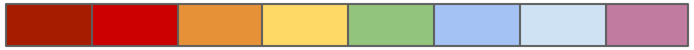
Ara VRF

- Vector with 8 elements
- 2 Byte/element

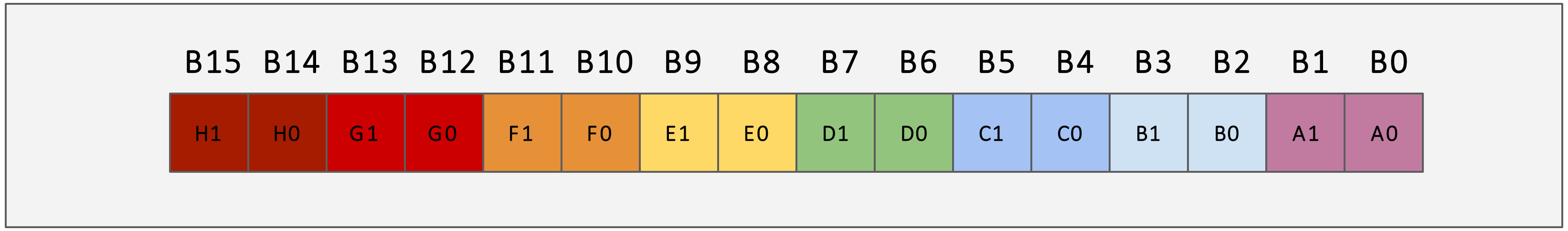


Ara VRF

- Vector with 8 elements
- 2 Byte/element



Memory (outside Ara)



B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

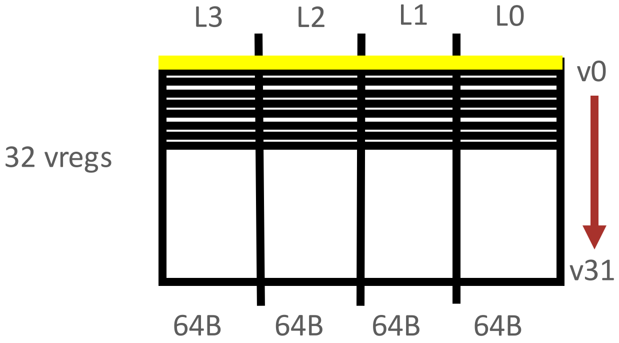
Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

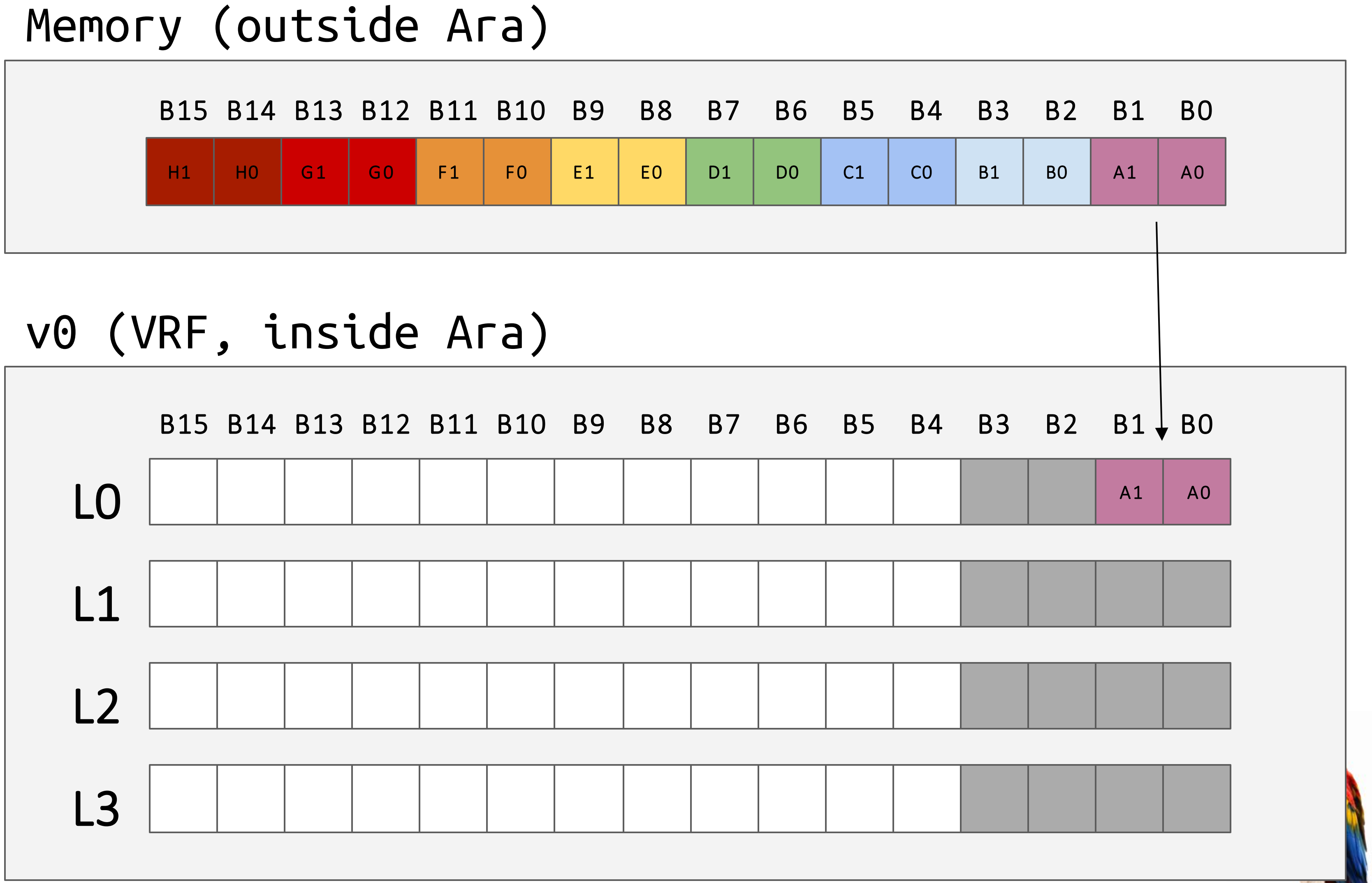
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													▨	▨	▨	▨
L1													▨	▨	▨	▨
L2													▨	▨	▨	▨
L3													▨	▨	▨	▨

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

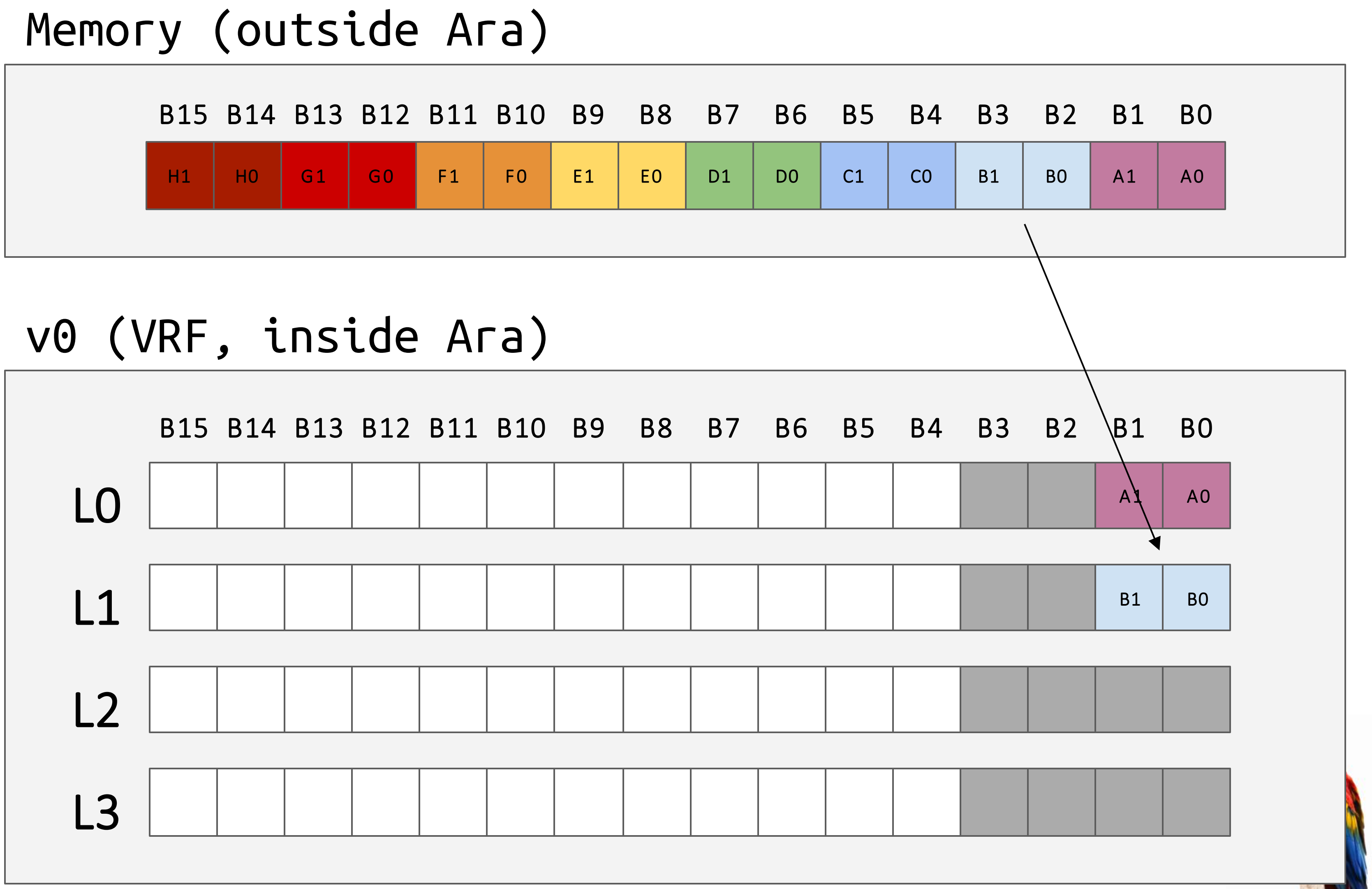
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0															A1	A0
L1																
L2																
L3																

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

v0 (VRF, inside Ara)

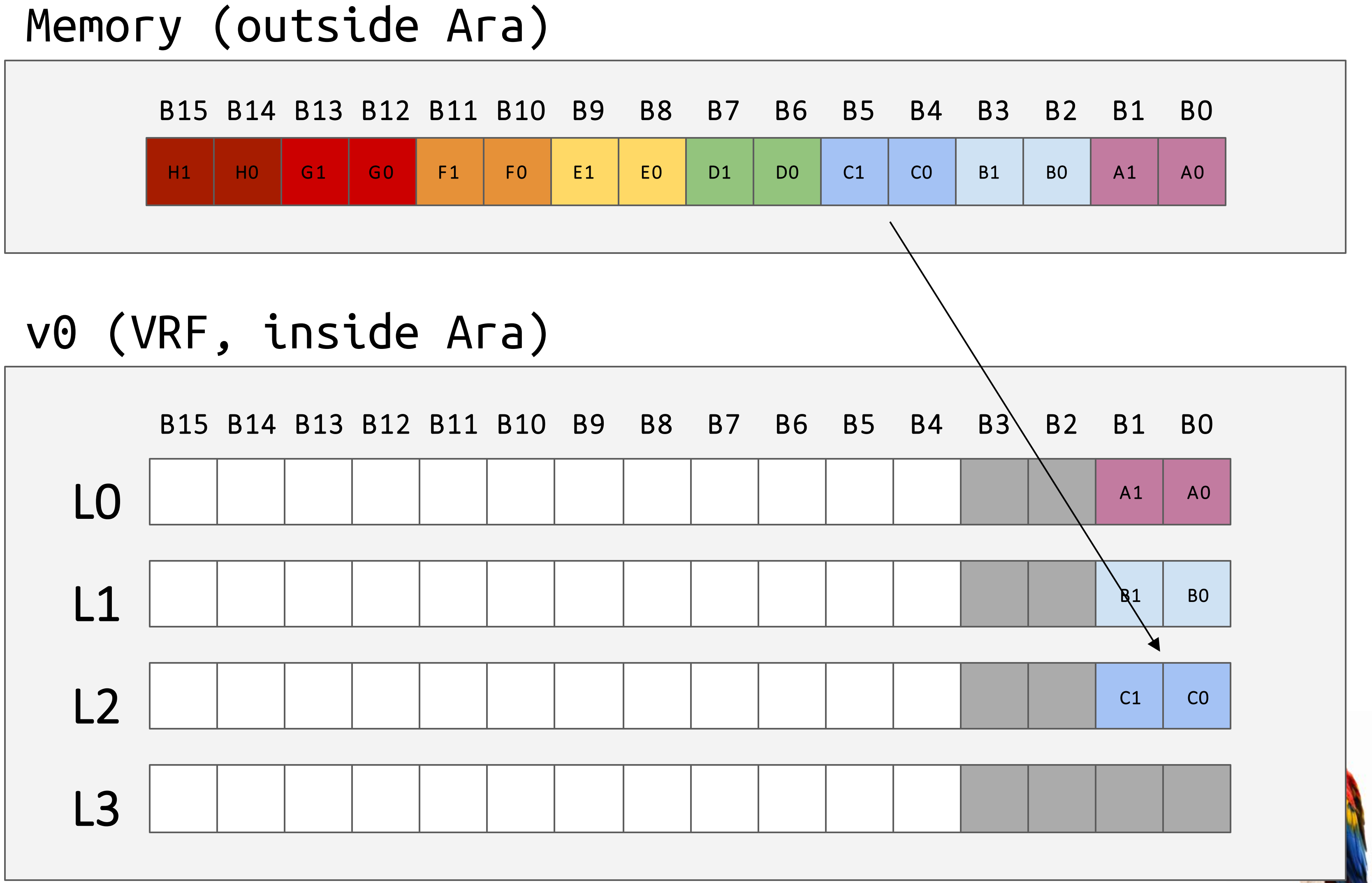
	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0															A1	A0
L1															B1	B0
L2																
L3																

Ara VRF

- Vector with 8 elements
- 2 Byte/element

🟥🟥🟧🟨🟩🟦🟦🟪

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

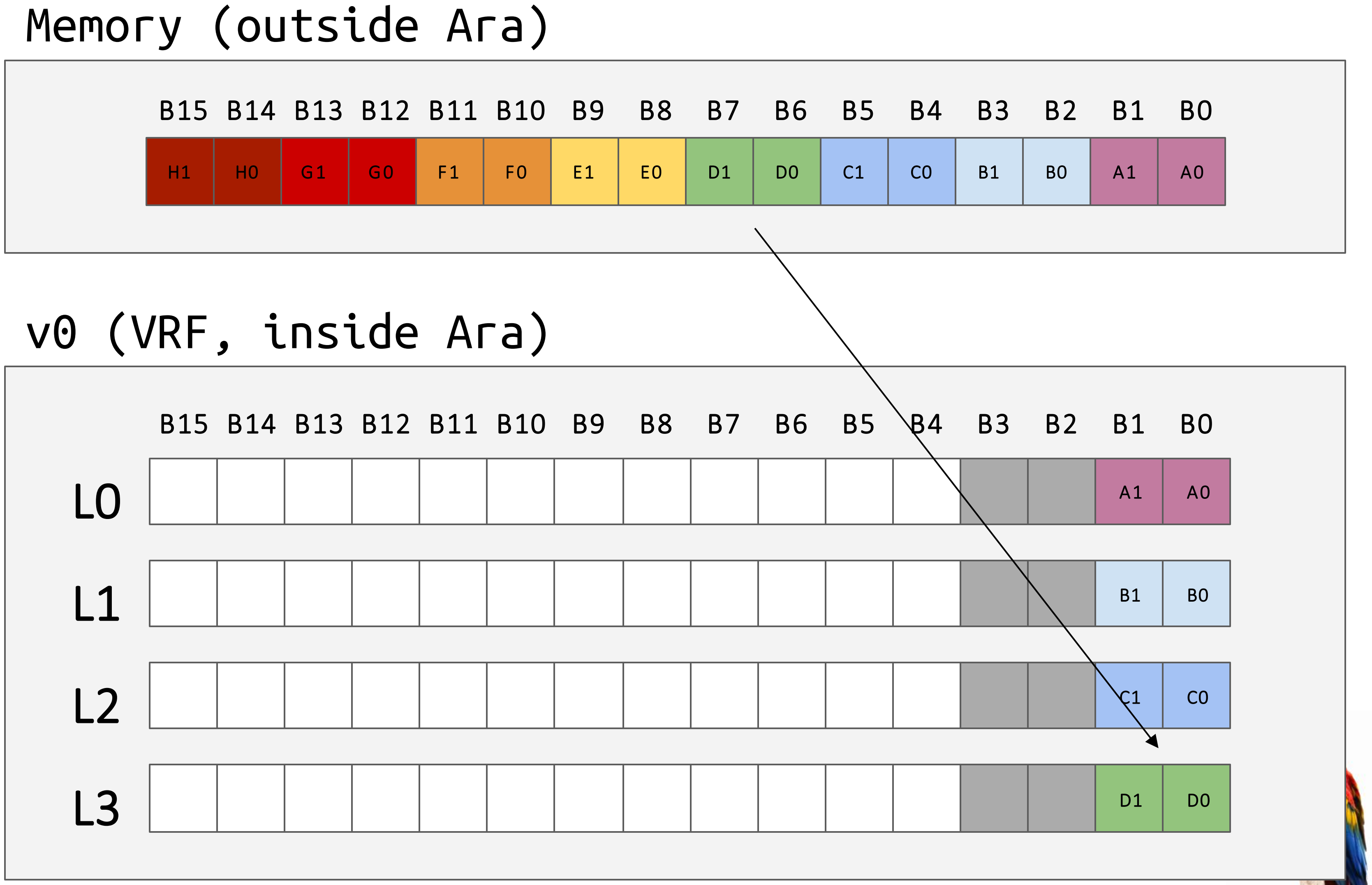
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													▨	▨	A1	A0
L1													▨	▨	B1	B0
L2													▨	▨	C1	C0
L3													▨	▨	▨	▨

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

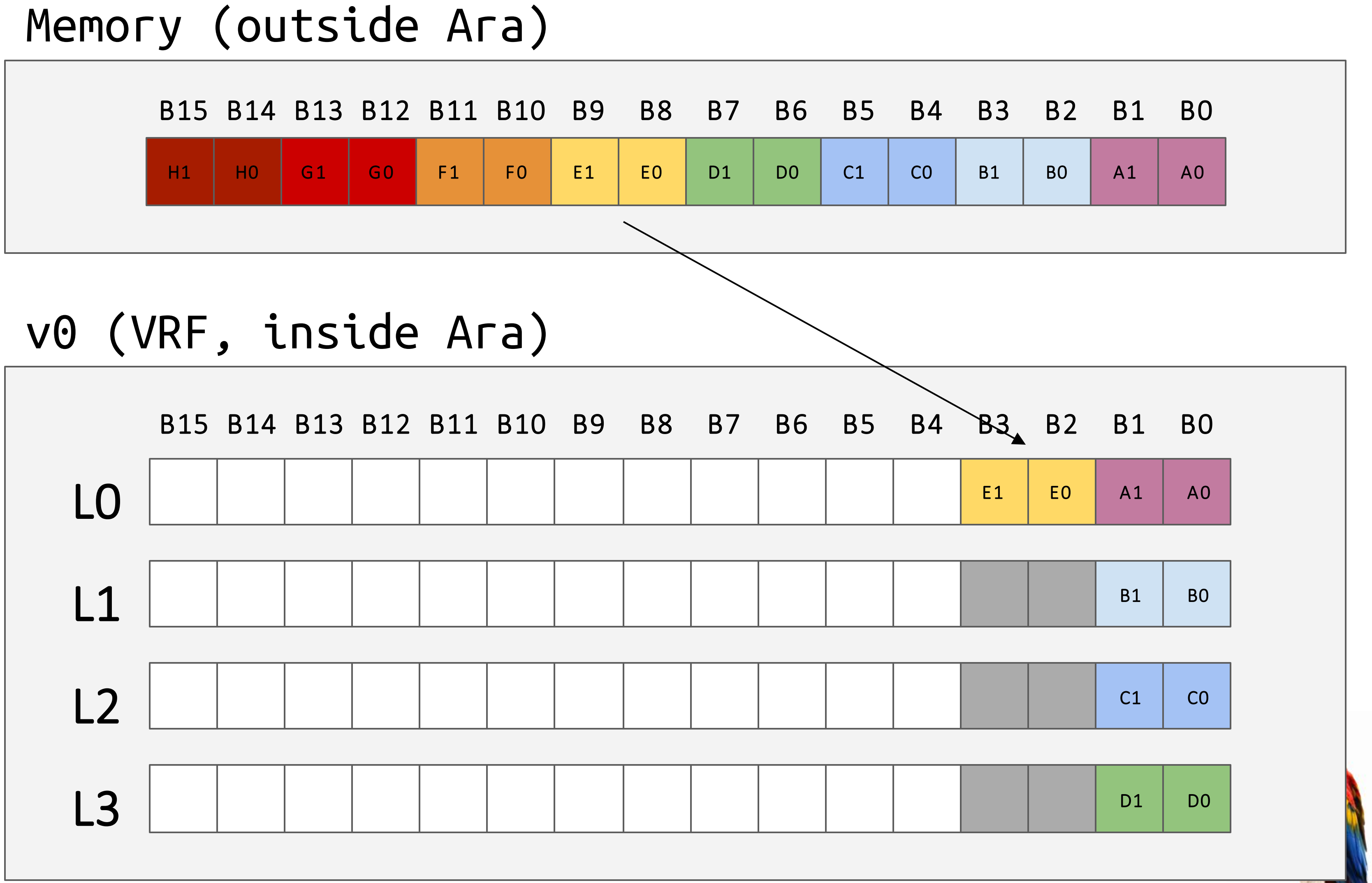
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													▨	▨	A1	A0
L1													▨	▨	B1	B0
L2													▨	▨	C1	C0
L3													▨	▨	D1	D0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

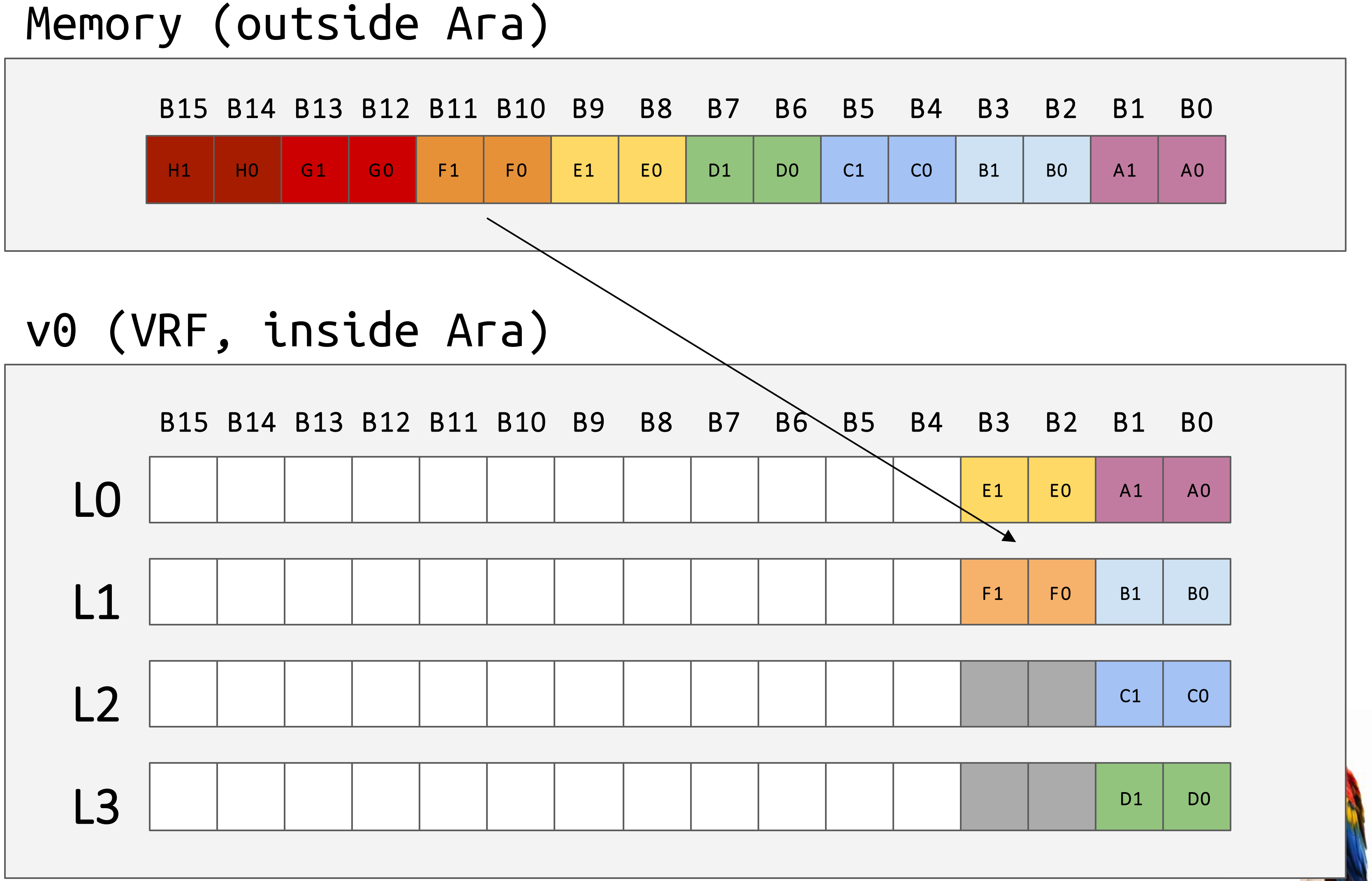
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													▨	▨	B1	B0
L2													▨	▨	C1	C0
L3													▨	▨	D1	D0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

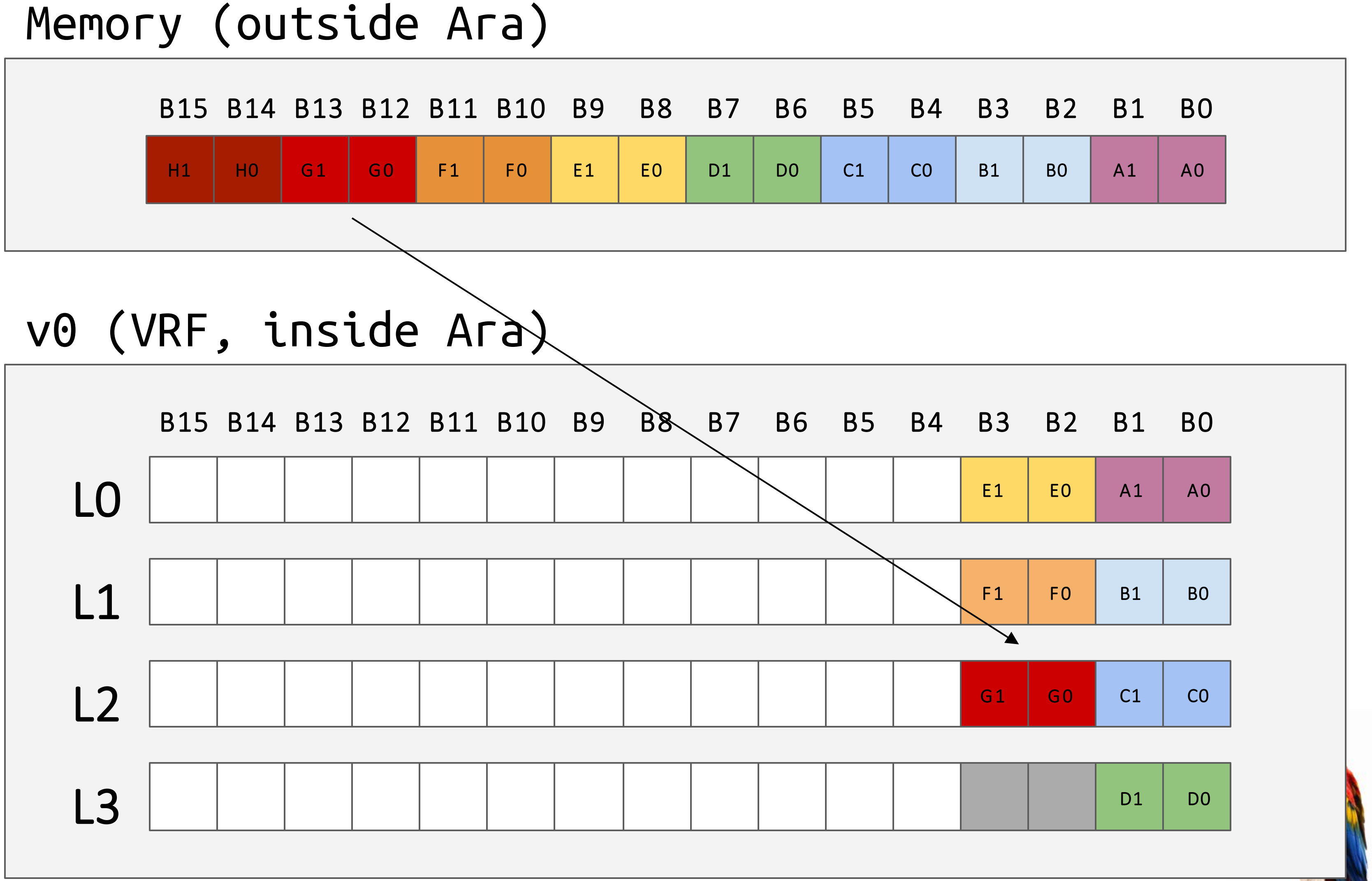
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2															C1	C0
L3															D1	D0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

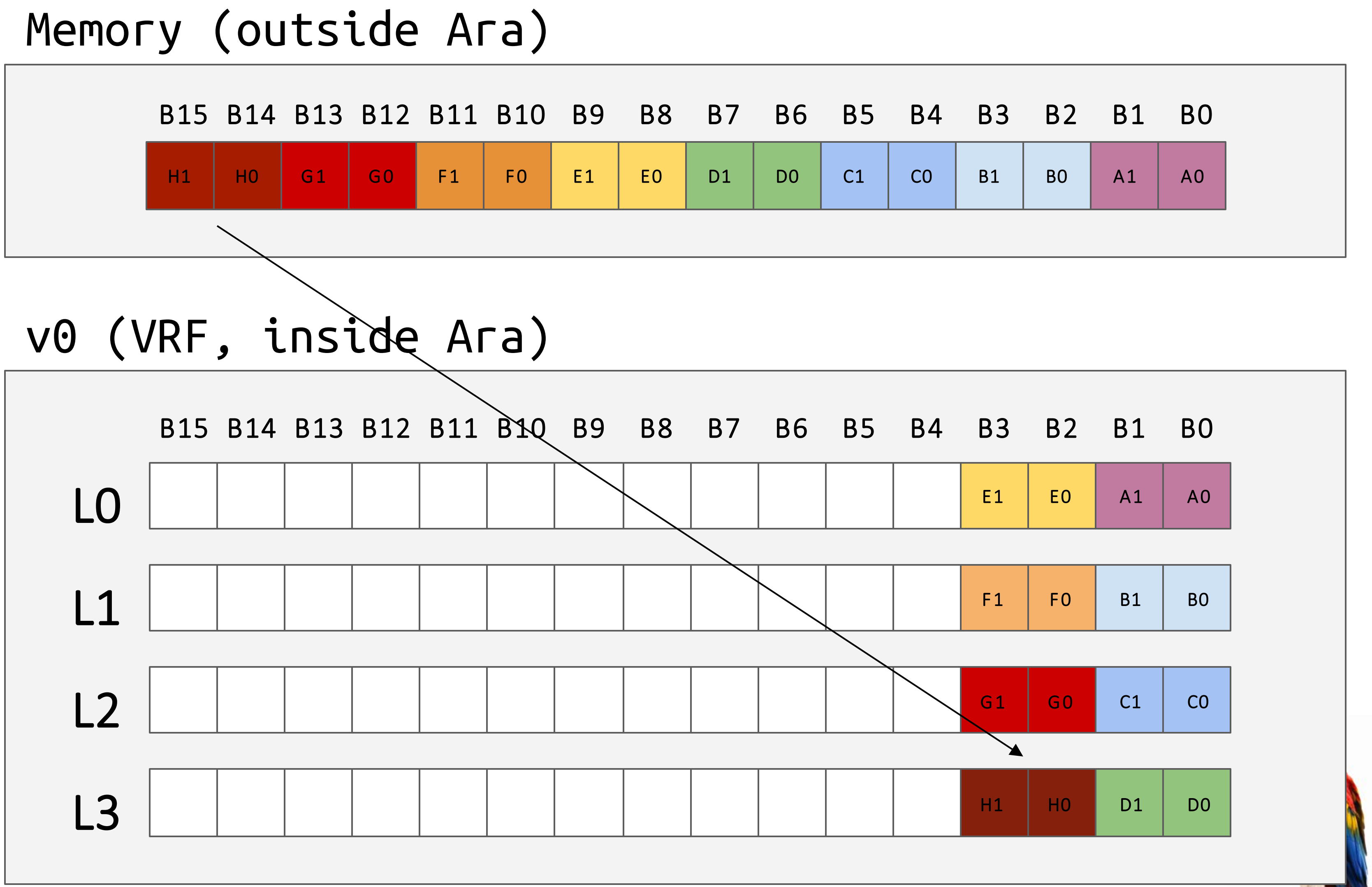
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3															D1	D0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

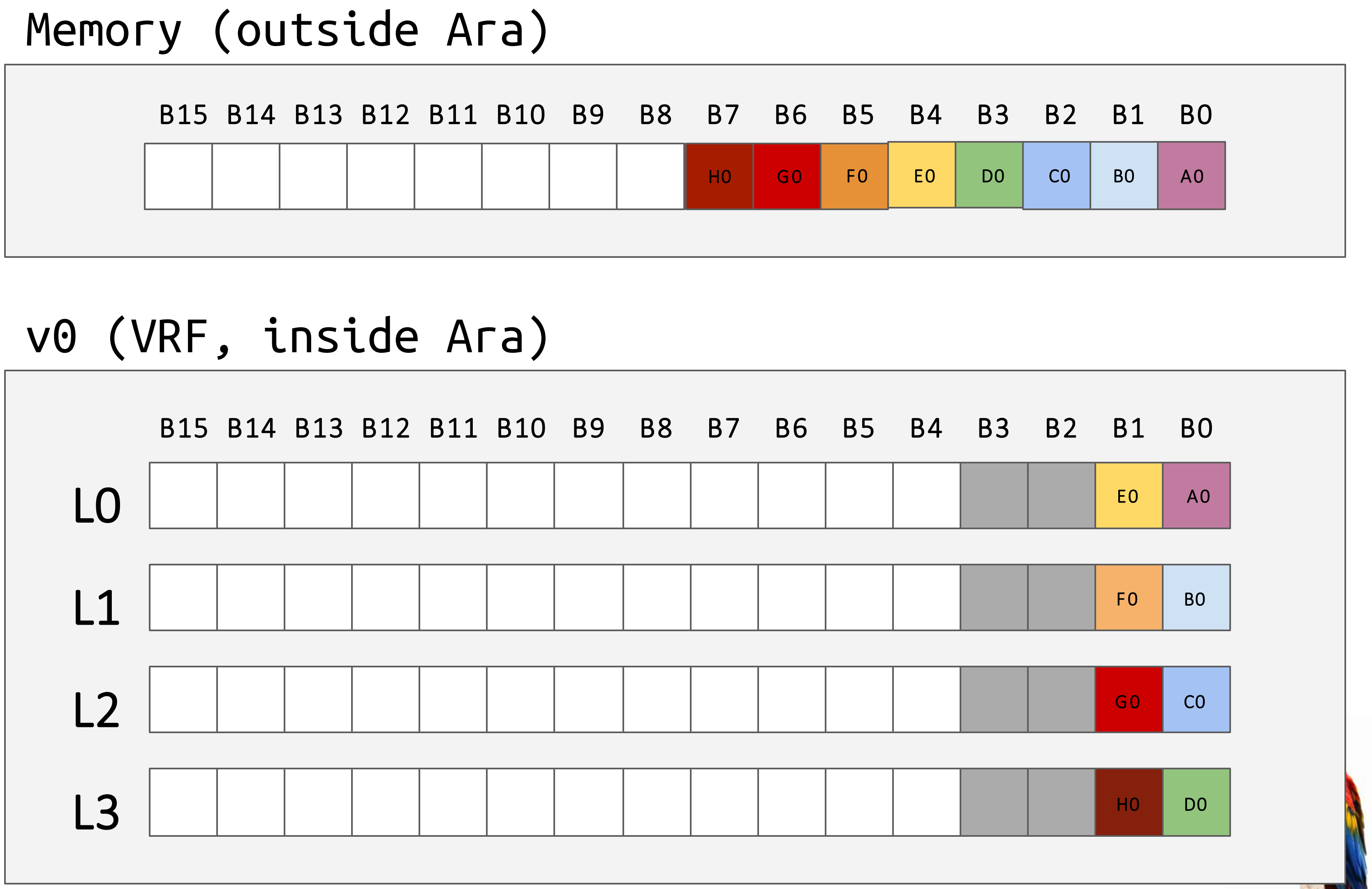
v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

Ara VRF

- Vector with 8 elements
- 1 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

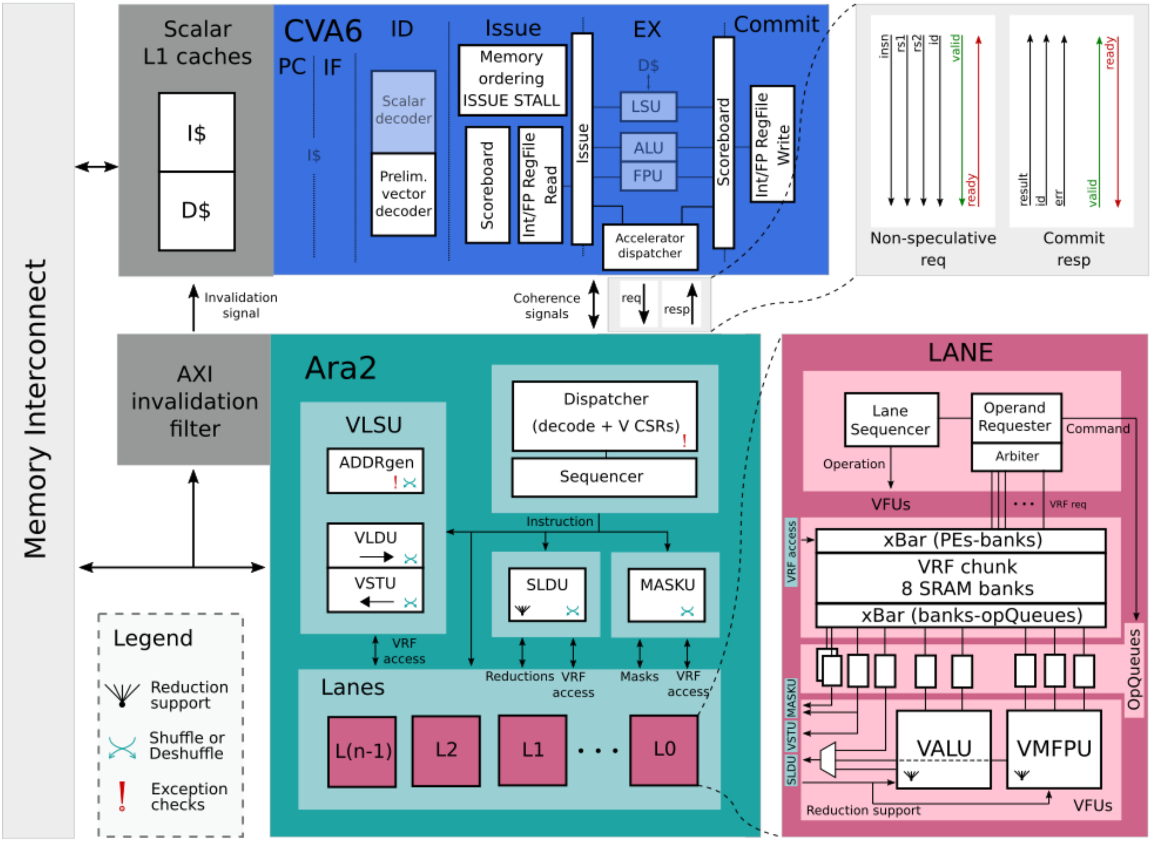
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
								H0	G0	F0	E0	D0	C0	B0	A0

v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0															E0	A0
L1															F0	B0
L2															G0	C0
L3															H0	D0

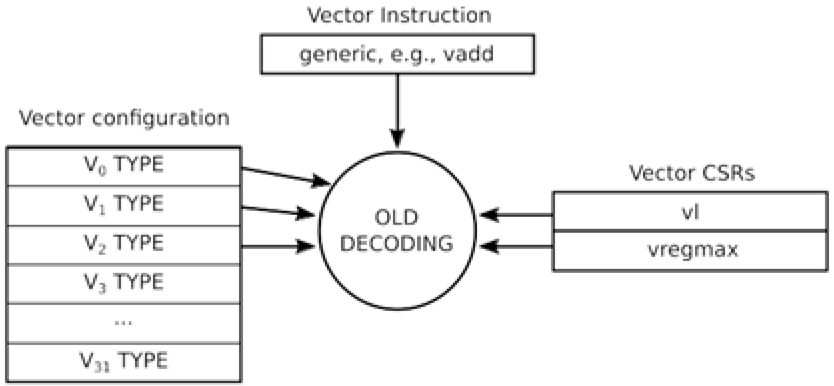
Differences between Ara1 and Ara2

- **Specs update** (RVV0.5 -> RVV1.0)
  - New VRF layout
  - Mask Unit
  - New insn encoding
- **Reshuffling** support
- **Fixed-point** support
- Int + FP **reductions**
- **Barber's pole**
- CVA6 – more agnostic on V
- Coherence/ordering protocol



From Ara1 to Ara2 – ISA update

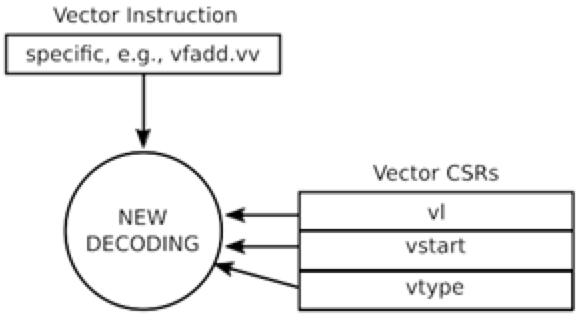
RVV 0.5



Polymorphic encoding

Compact ISA, harder decoding

RVV 1.0



Monomorphic encoding

Longer ISA, easier decoding

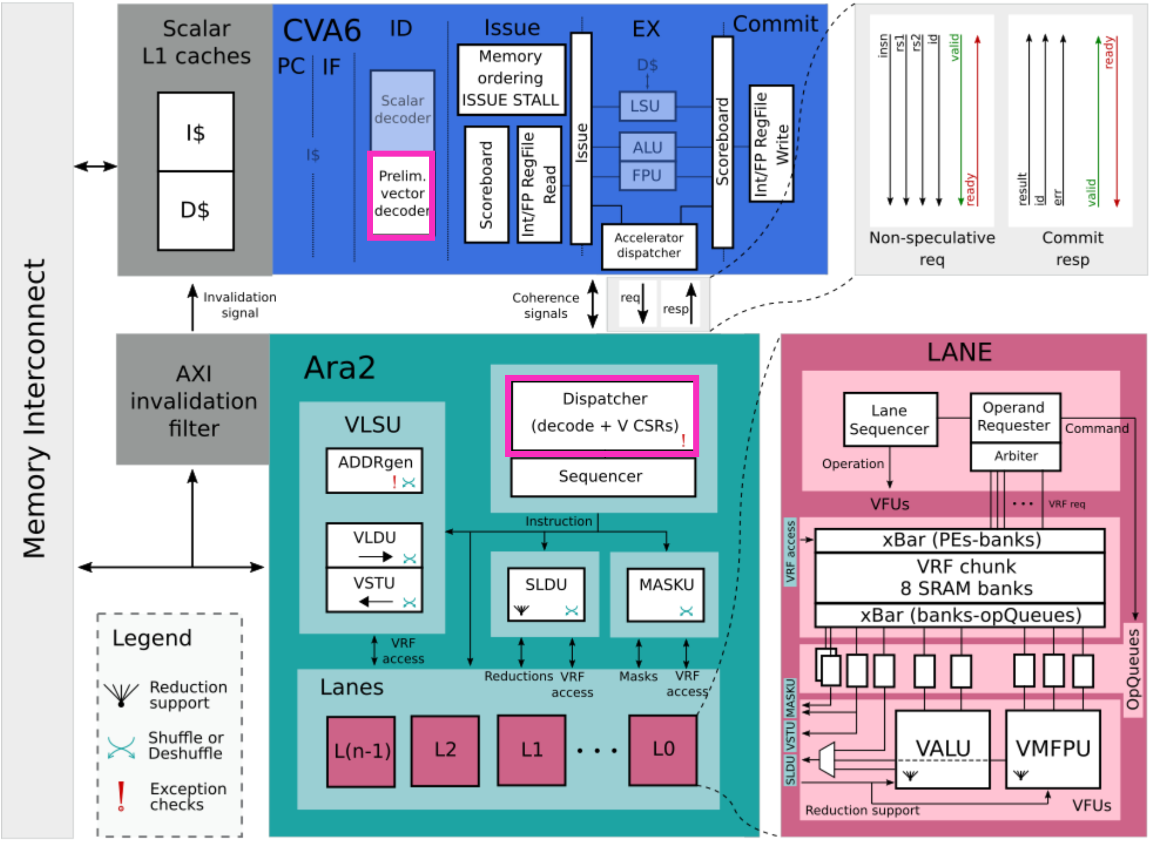
From Ara1 to Ara2 – ISA update

RVV 0.5

- Need for VRF state to decode instruction
- CVA6 was not vector agnostic
- V-decoder in CVA6

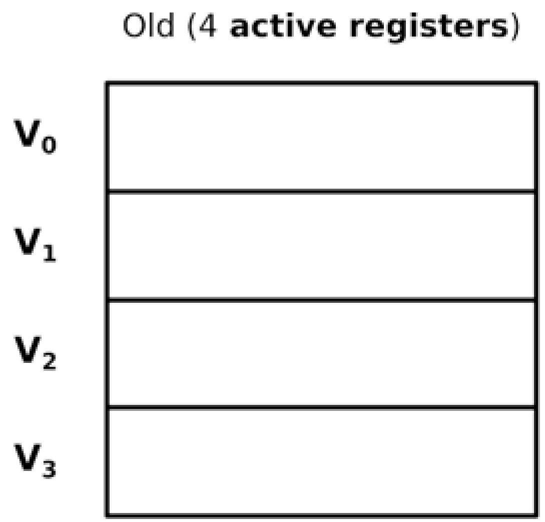
RVV 1.0

- Easier to decode instructions
- CVA6 needs only minimal information to process V instructions
- Push V-related hardware in Ara!



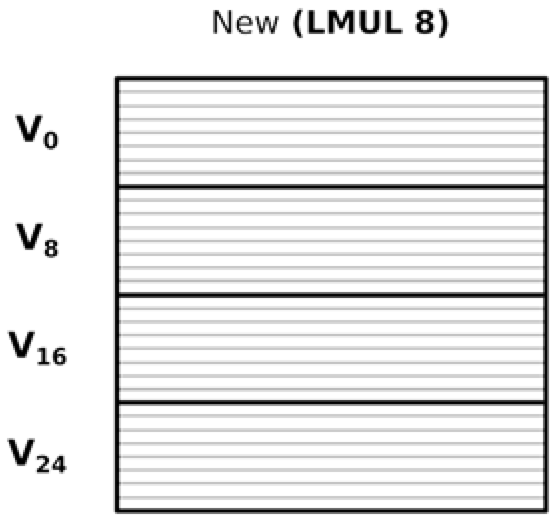
From Ara1 to Ara2 – ISA update

RVV 0.5



Reg enable/length modified at runtime
Data-type per register

RVV 1.0



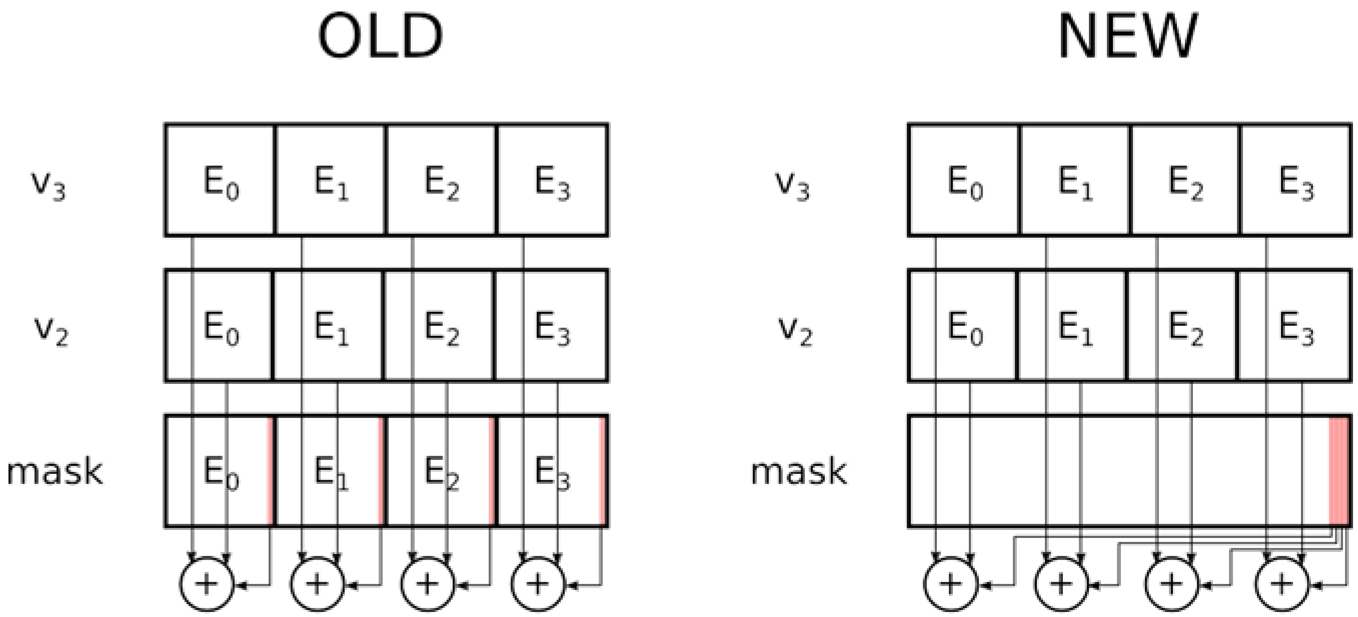
Constrained VRF setting
Data-type agnostic

From Ara1 to Ara2 – ISA update

- **Simpler VRF configuration**
- **VRF** configuration is now static
- **32 vector registers**, but **they can be merged** in groups of 2, 4, 8 registers (**LMUL** parameter == 2, 4, 8)
- Interpret vector registers depending on SEW (single-element width) only

From Ara1 to Ara2 – ISA update

RVV 0.5 RVV 1.0



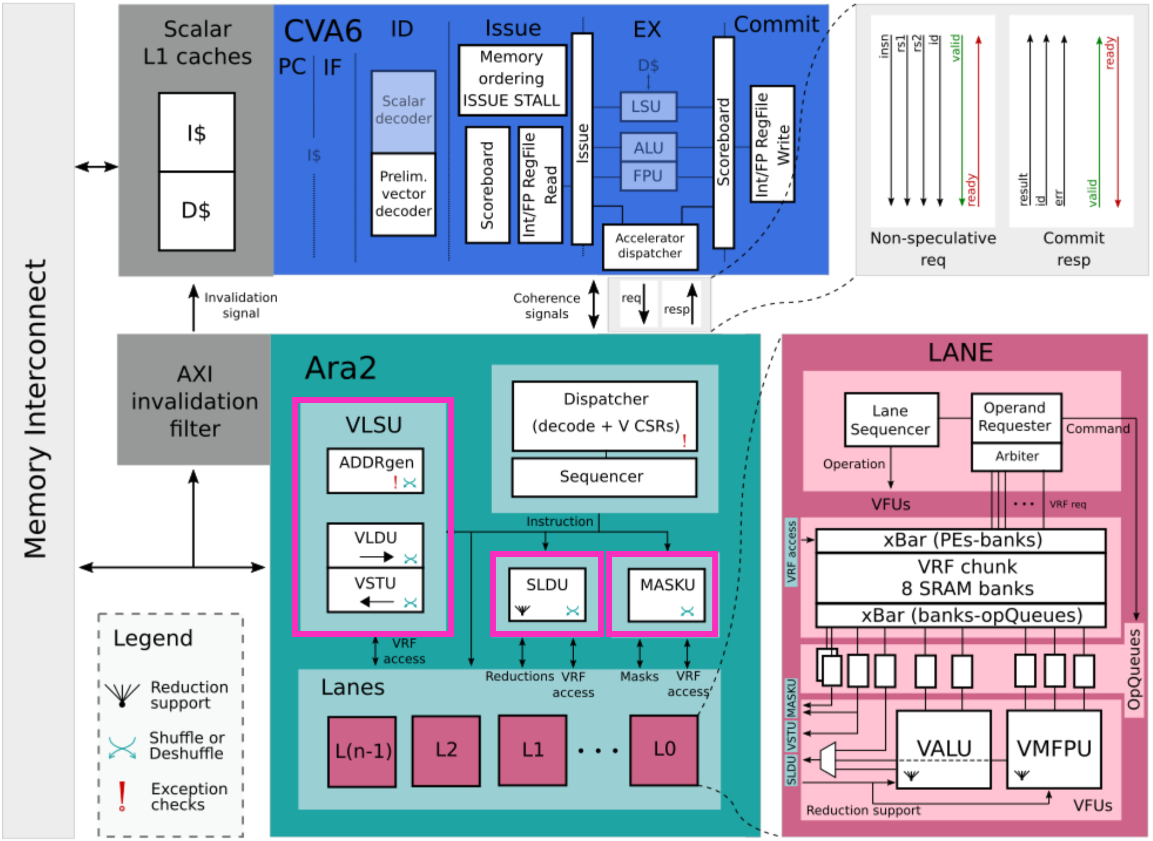
Mask bits aligned with elements

Mask bits compressed

Need for a dedicated unit that packs-unpacks mask bits - MASKU

From Ara1 to Ara2 – ISA update

- **Some units** access the **whole VRF**
- Each unit is **connected to all the lanes** (in and out)
- **All-to-all** connected **units:**
  - VLSU
  - SLDU
  - **MASKU** (new!)
- **All-to-all** connected units **complicate rounting**



Coherence/ordering protocol

Before

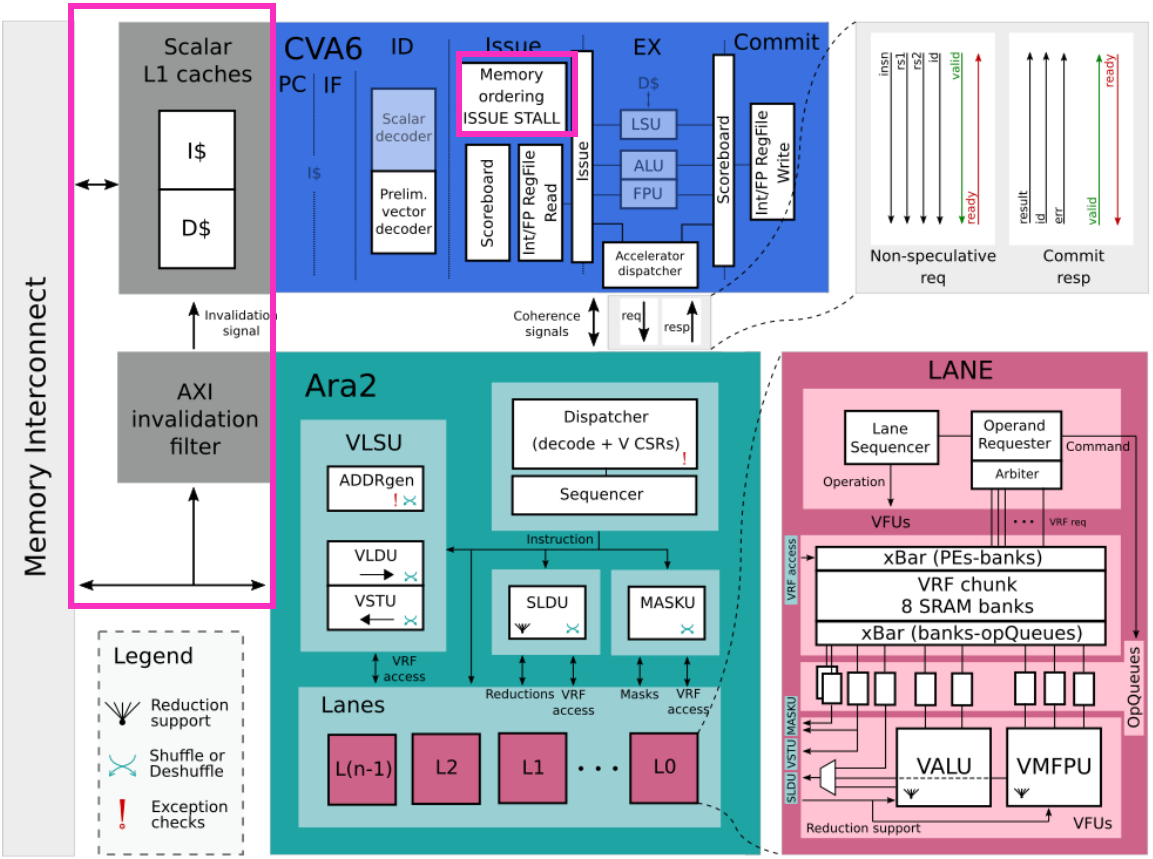
- No hardware solution
- Need for software fences

Memory Ordering

- If vector store is ongoing block scalar loads/stores
- If scalar store ongoing block vector loads/stores
- If scalar/vector load is ongoing block vector/scalar stores

Cache coherence

- Invalidate scalar cache lines upon vector store



ETHzürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA Future Computing Laboratory

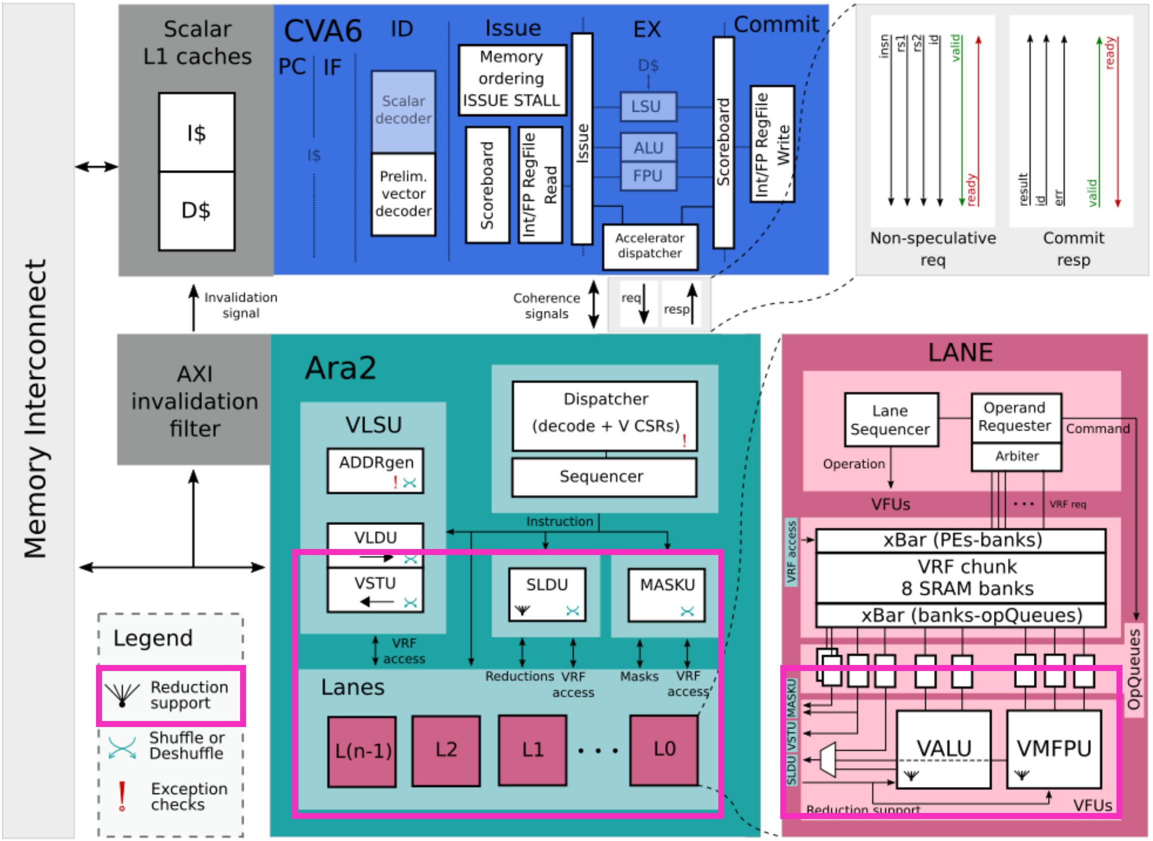
Vector Reductions

Before

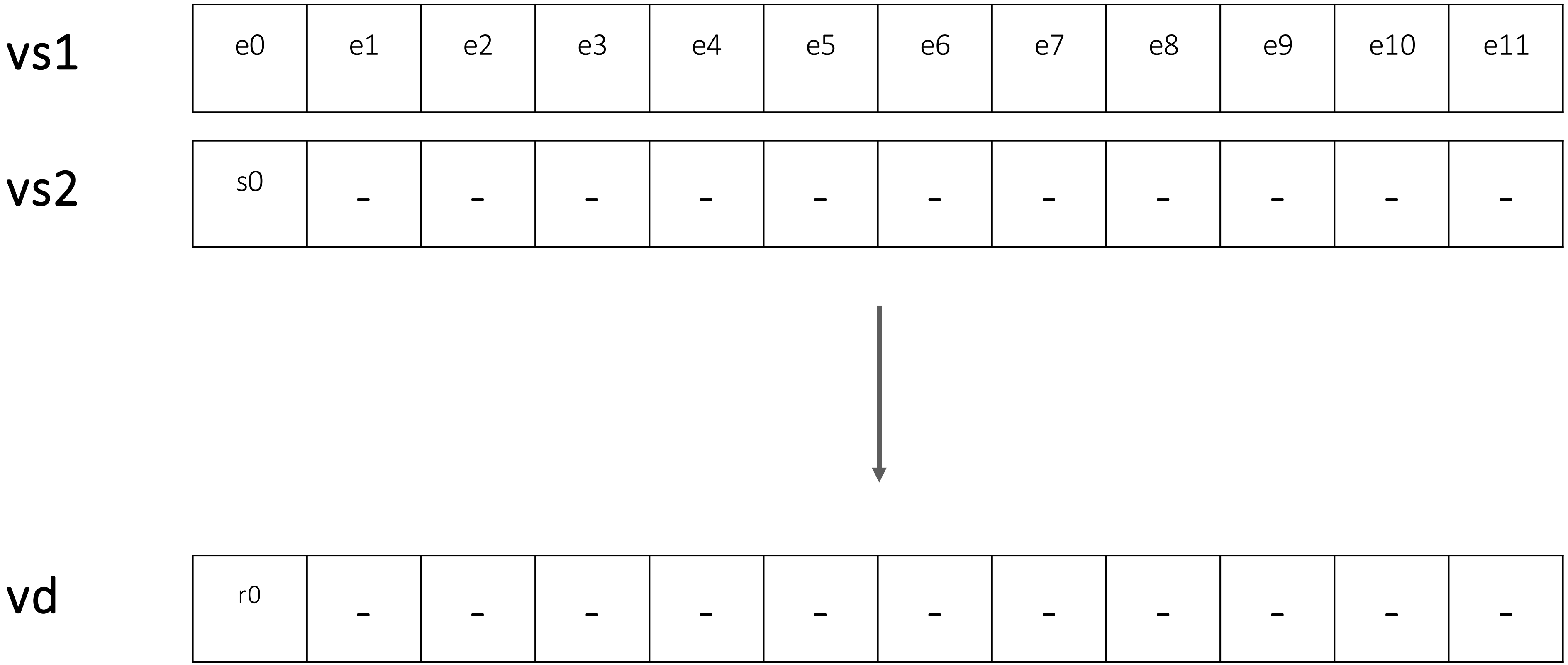
- No vector reductions

After the update

- Integer reduction
- Floating-Point (FP) reduction
- Ordered FP reduction



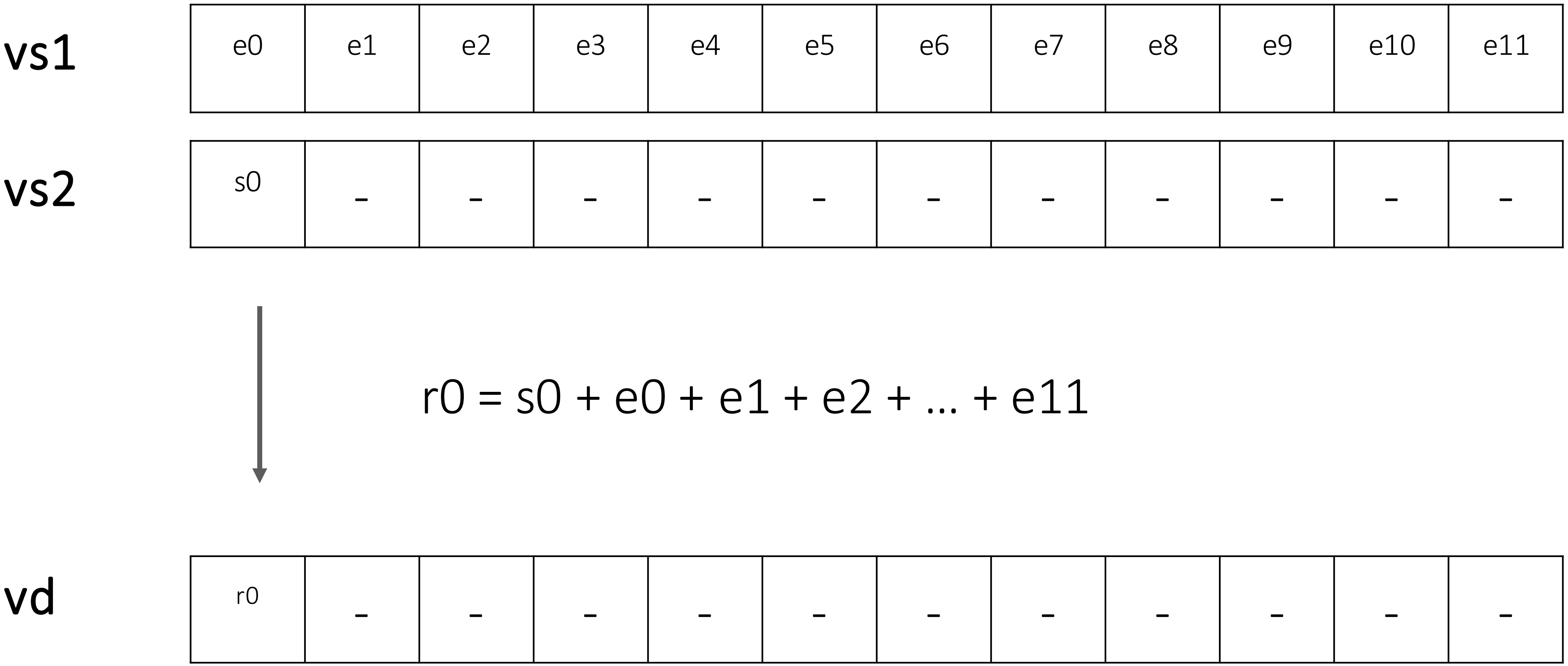
Vector Reductions



vs1	e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11
vs2	s0	_	_	_	_	_	_	_	_	_	_	_
vd	r0	_	_	_	_	_	_	_	_	_	_	_

Vector Reductions

Example: vredsum vd, vs1, vs2



vs1	e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11
vs2	s0	-	-	-	-	-	-	-	-	-	-	-

r0 = s0 + e0 + e1 + e2 + … + e11

vd	r0	-	-	-	-	-	-	-	-	-	-	-

Vector Reductions

Example: vredsum vd, vs1, vs2

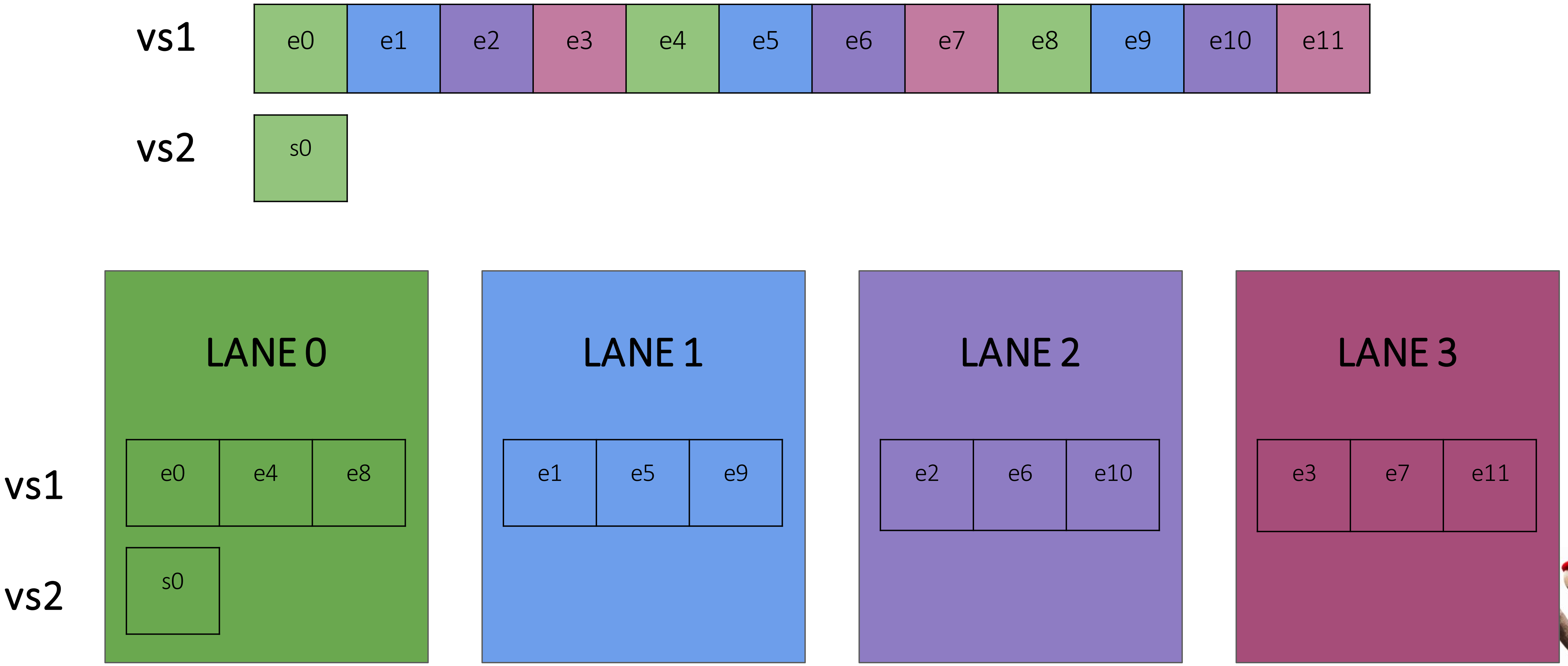
vs1	e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11
vs2	s0	-	-	-	-	-	-	-	-	-	-	-

↓

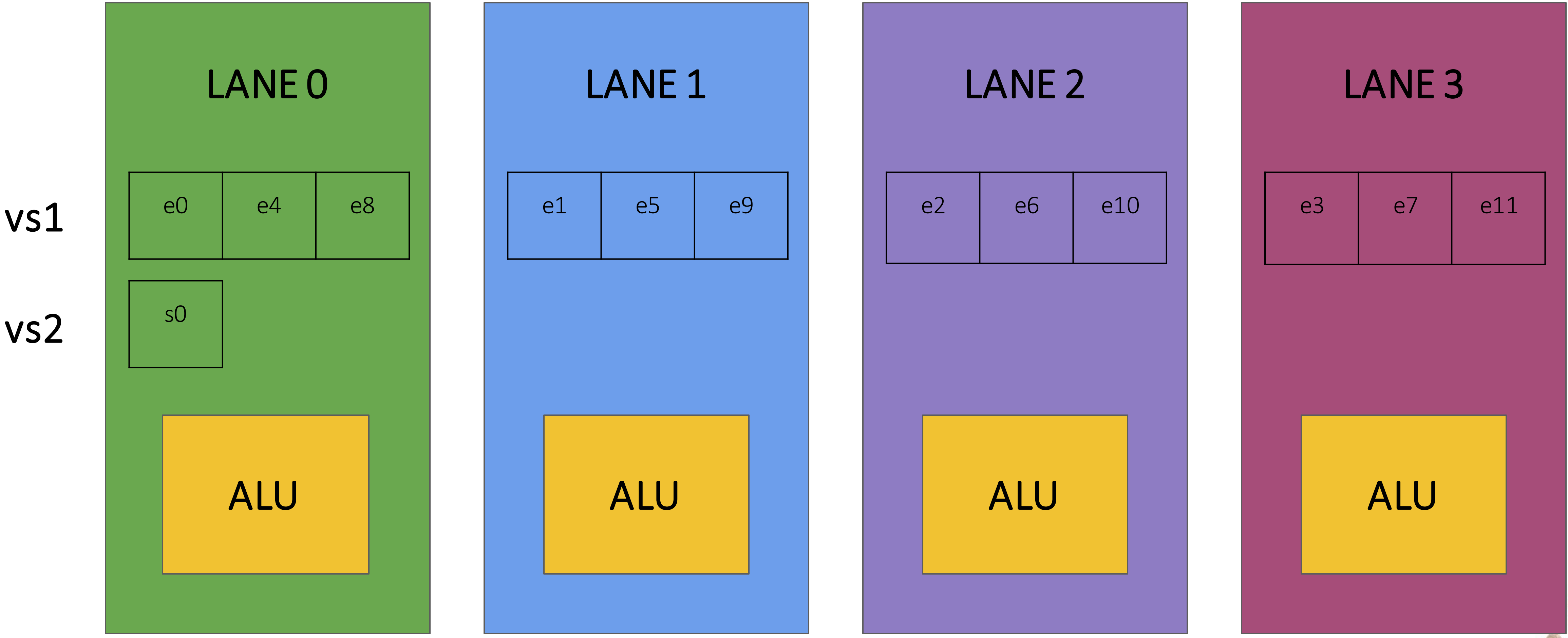
r0 = s0 + e0 + e1 + e2 + … + e11

vd	r0	-	-	-	-	-	-	-	-	-	-	-

Vector Reductions – Intra-lane phase start



Vector Reductions



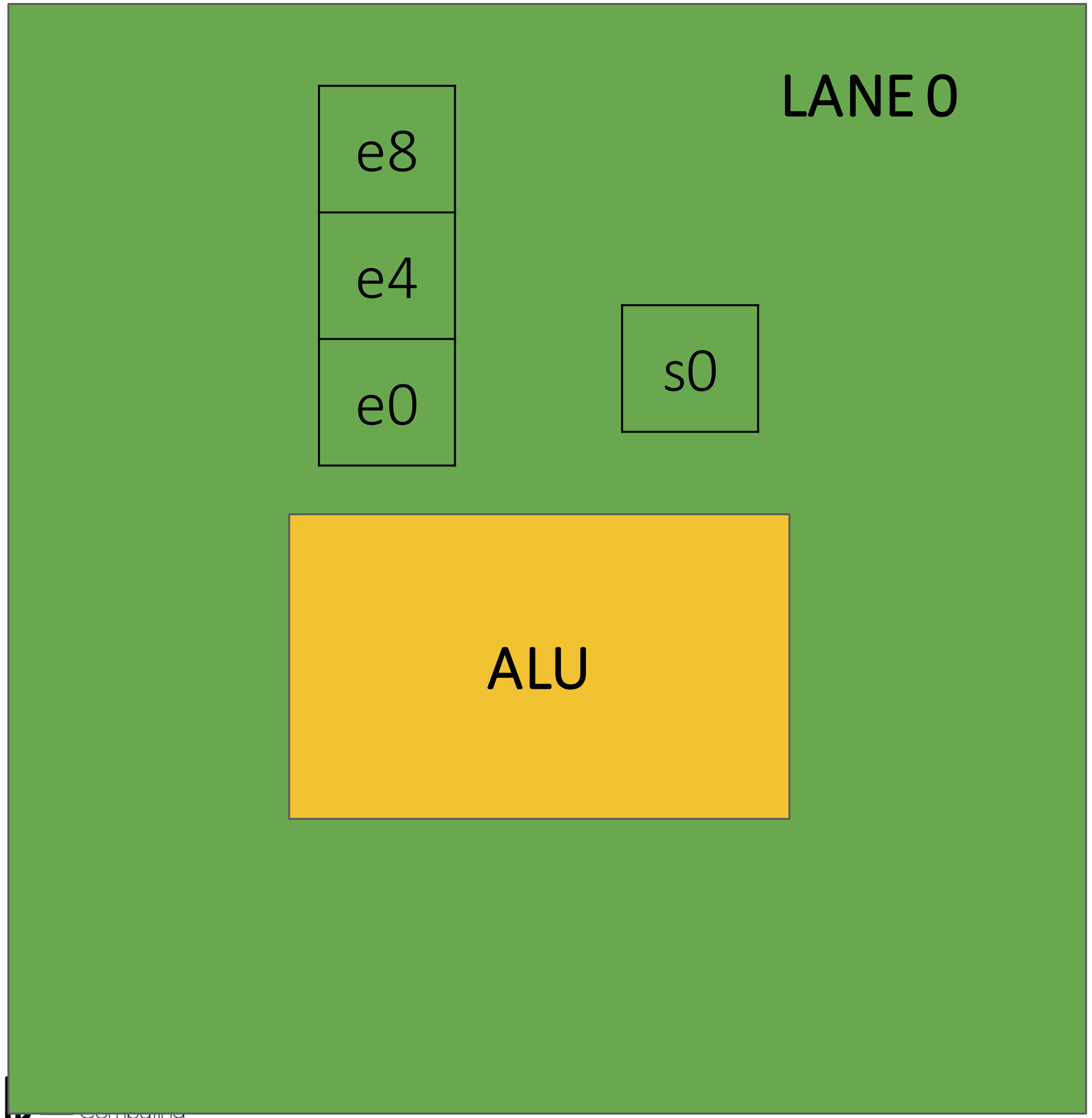
temp0 = s0 + e0 + e4 + e8

temp1 = e1 + e5 + e9

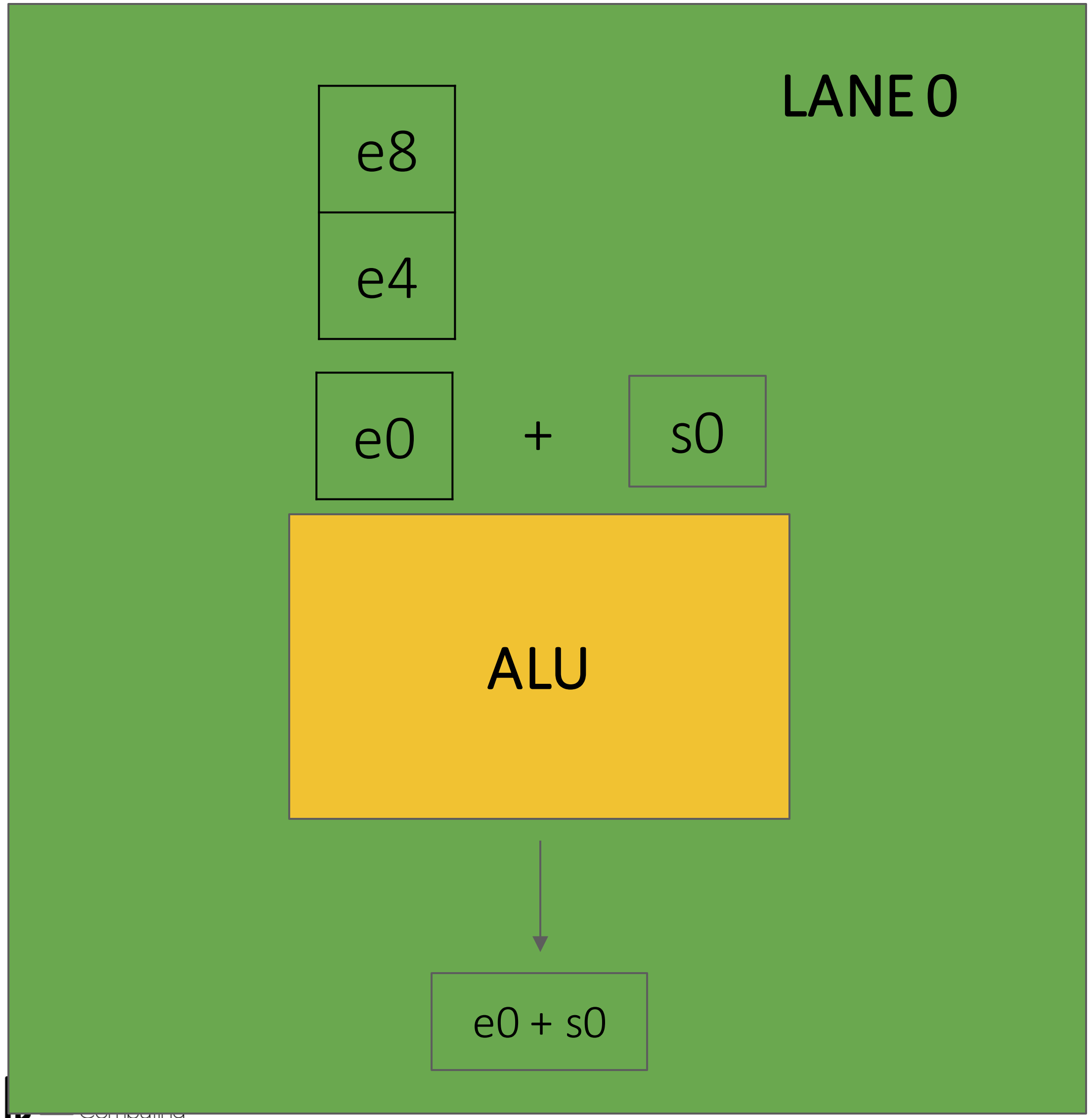
temp2 = e2 + e6 + e10

temp3 = e3 + e7 + e11

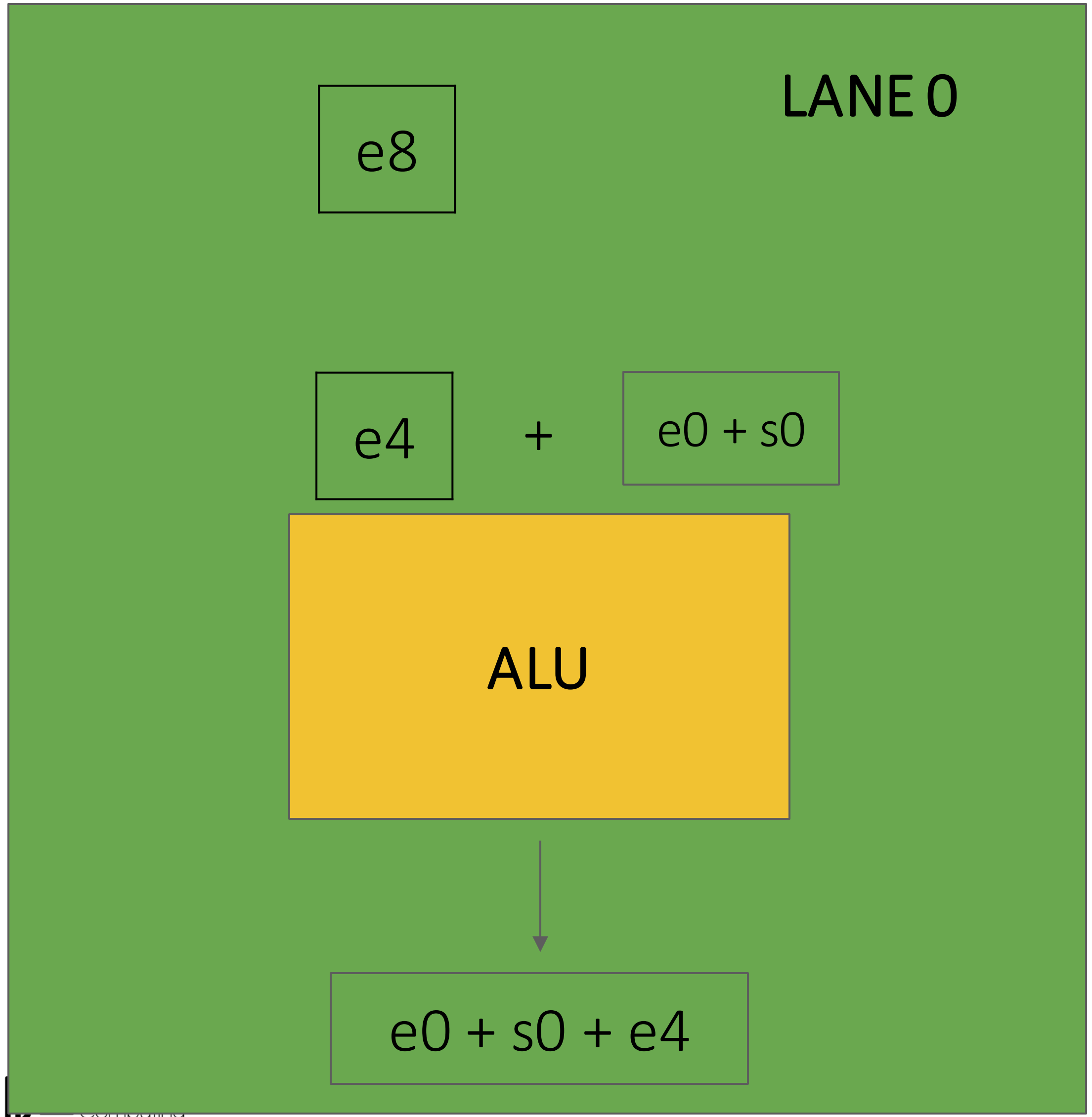
Vector Reductions



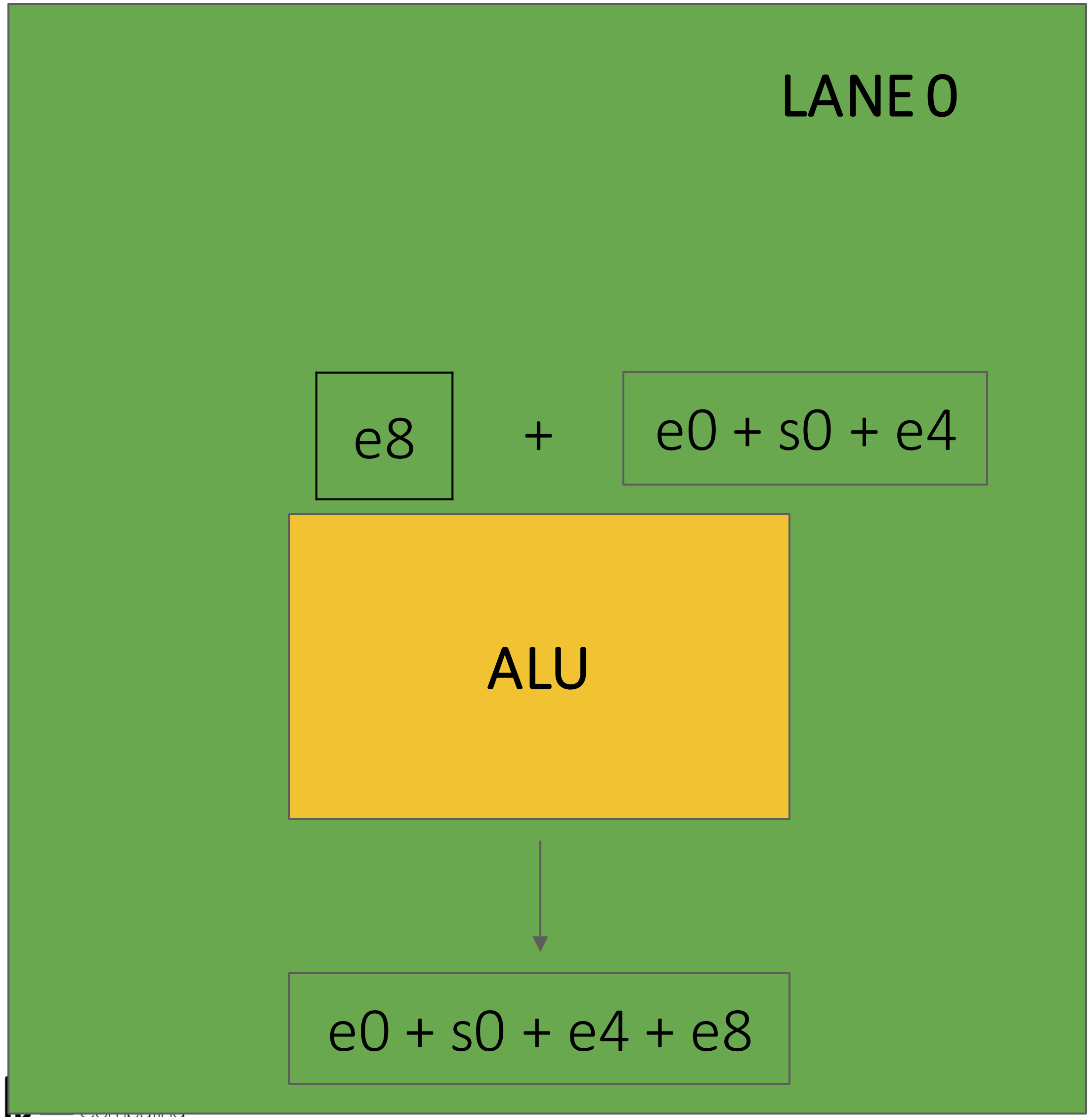
Vector Reductions



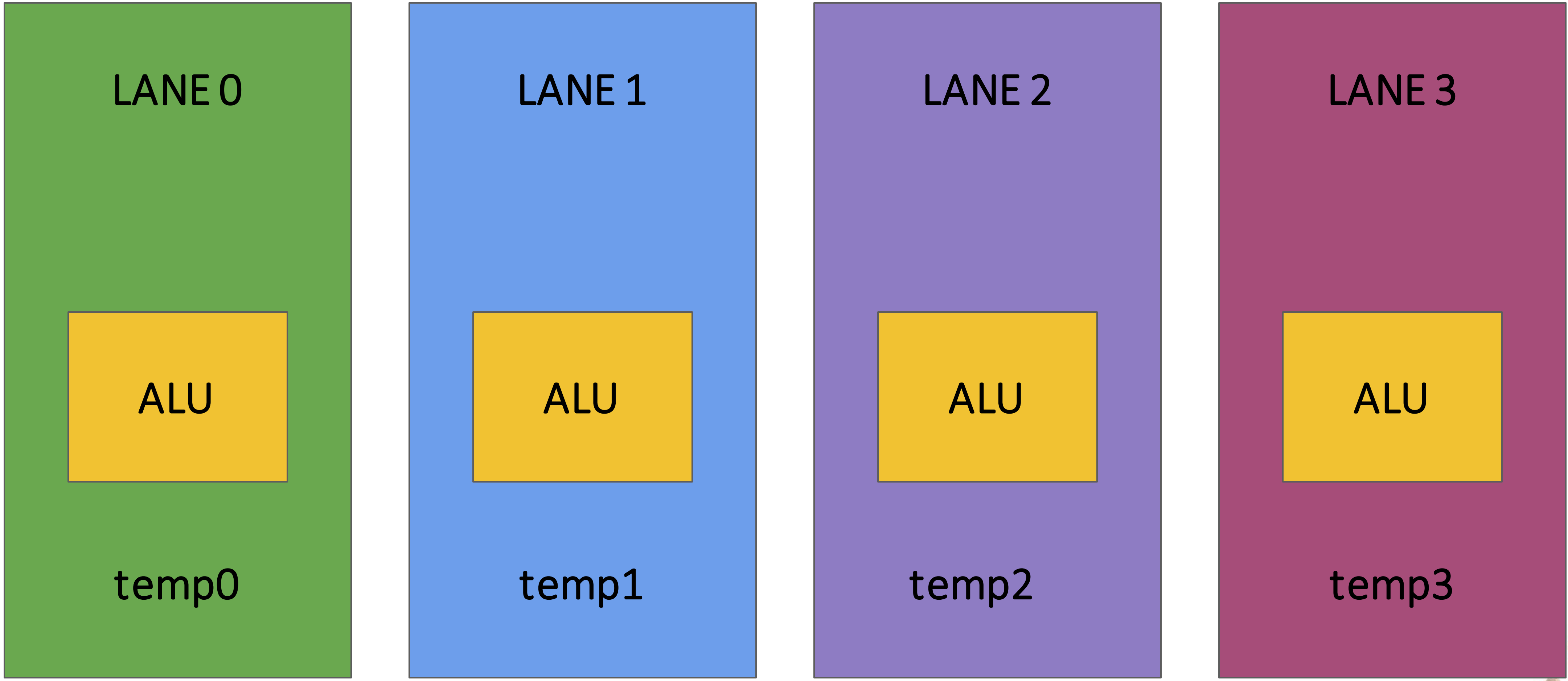
Vector Reductions



Vector Reductions



Vector Reductions – Intra-Lane phase end



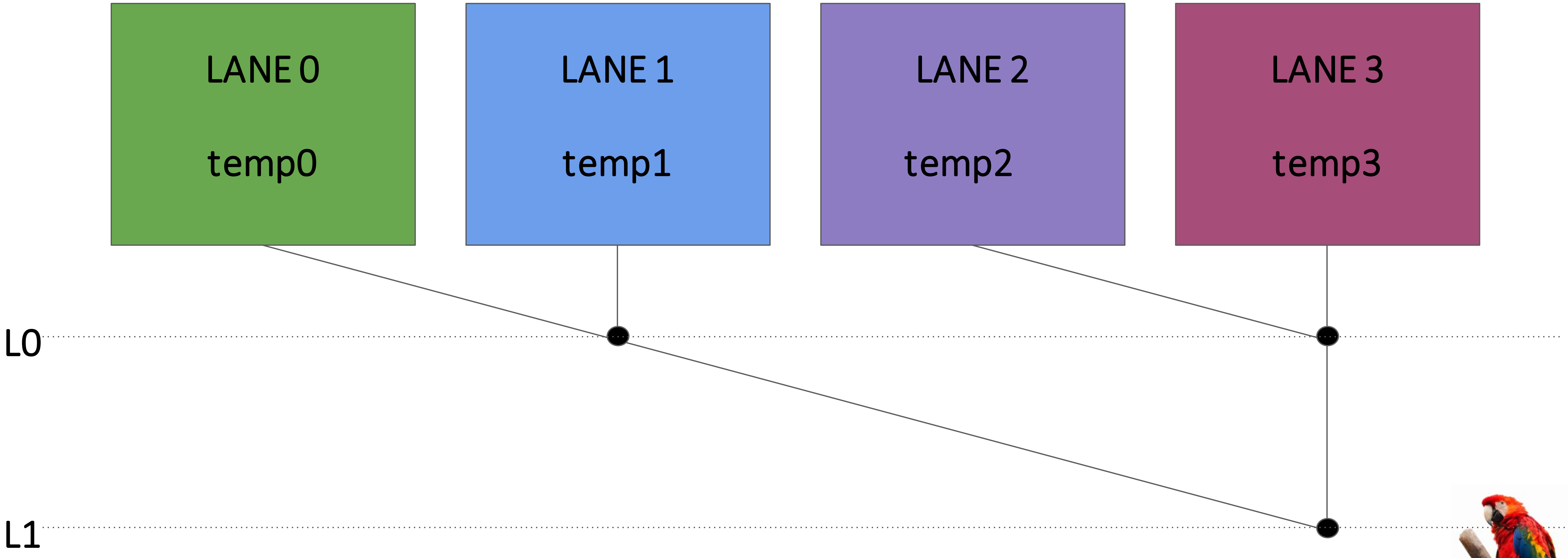
temp0 = s0 + e0 + e4 + e8

temp1 = e1 + e5 + e9

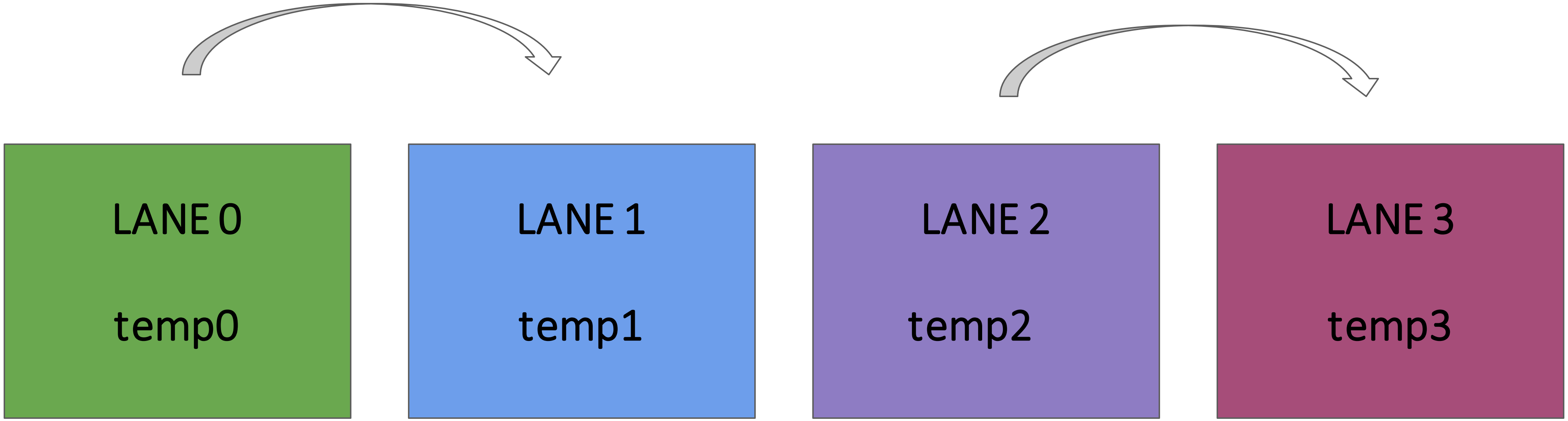
temp2 = e2 + e6 + e10

temp3 = e3 + e7 + e11

Vector Reductions – Inter-Lane phase start



Vector Reductions



LANE 0

temp0

LANE 1

temp1

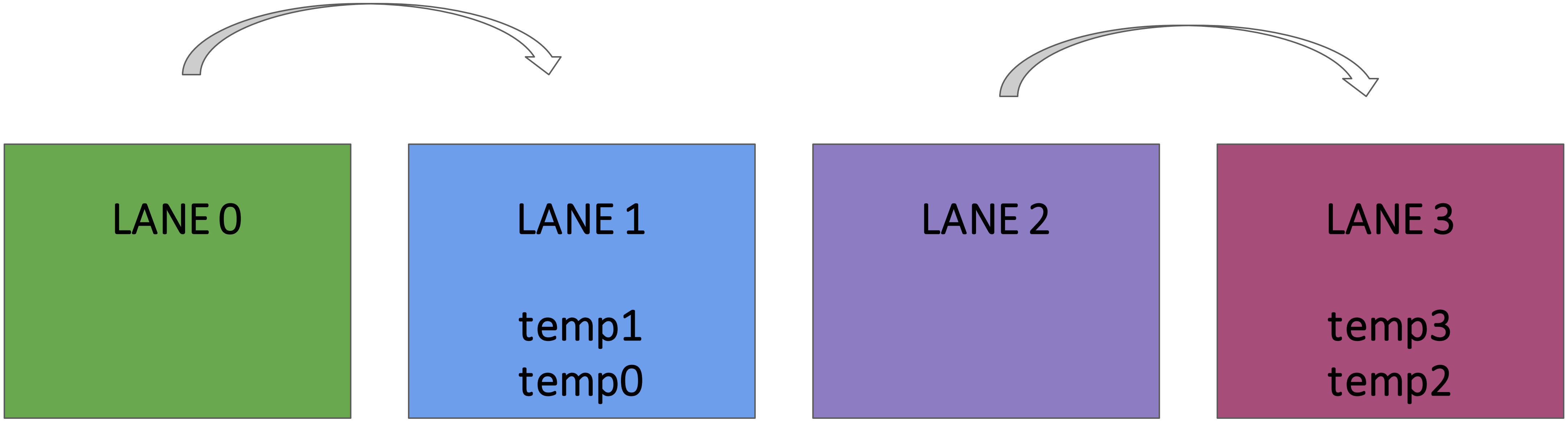
LANE 2

temp2

LANE 3

temp3

Vector Reductions



LANE 0

LANE 1

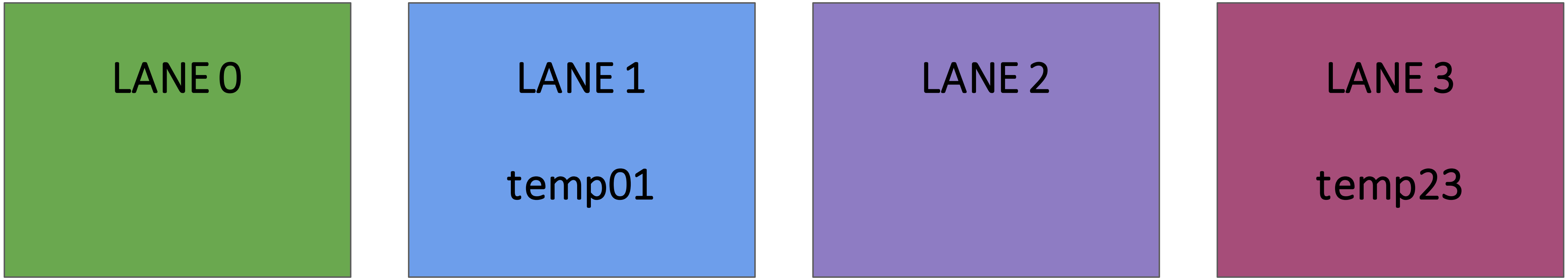
temp1
temp0

LANE 2

LANE 3

temp3
temp2

Vector Reductions


LANE 0

LANE 1

temp01

LANE 2

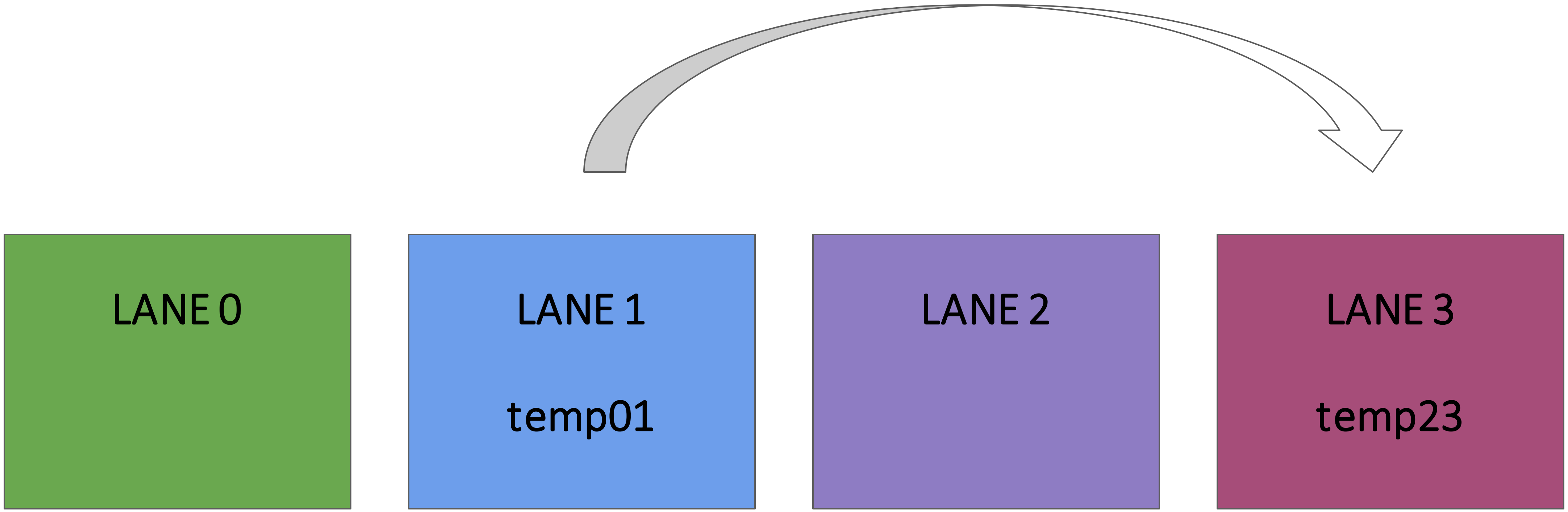
LANE 3

temp23

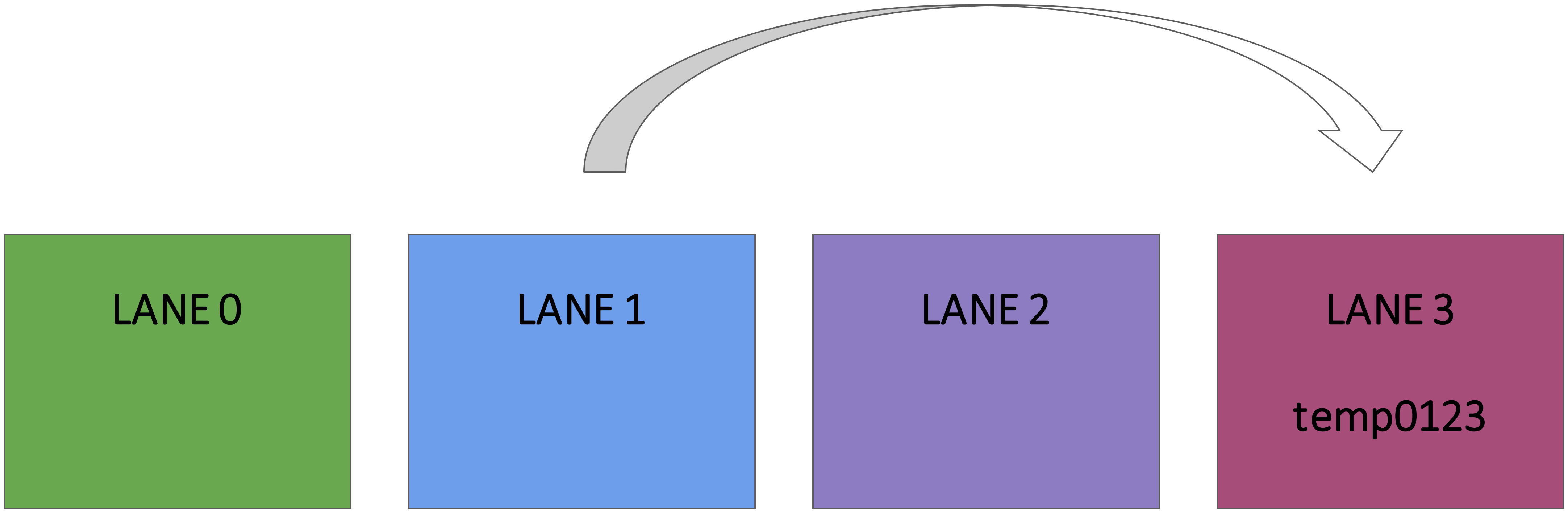
temp01 = temp0 + temp1

temp23 = temp2 + temp3

Vector Reductions

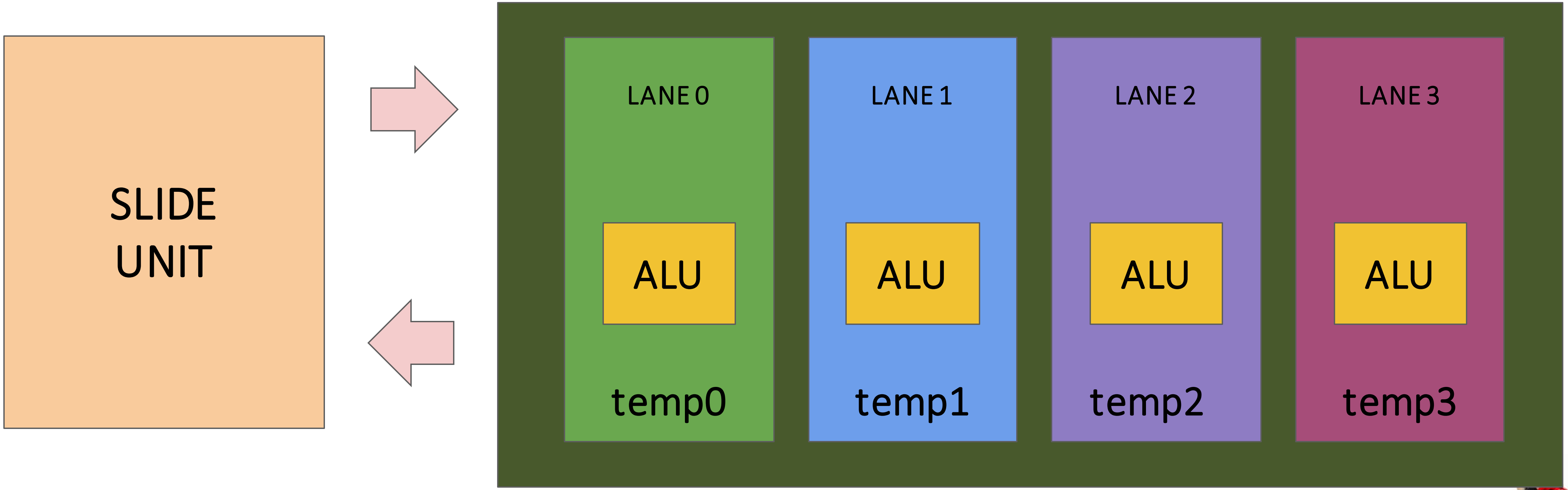


Vector Reductions

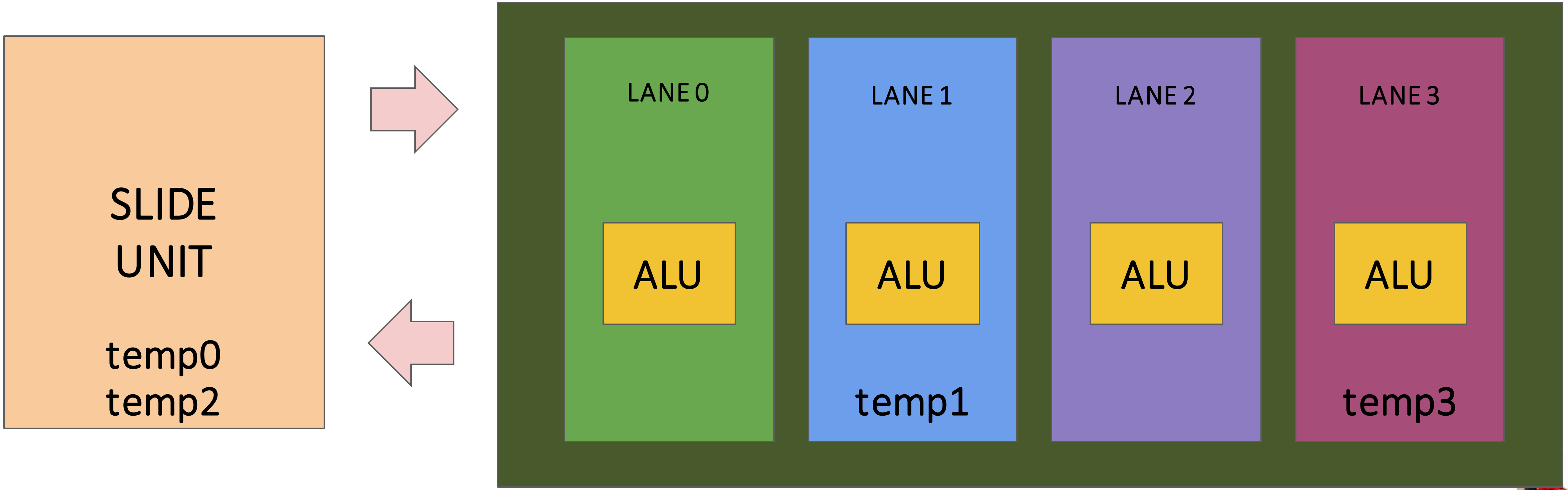


temp0123 = temp01 + temp23

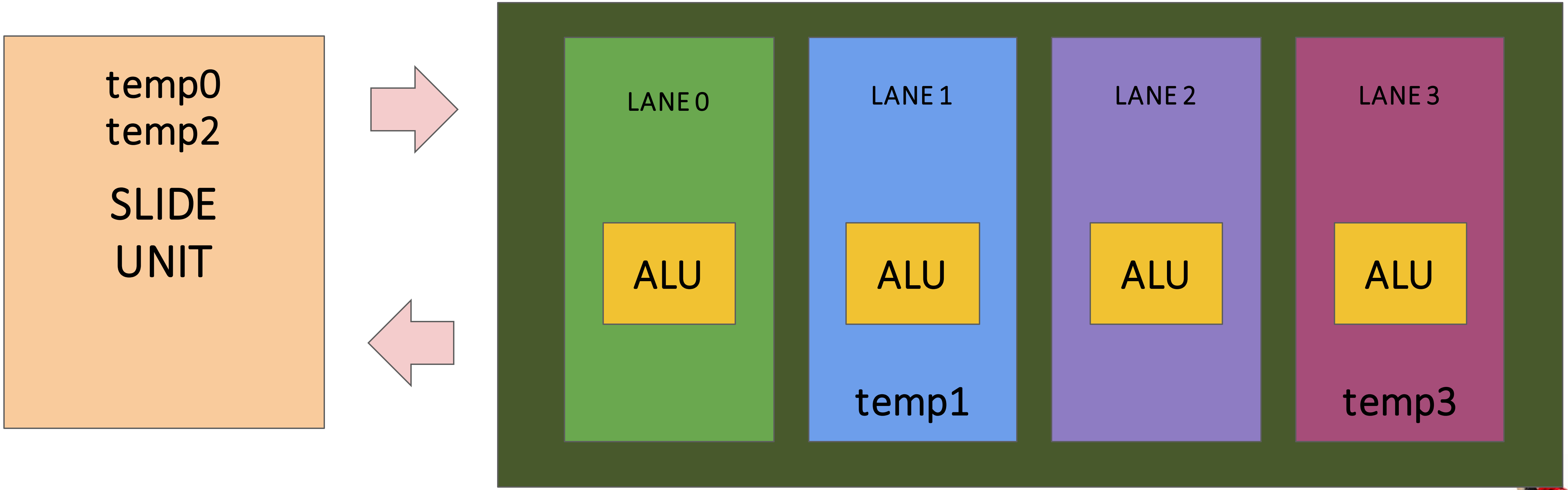
Vector Reductions



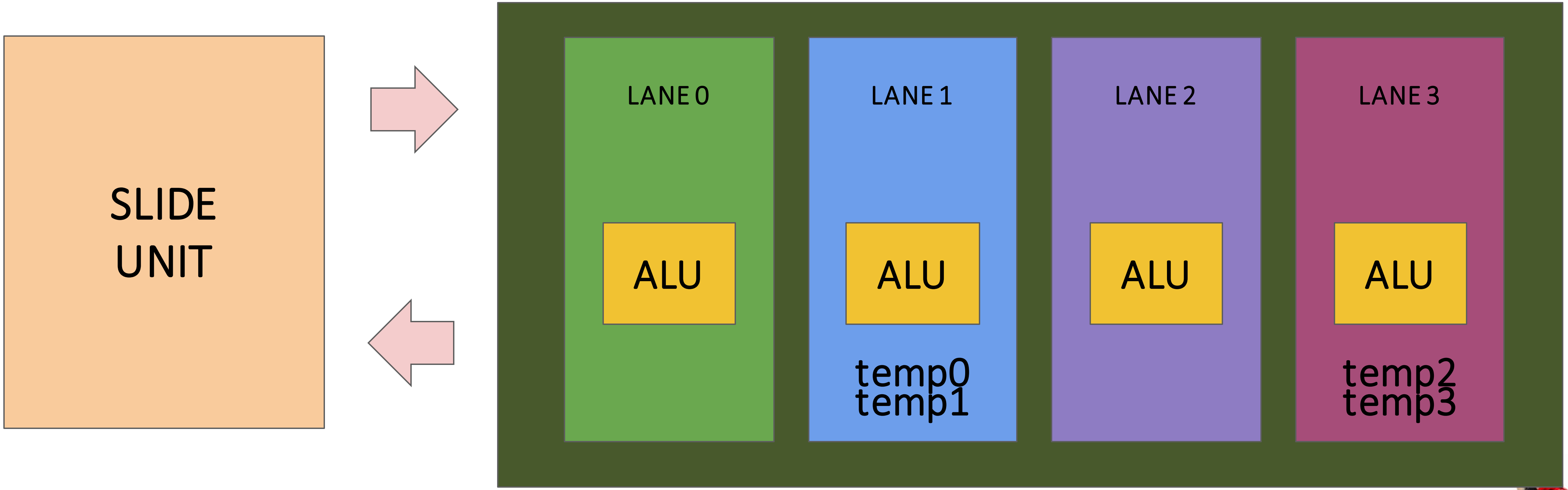
Vector Reductions



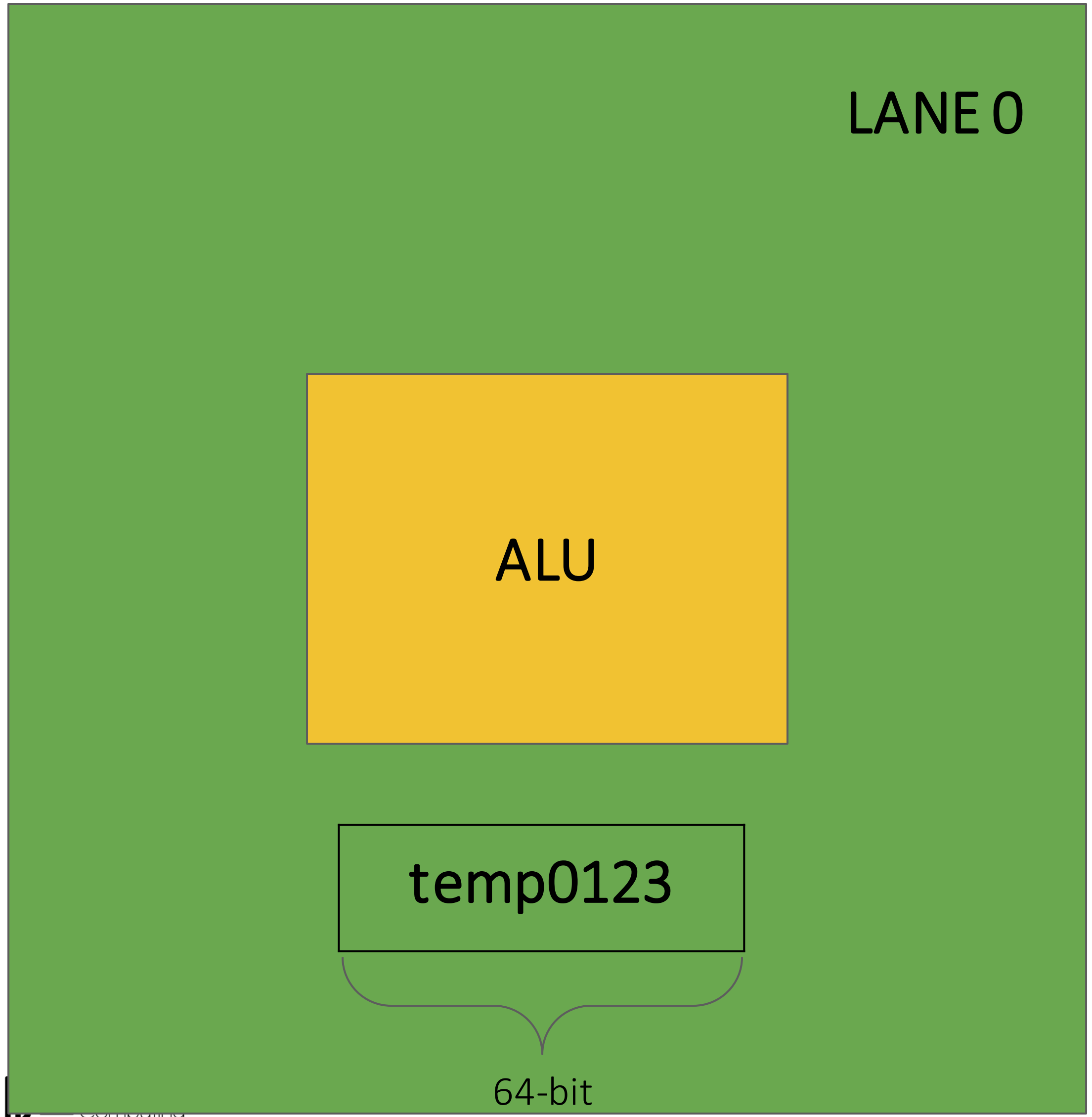
Vector Reductions



Vector Reductions – Inter-Lane phase end

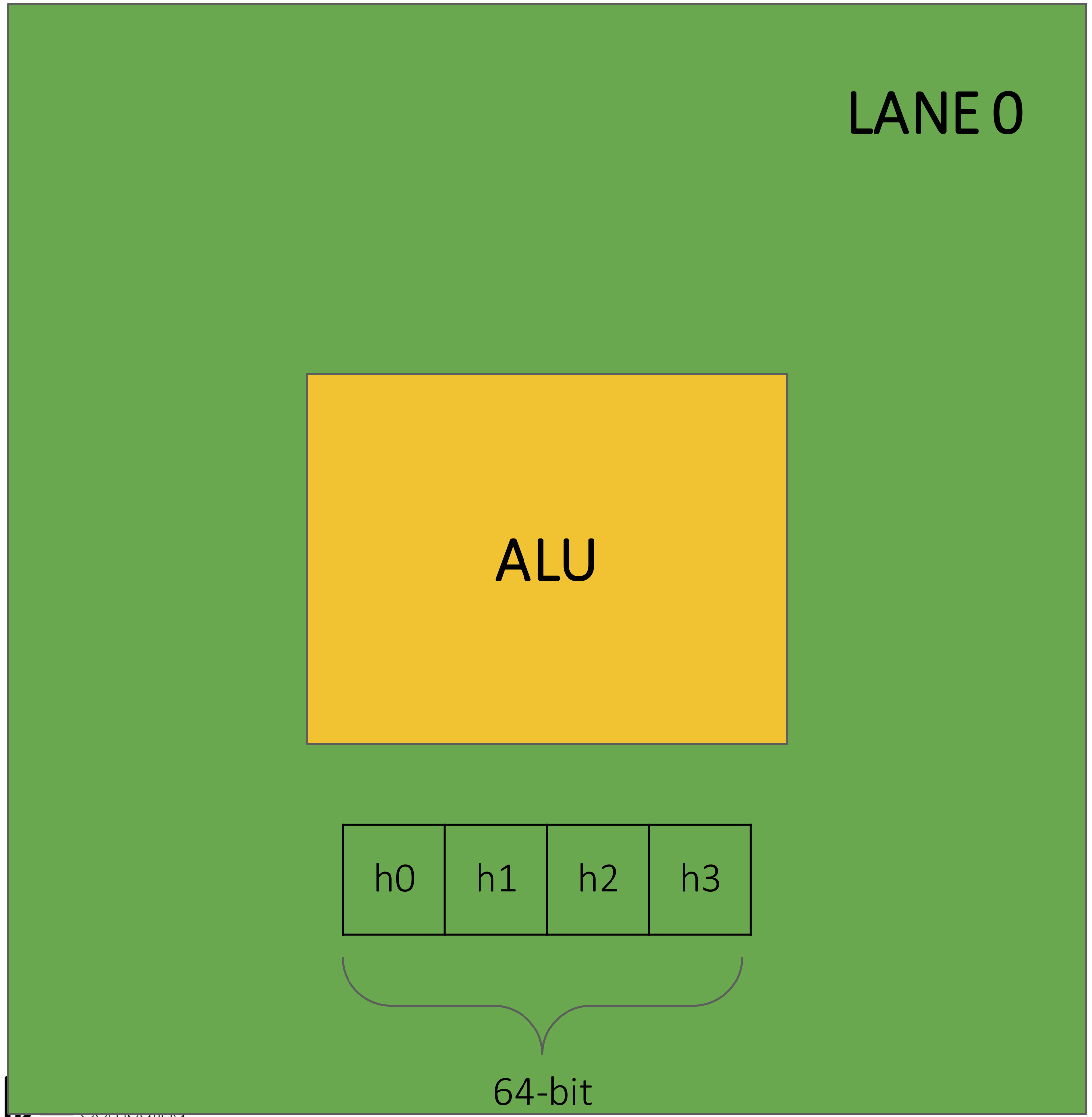


Vector Reductions – SIMD phase start



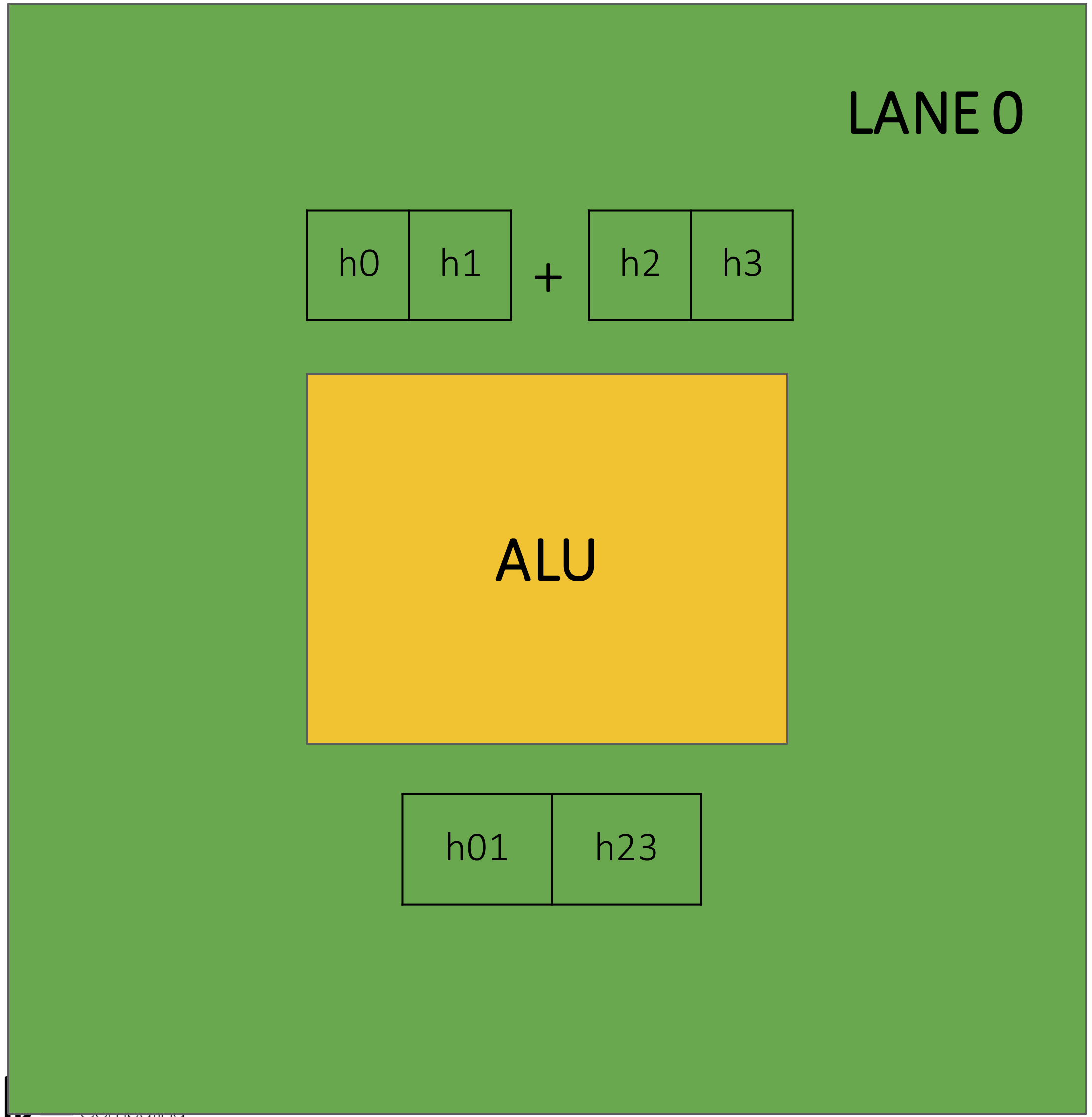
Vector Reductions

- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



Vector Reductions

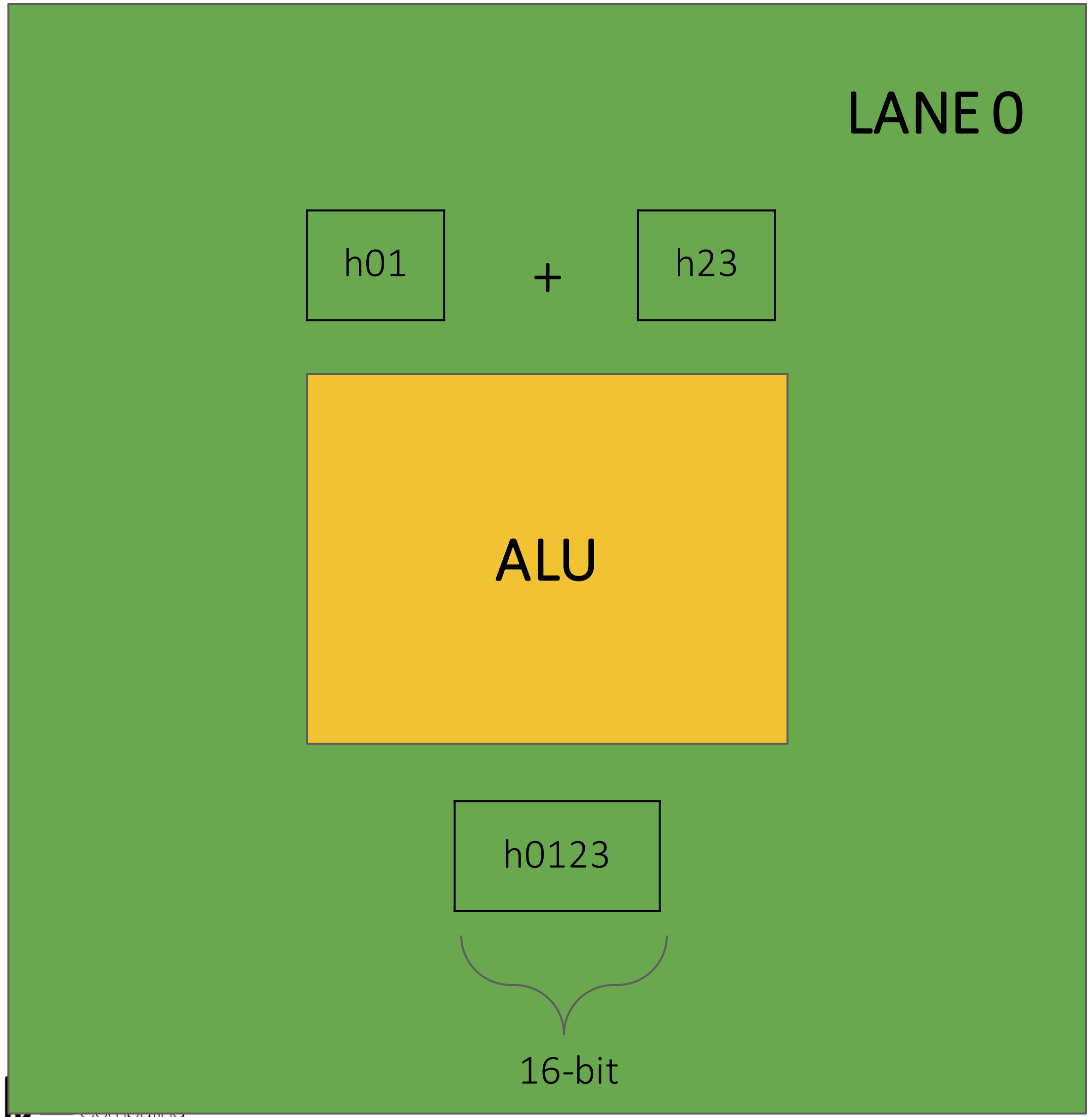
- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



First operation

Vector Reductions – SIMD phase end

- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



Second operation

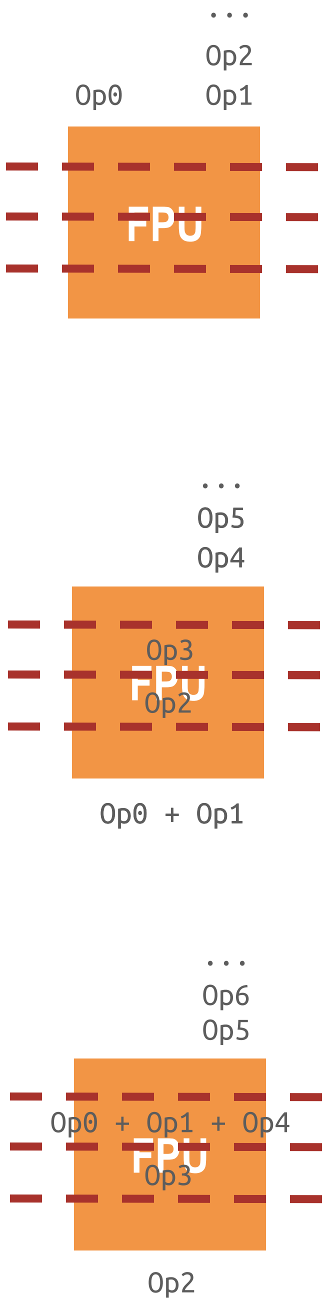
Floating-point reductions?

- FPUs are pipelined! Pipeline levels (R) depend on SEW
- Wait R cycles before a new operation?
- Trick: use pipeline registers to store partial accumulators
- In the end: final round of accumulation
- Intra-Lane reduction latency

$$\frac{N}{L} + R \times (1 + \log_2 \lceil R \rceil) - (\lceil R \rceil - R)$$

- For R power of 2 (superlinear)

$$\frac{N}{L} + R \times (1 + \log_2 (R))$$



Verification and Software Eco-System

Verification

- Update and extend riscv-tests repository
- Every instruction has its own set of dedicated tests
- Multiple self-verifying benchmarks implemented
- Additional tests from external company
- CI integrated on GitHub

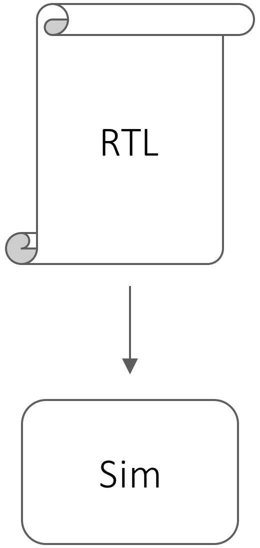
Software eco-system

- LLVM compiler + SPIKE updated more than once to follow specs update
- Vector intrinsics support - Still in evolution from RISC-V community!
- Verilator (open-source) to simulate the system
- Initial support for auto-vectorization

Evaluation

Benchmarks

- Linear Algebra, math: fmatmul, fconv2d, jacobi2d, dotp, exp, log, cos
- Routing Algorithm: pathfinder
- DSP: fft, dwt
- ML: softmax, roi_align, dropout
- AWB, spmv, ReLu, ... : not in the analysis



Throughput

- vs. number of lanes
  (2, 4, 8, 16)
- vs. vector length
  (4, 8, 16, 32, 64, 128 elements)

How scalar core and memory system influence performance?

Evaluation

Benchmarks

- LA: fmatmul, fconv2d, jacobi2d, dotp, exp
- RA: pathfinder
- DSP: fft, dwt
- ML: softmax, roi_align, dropout

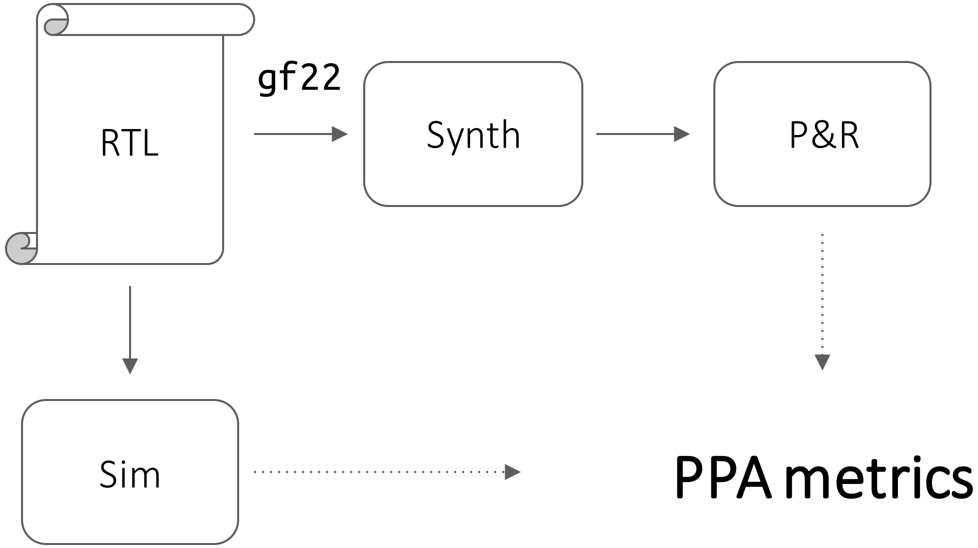
Throughput

- vs. number of lanes
- vs. vector length

How scalar core and memory system influence performance?

Physical implementation

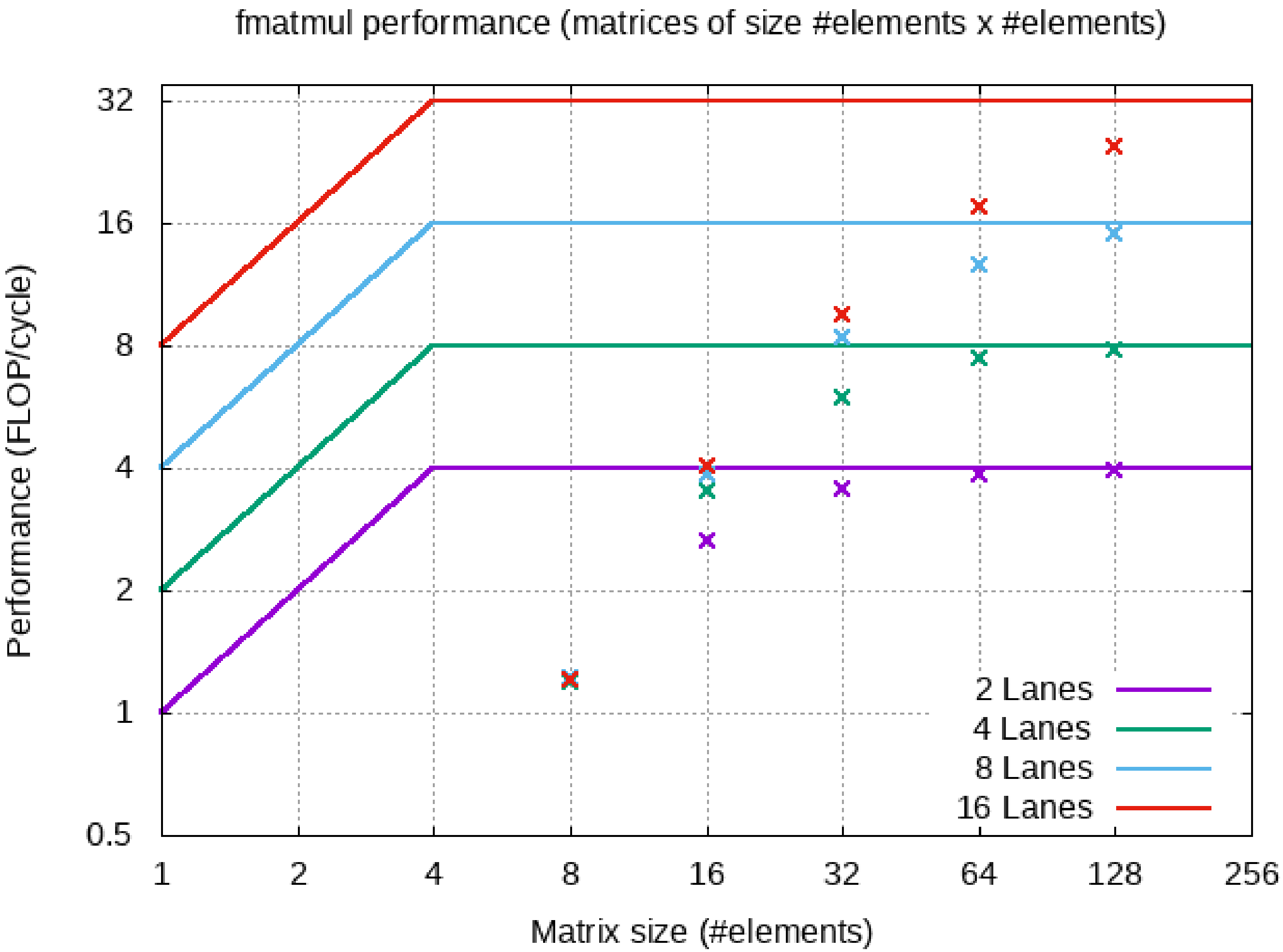
- PPA metrics
- Efficiency study
- Scaling analysis



System performance – A focus on matmul

- Up to **32 DP-FLOP/cycle**
  - With 16 Lanes

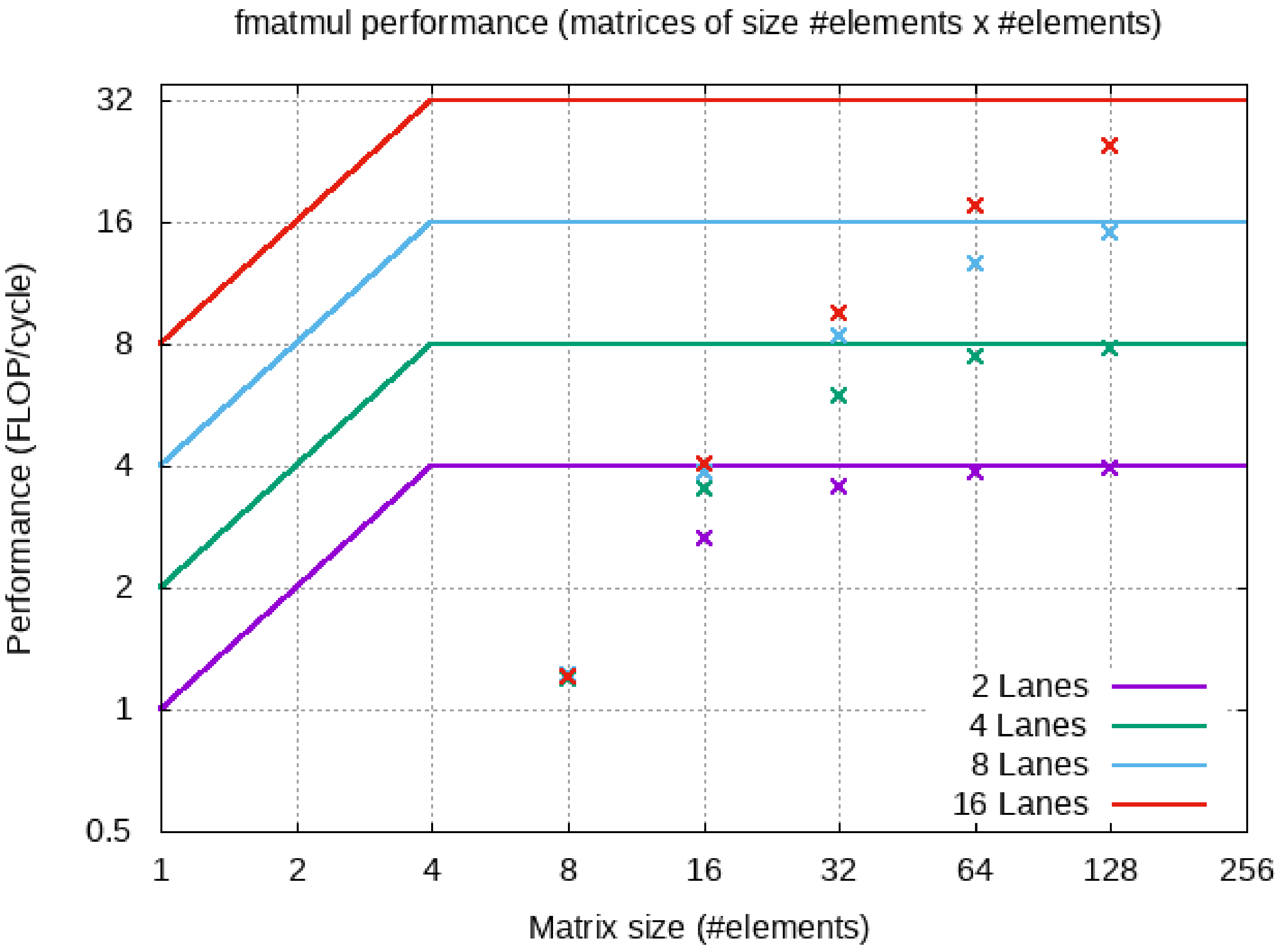
- FP-matmul
  - A[N][N] * B[N][N]



System performance – A focus on matmul

- Up to **32 DP-FLOP/cycle**
  - With 16 Lanes

- FP-matmul
  - A[N][N] * B[N][N]
  - X-axis: Matrix Size N

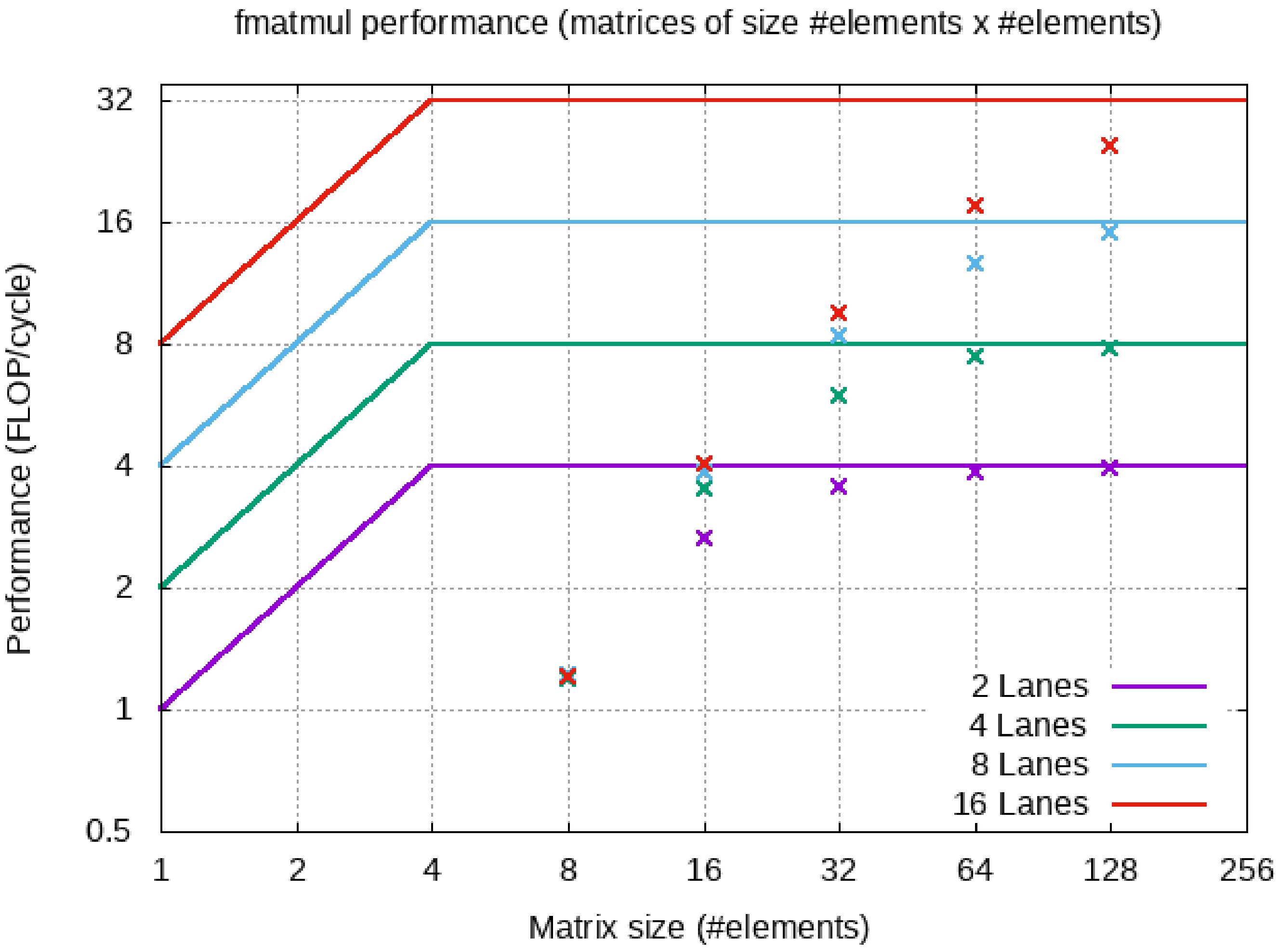


Increasing arithmetic intensity!

System performance – A focus on matmul

- Up to **32 DP-FLOP/cycle**
  - With 16 Lanes

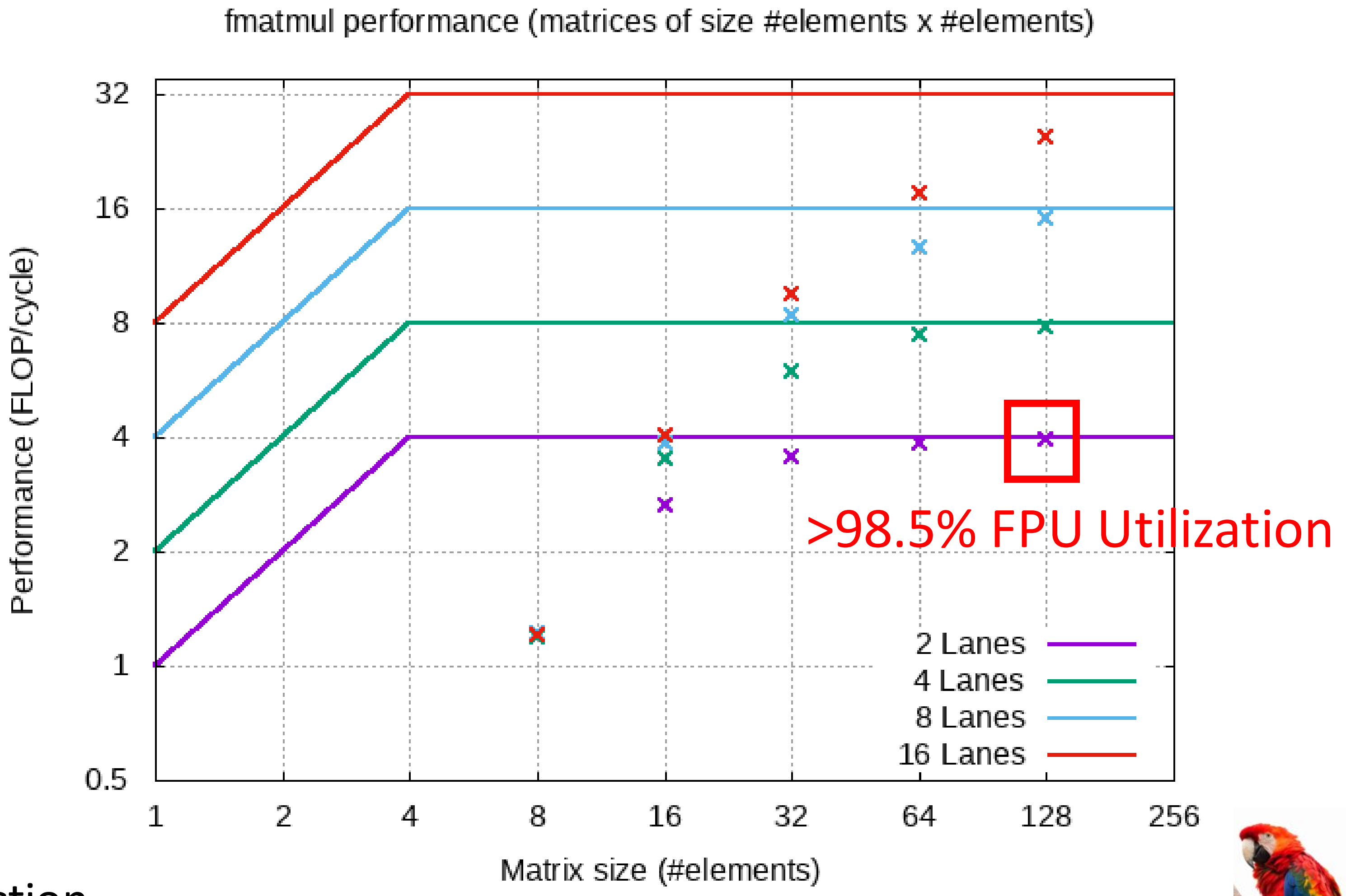
- FP-matmul
  - A[N][N] * B[N][N]
  - X-axis: Matrix Size N
  - Always computation-bound



Increasing arithmetic intensity!

System performance – A focus on matmul

- Up to **32 DP-FLOP/cycle**
  - With 16 Lanes
- FP-matmul
  - A[N][N] * B[N][N]
  - X-axis: Matrix Size N
  - Always computation-bound
- Longer vector, better performance
  - Amortize setup time for V instruction
  - Amortize CVA6 non-idealities



Issue-rate limitation

- There is another intrinsic limitation to performance
- Issue-rate limitation:
  - 4-element vector
  - 4-lane Ara (4 FPUs)
  - Max Throughput = 1 vfmacc/cycle
  - To reach full performance, Ara would need 1 `vfmacc` per cycle!

```
// Loop kernel is a sequence of:
  // bump pointer
  add addr, 8
  // load scalar from mtx A
  ld x, @addr
  // issue vfmacc
  vfmacc v0, v1, x
```

BUT, in the best case:

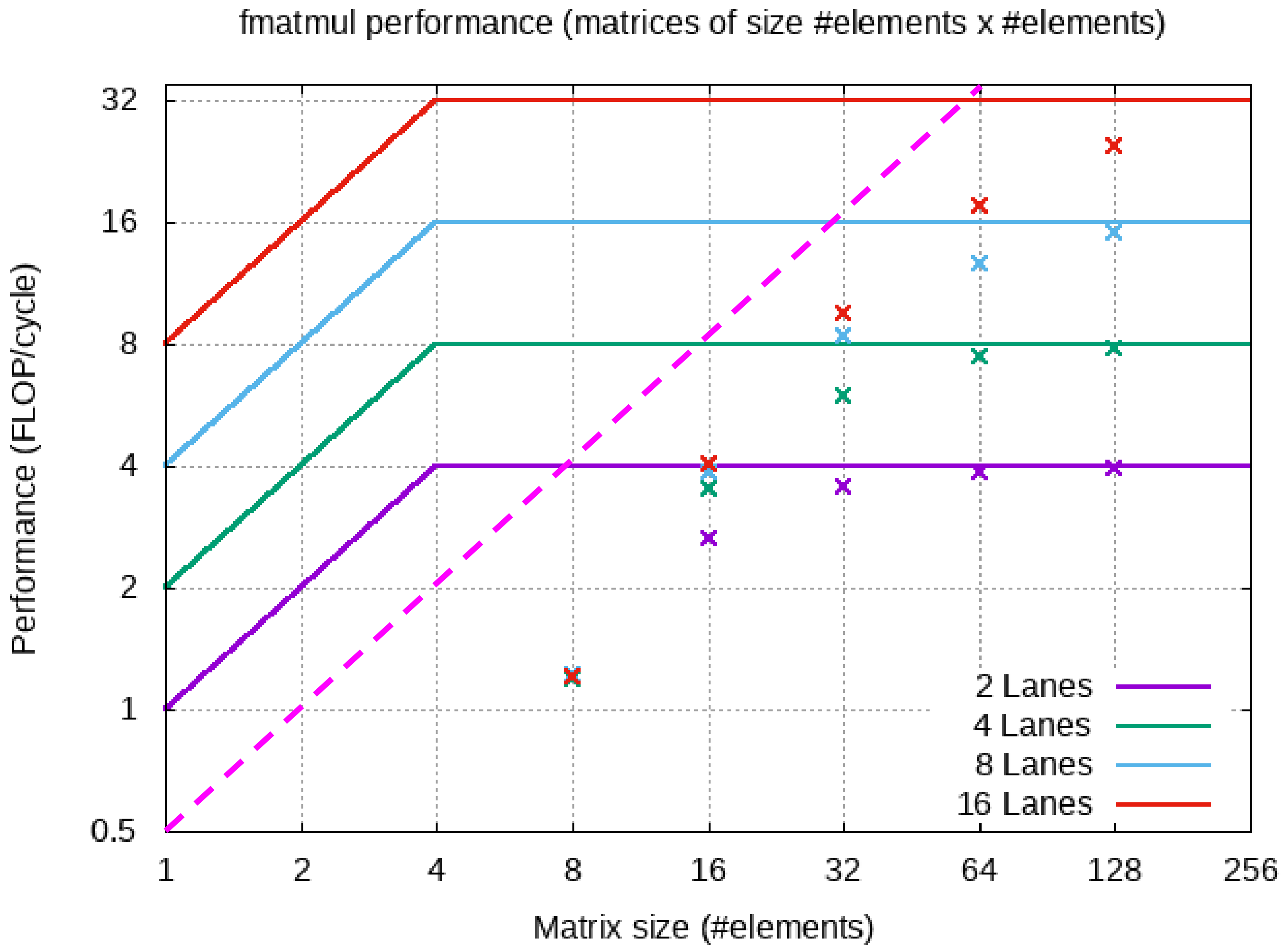
1 `vfmacc` every 3 cycles

System performance – A focus on matmul

Issue-rate limitation hits short/medium-vectors

Longer vectors hide CVA6's non-ideal issue-rate latency

i.e., with longer vectors, Ara does not starve



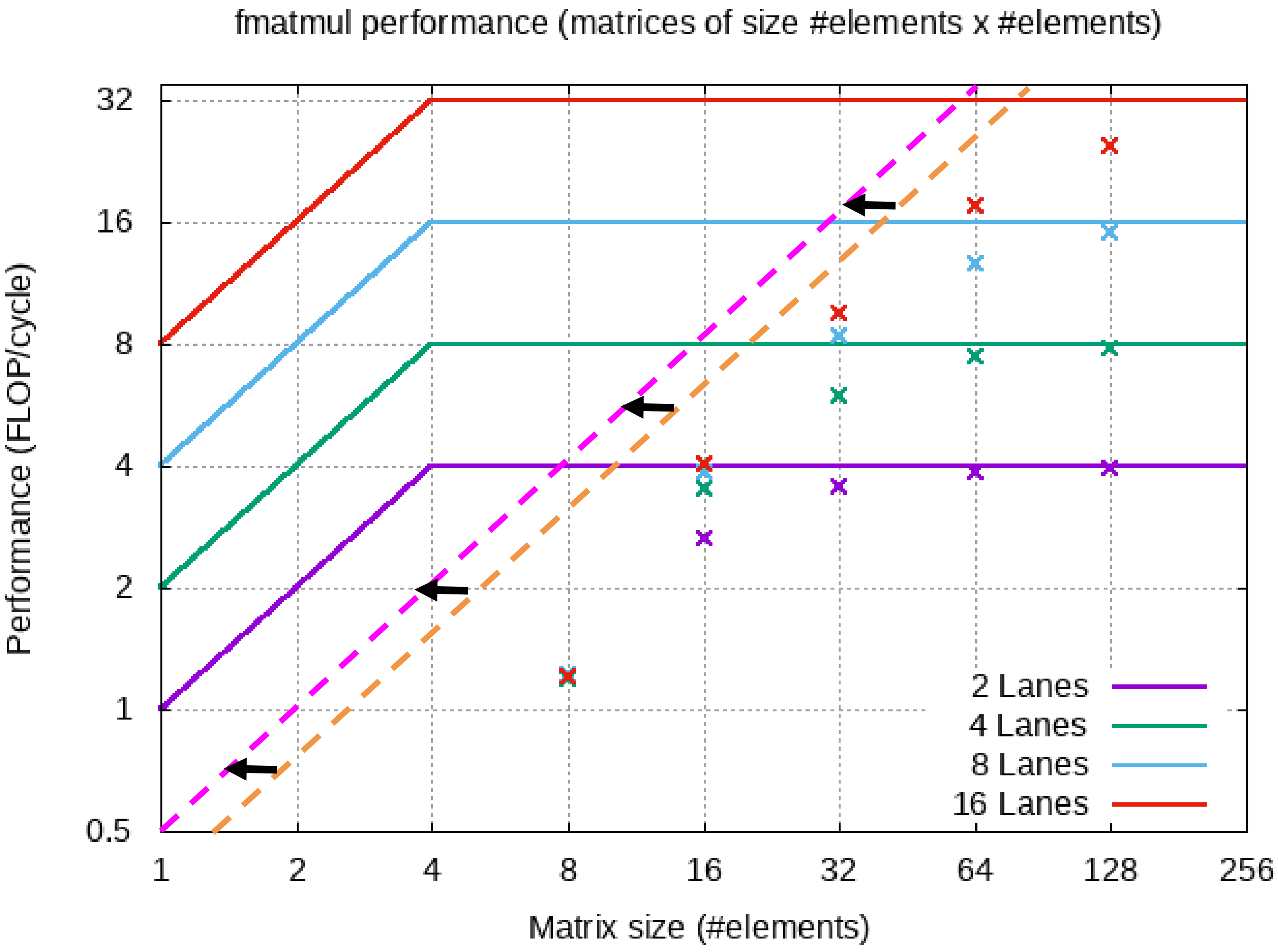
System performance – A focus on matmul

RVV 0.5:

- 2 instructions for vector-scalar operation
- Issue-rate: 1/5 cycles

RVV 1.0:

- 1 instruction for vector-scalar operation
- Issue-rate: 1/4 cycles



Software

Kernel	Domain	Format	Compute bound	Mask	Slide	Stride Mem	Idx Mem	Reduction	Max Perf [DP-FLOP/cycle]
matmul	Linear Algebra, ML	FP64, int64	Y	N	N	N	N	N	1 × 2.0 × L
conv2d	Signal Processing, ML	FP64, int64	Y	N	Y	N	N	N	1 × 2.0 × L
dotproduct	Linear Algebra	FP64, int64	Y	N	N	N	N	Y	1 × 0.5 × L*
jacobi2d	Stencil	FP64	Y	N	Y	N	N	N	1 × 1 × 1.0 × L
dropout	ML	FP32	N	Y	N	N	N	N	2 × 0.25 × L
fft	Signal Processing,	FP32	Y	Y	Y	N	Y	N	2 × 5/4 × L
dwt	Signal Processing,	FP32	N	N	N	Y	N	N	2 × 0.5 × L
pathfinder	Routing Algorithm	int32	N	Y	N	N	N	N	2 × 1.0 × L
exp	Scientific, ML	FP64	Y	N	N	N	N	N	1 × 28/21 × L
softmax	ML	FP32	Y	N	N	N	N	Y	2 × 32/25 × L
roi-align	ML	FP32	N	N	N	N	N	N	2 × 3/5 × L

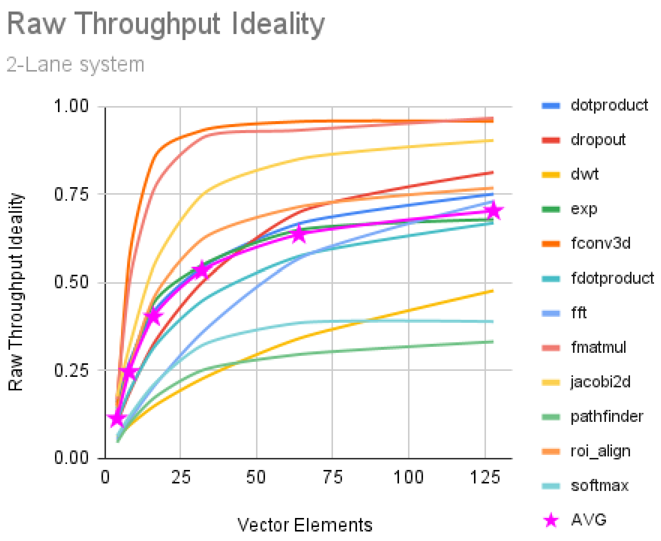
Software

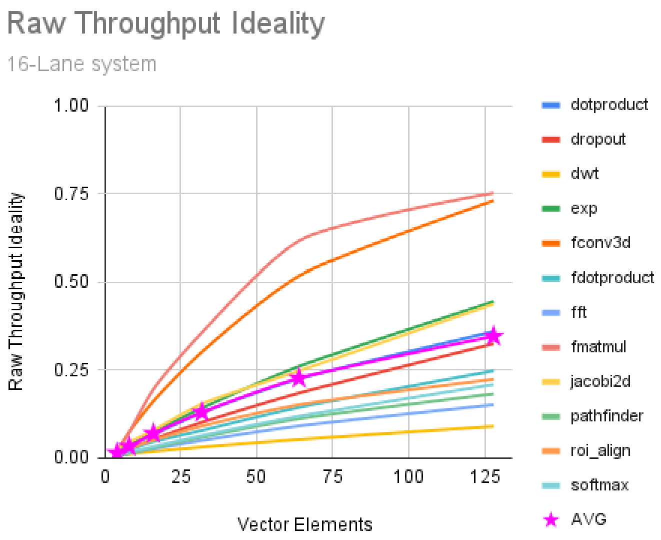
Kernel	Domain	Format	Compute bound	Mask	Slide	Stride Mem	Idx Mem	Reduction	Max Perf [DP-FLOP/cycle]
matmul	Linear Algebra, ML	FP64, int64	Y	N	N	N	N	N	1 × 2.0 × L
conv2d	Signal Processing, ML	FP64, int64	Y	N	Y	N	N	N	1 × 2.0 × L
dotproduct	Linear Algebra	FP64, int64	Y	N	N	N	N	Y	1 × 0.5 × L*
jacobi2d	Stencil	FP64	Y	N	Y	N	N	N	1 × 1 × 1.0 × L
dropout	ML	FP32	N	Y	N	N	N	N	2 × 0.25 × L
fft	Signal Processing	FP32	Y	Y	Y	N	Y	N	2 × 5/4 × L
dwt	Signal Processing	FP32	N	N	N	Y	N	N	2 × 0.5 × L
pathfinder	Routing Algorithm	int32	N	Y	N	N	N	N	2 × 1.0 × L
exp	Scientific, ML	FP64	Y	N	N	N	N	N	1 × 28/21 × L
softmax	ML	FP32	Y	N	N	N	N	Y	2 × 32/25 × L
roi-align	ML	FP32	N	N	N	N	N	N	2 × 3/5 × L

Throughput ideality

Benchmark performance analysis

- Throughput ideality = Throughput/Max_Throughput
- It measures **degree of ideality of** the **Ara** system **for that kernel** from 0% to 100%
- **Throughput ideality** strongly **depends on number of lanes** and **vector length**
- >50% of performance on average from 16 Elements/Lane on
- >75% of performance from 8 Elements/Lane for highly intensive kernels (fmatmul, fconv2d, ~jacobi2d)



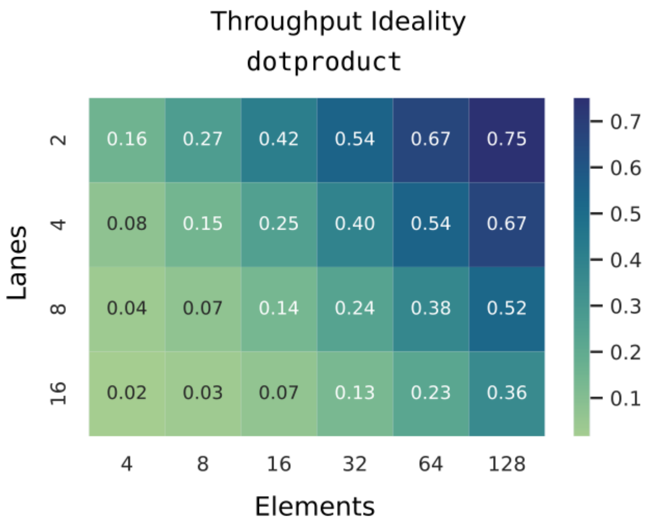


ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA Future Computing Laboratory

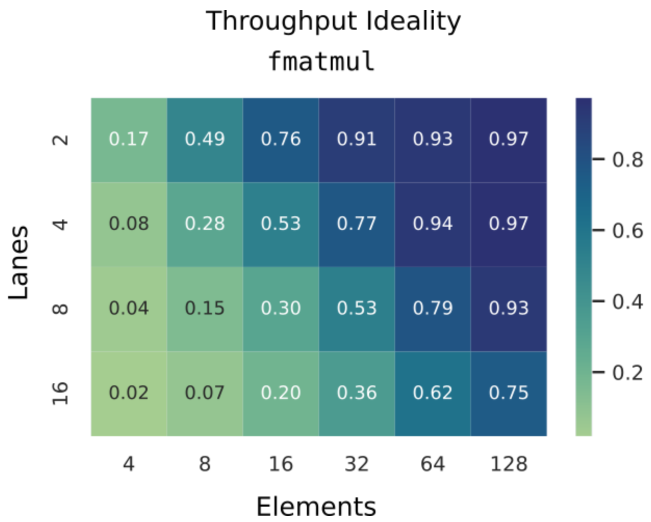
Throughput ideality - Byte/Lane ratio

Throughput ideality
i.e., actual/ideal performance
Measure of how ideally Ara behaves

- The longer the vector, the better the throughput ideality (until peak)
- How long is a "sufficiently long" vector?
- Performance ideality correlated with Byte/Lane ratio (~Element/Lane)
- Same computational efficiency with 2*L? Our vector should be as twice as long
- i.e., to double performance, we need a longer vector too


Throughput Ideality
dotproduct

Lanes \ Elements	4	8	16	32	64	128
2	0.16	0.27	0.42	0.54	0.67	0.75
4	0.08	0.15	0.25	0.40	0.54	0.67
8	0.04	0.07	0.14	0.24	0.38	0.52
16	0.02	0.03	0.07	0.13	0.23	0.36


Throughput Ideality
fmatmul

Lanes \ Elements	4	8	16	32	64	128
2	0.17	0.49	0.76	0.91	0.93	0.97
4	0.08	0.28	0.53	0.77	0.94	0.97
8	0.04	0.15	0.30	0.53	0.79	0.93
16	0.02	0.07	0.20	0.36	0.62	0.75

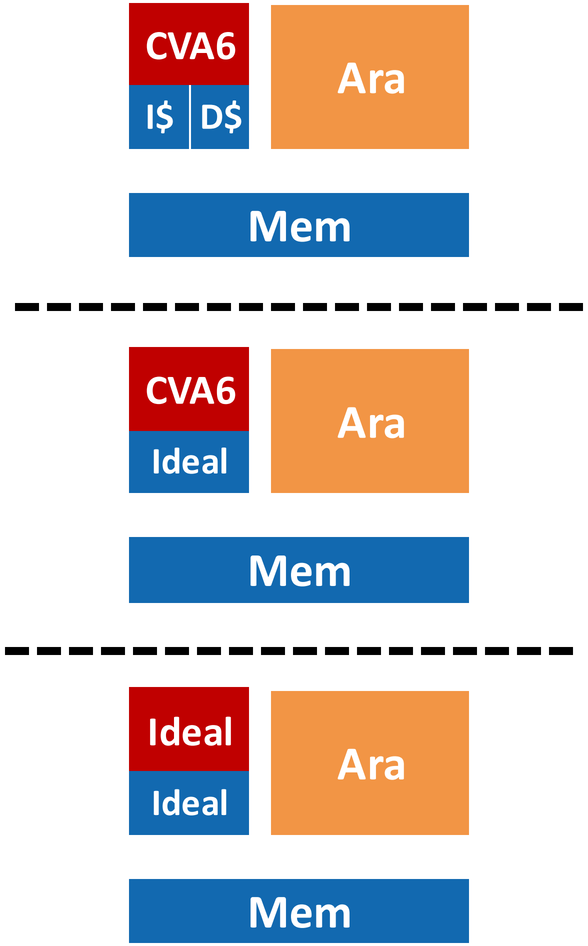
Performance analysis

Ara2 suffers from short/medium vector performance loss

- Who to blame?
  CVA6?
  Memory System?
  Vector accelerator?

Analyze three systems

- CVA6 + Scalar I/D$ + Ara
  - Show limitations due to the whole system
- CVA6 + Ideal I/D$ + Ara
  - Show limitations due to CVA6 and Ara
- Ideal CVA6 + Ideal I/D$ + Ara
  - Show limitations due to Ara



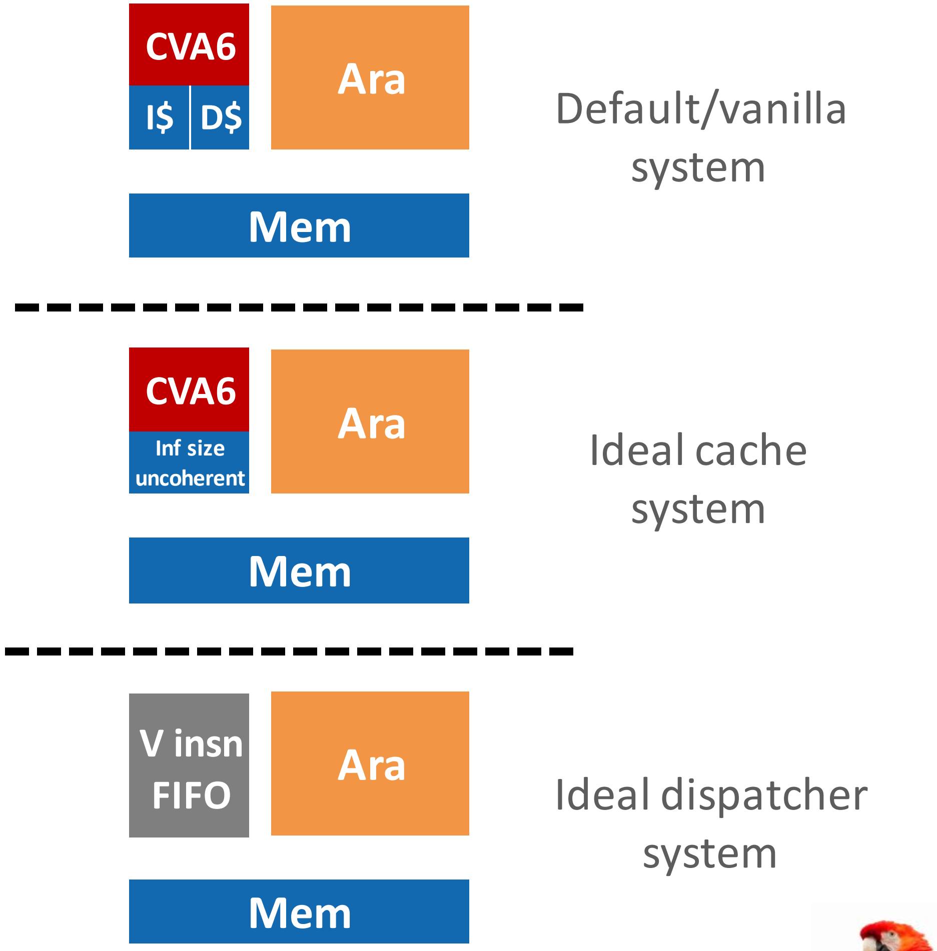
Performance analysis

Ara2 suffers from short/medium vector performance loss

- Who to blame?
  CVA6?
  Memory System?
  Vector accelerator?

Analyze three systems

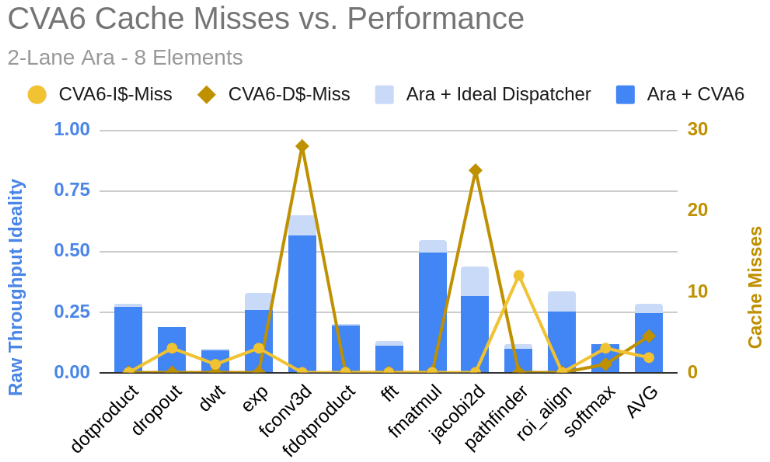
- CVA6 + Scalar I/D$ + Ara
  - Show limitations due to the whole system
- CVA6 + Ideal I/D$ + Ara
  - Show limitations due to CVA6 and Ara
- Ideal CVA6 + Ideal I/D$ + Ara
  - Show limitations due to Ara

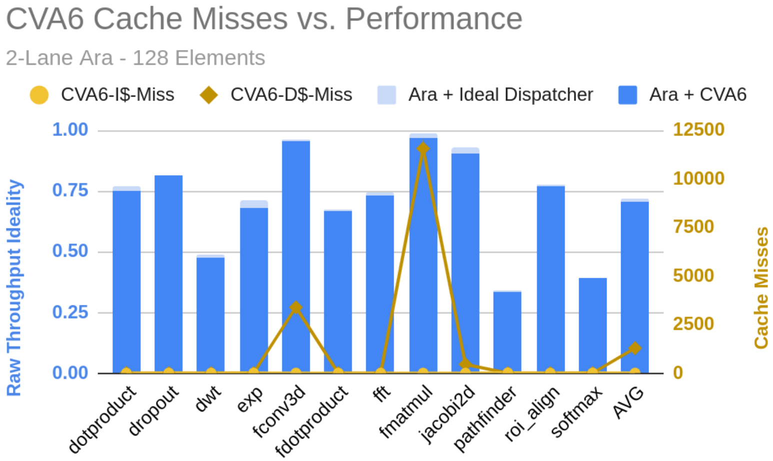


CVA6 influence on performance

Ideal dispatcher does **not improve performance** much!
(2-lane Ara, 8-element vector)

Almost **no performance** gain **for long vectors**
(2-lane Ara, 128-element vector)

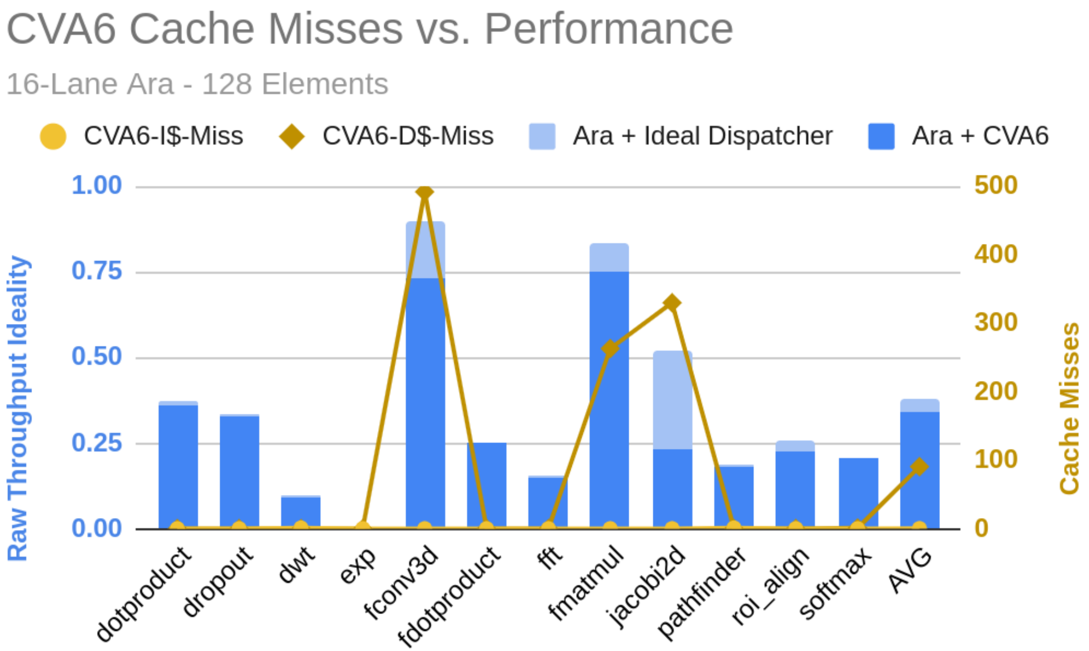




CVA6 influence on performance

16 lanes, 128 elements case

Correlation with D$ misses even more evident



CVA6 influence on performance

The performance gain should be attributed to the ideal caches, and not to the ideal CVA6!

All three kernels rely on scalar forwarding from scalar to vector core!

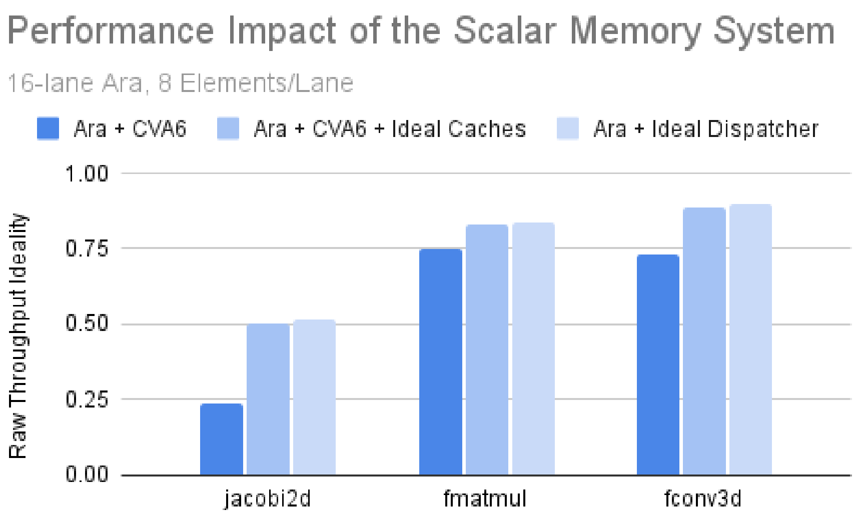
D$ misses heavily affect performance!

fmatmul, fconv2d

- Source and destination separated in memory
- Problem: cache line/bus size

jacobi2d:

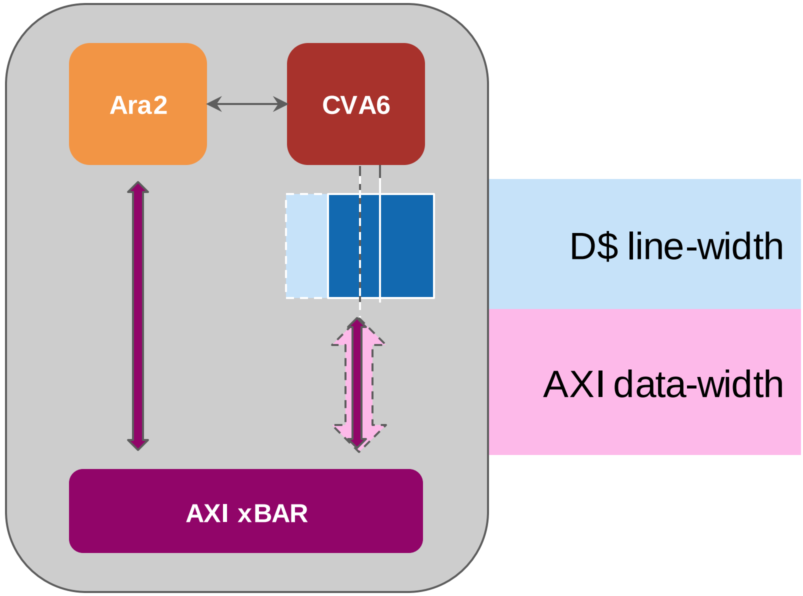
- Iterative algorithm
- Elements written by vector store get read back from the scalar core (from the D$!)
- Problem: scalar cache



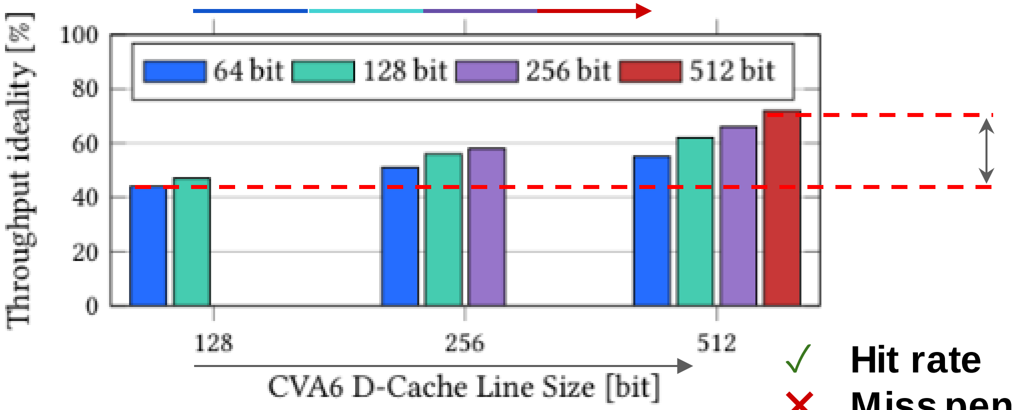
Scalar D$ influence on performance

- `16 Lanes`, `fmatmul`, `n = 16`

- **Throughput ideality** vs.
  - AXI data-width
  - D$ line-width

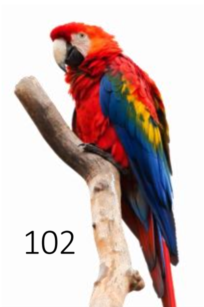


AXI data-width ✓ **Miss penalty**



From 42% to 72%!

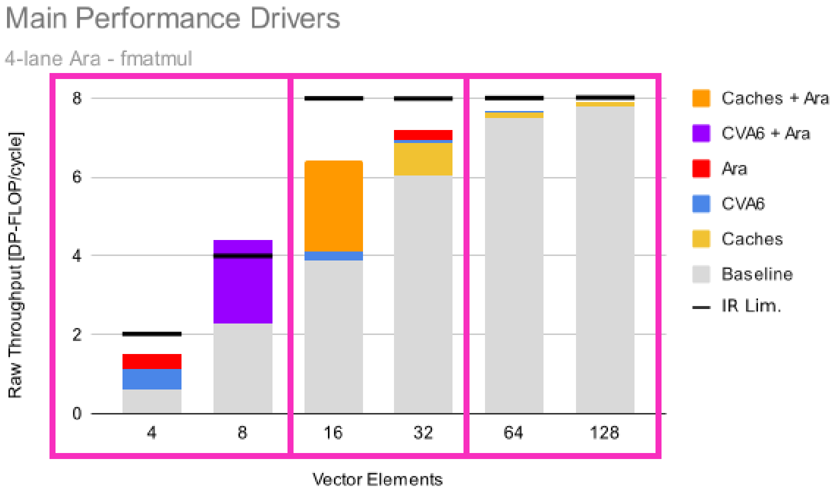
✓ **Hit rate**
✗ **Miss penalty**



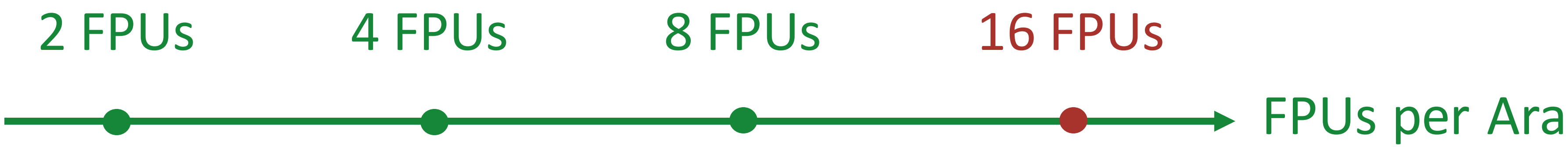
Main performance drivers

Absolute **fmatmul** performance of a **4-lane Ara**

- Different performance drivers for different sizes
- Orange and violet are merged gains
- Huge CVA6 impact with short vectors only
- Caches play a role for medium vectors
- Long vectors are already at their peak
- Issue-Rate limitation does not hold for ideal CVA6, but holds for the real system!



Ara wants **more Computational Power**



2 FPUs

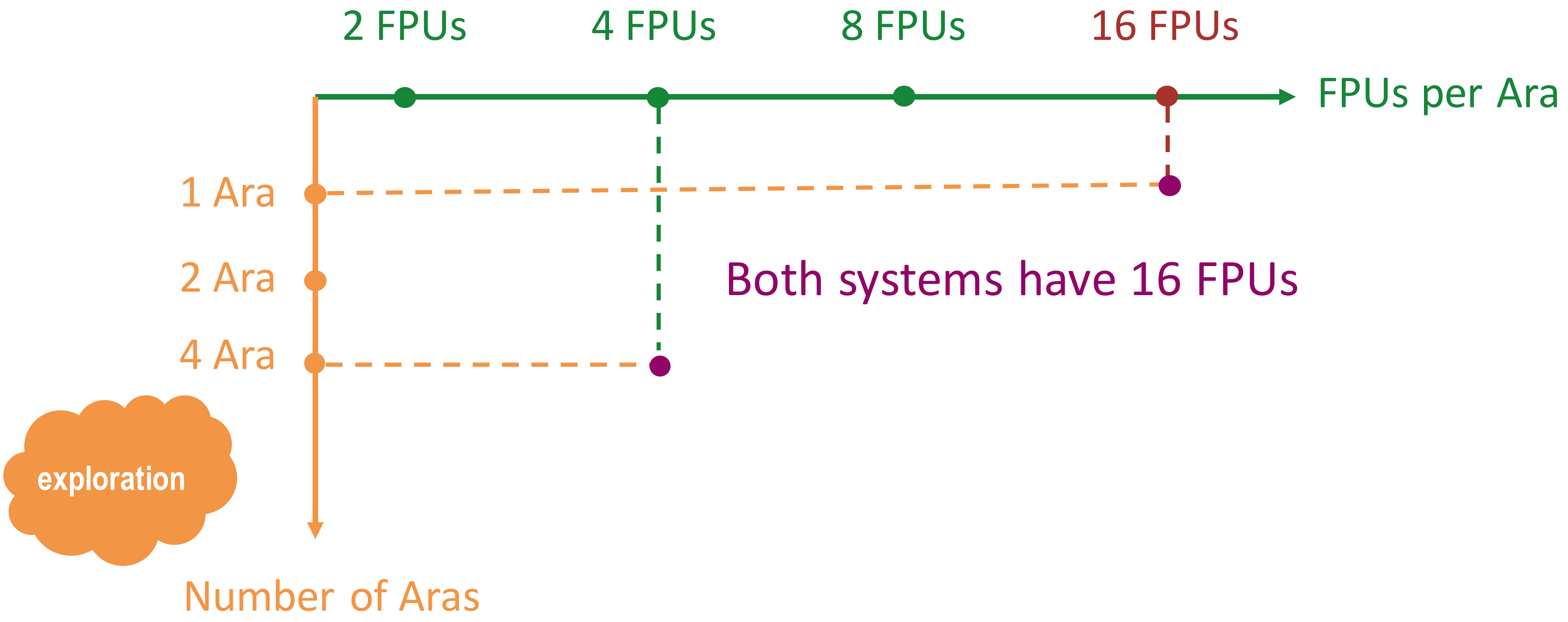
4 FPUs

8 FPUs

16 FPUs

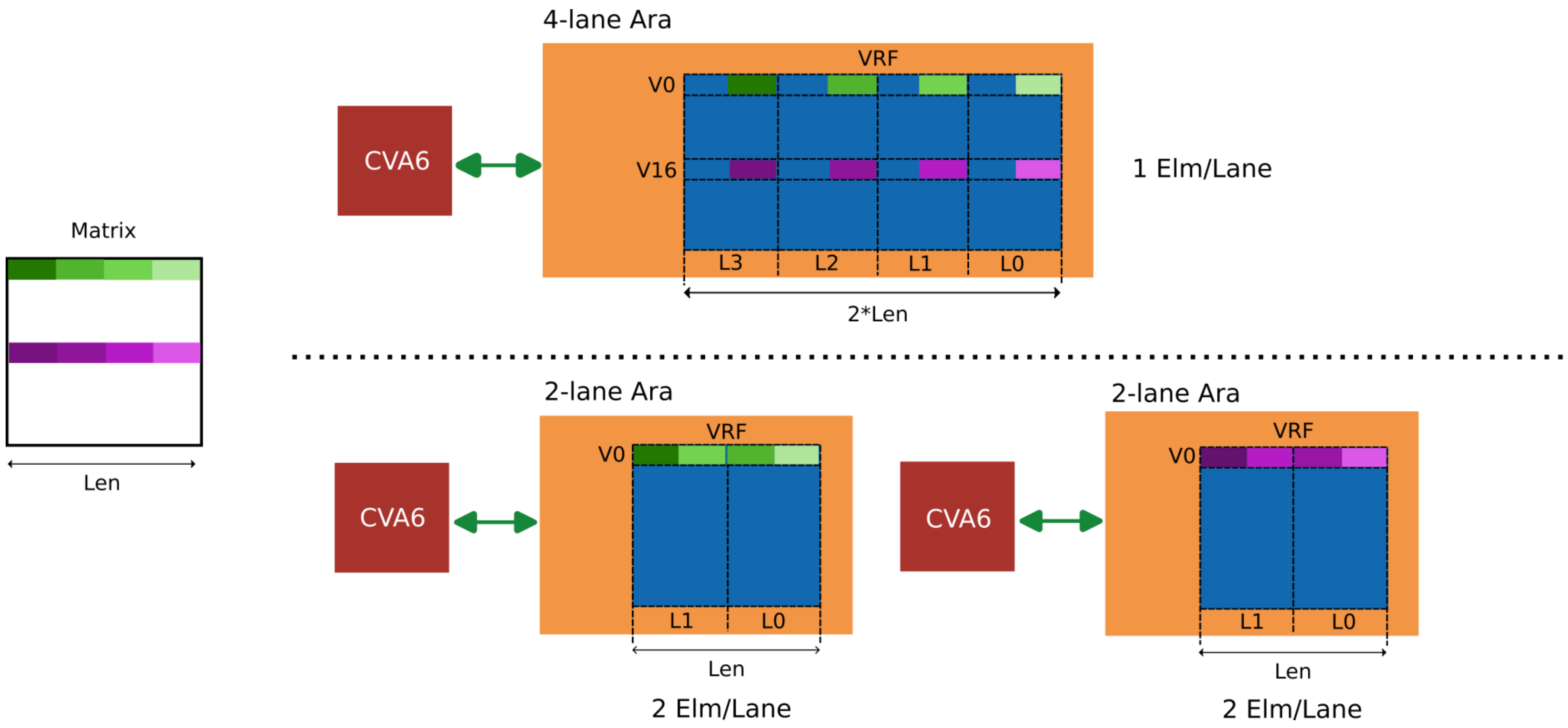
FPUs per Ara

Ara wants more Computational Power



Single core vs. Multi core

- Increase Byte/Lane ratio with Multi-Core?
- Exploit second dimension!



Single core vs. Multi core system

16 FPU system in different flavors

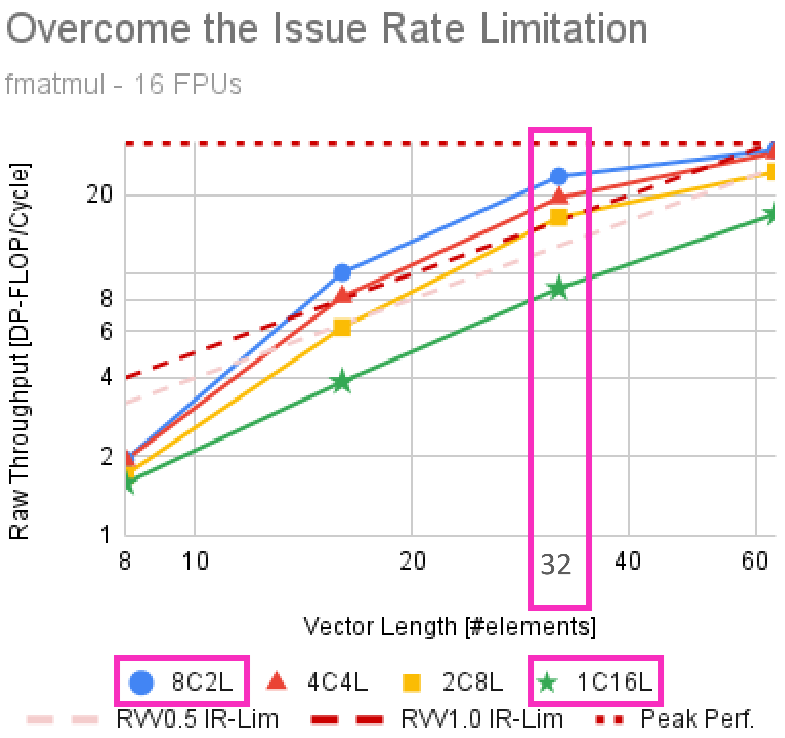
- 1C16L: one 16-lane Ara
- 8C2L: eight 2-lane Ara

e.g., 32x32x32 matmul performance

32 elements processed by a single 16-lane Ara under-utilize the system's resources (2 Elm/Lane)

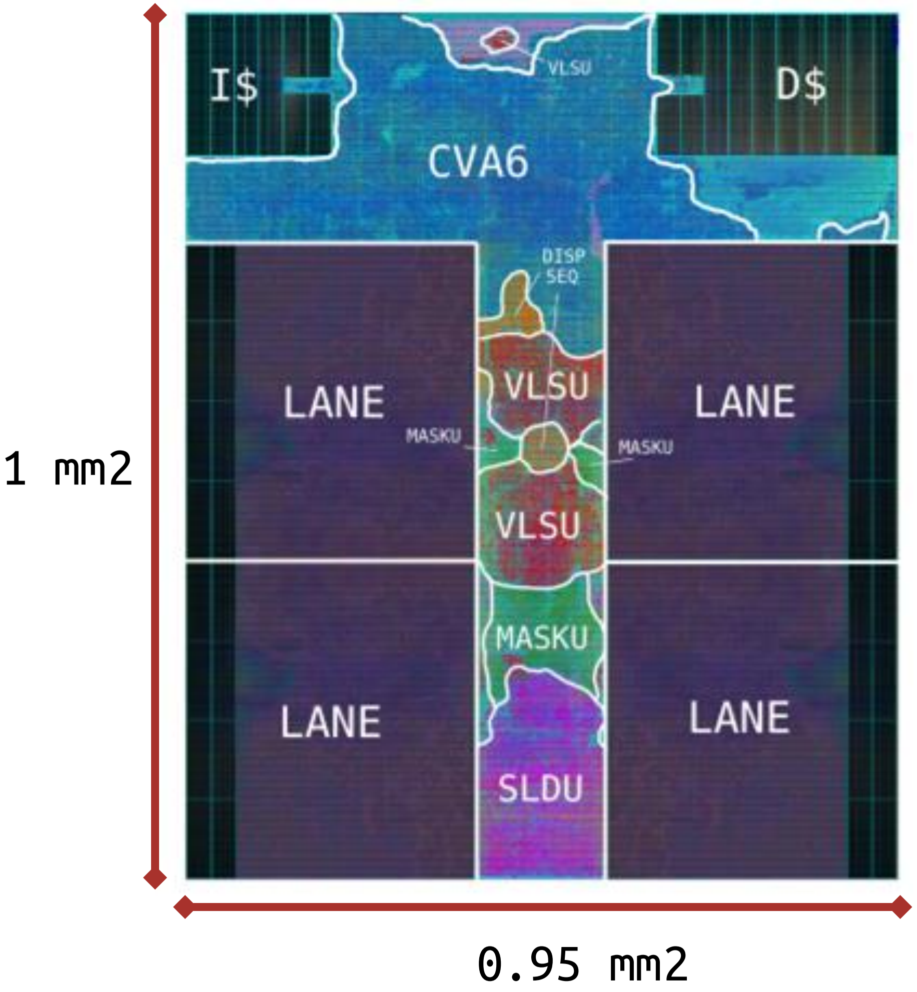
Eight 2-lane Aras exploit the second dimension of the matrix to speed up the computation (16 Elm/Lane)

The issue-rate limitation does not exist for 8C2L

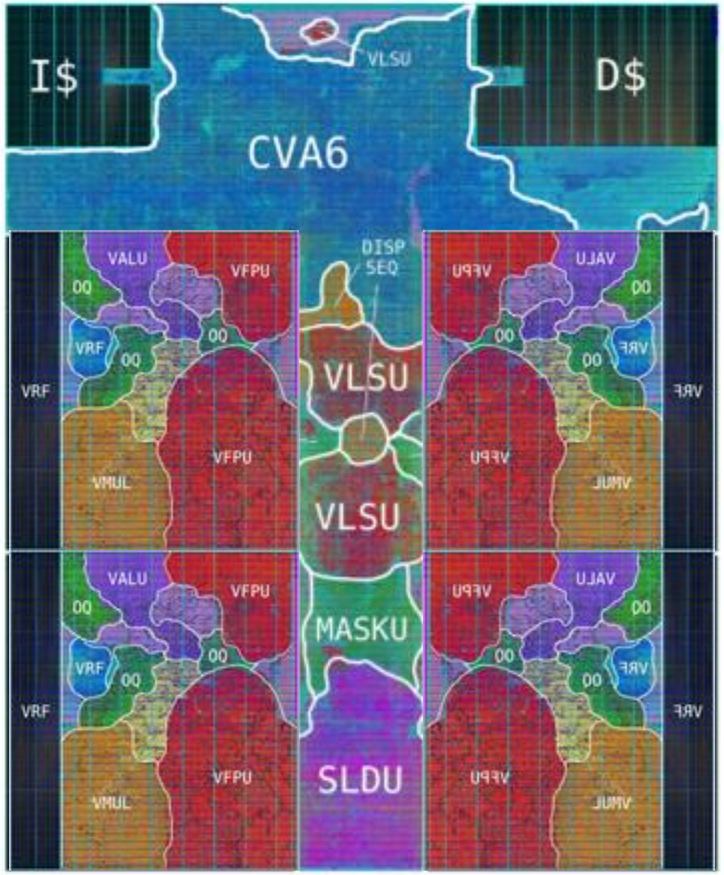


Physical Implementation

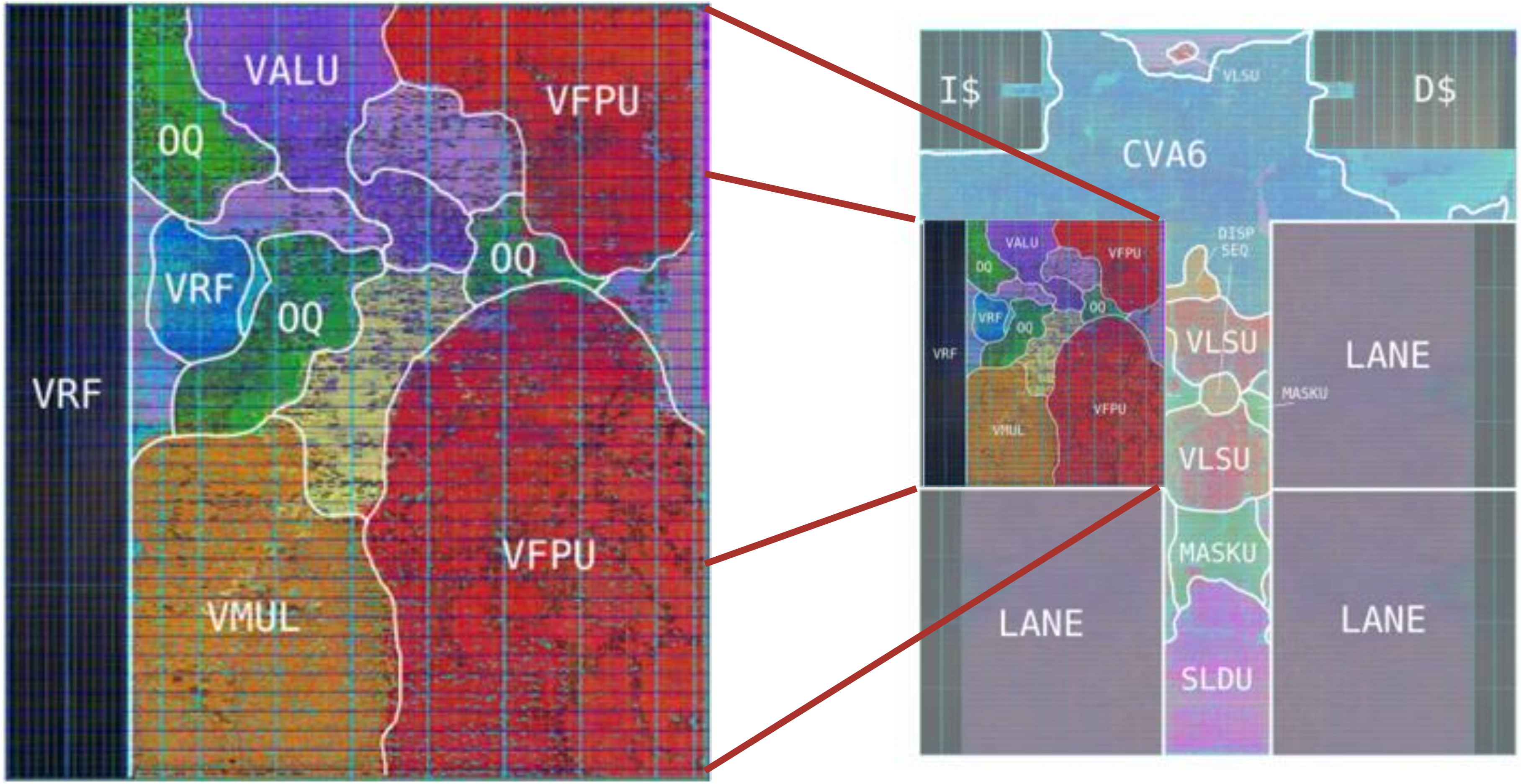
- GF22 FDX FD-SOI
- SS frequency: 950 MHz
- TT frequency: 1.35 GHz
- Efficiency: 37.8 DP-GFLOPS/W
  - 1.35GHz, fp-matmul



Physical Implementation

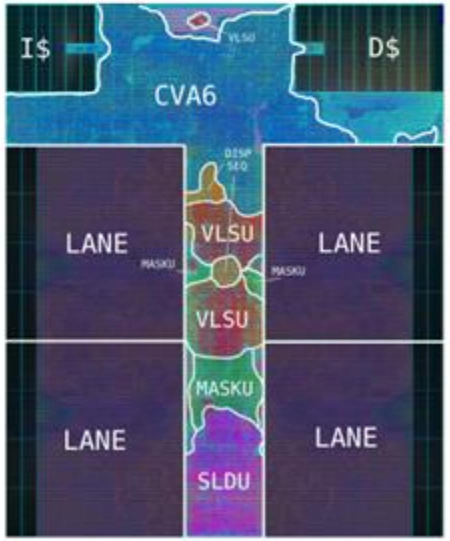


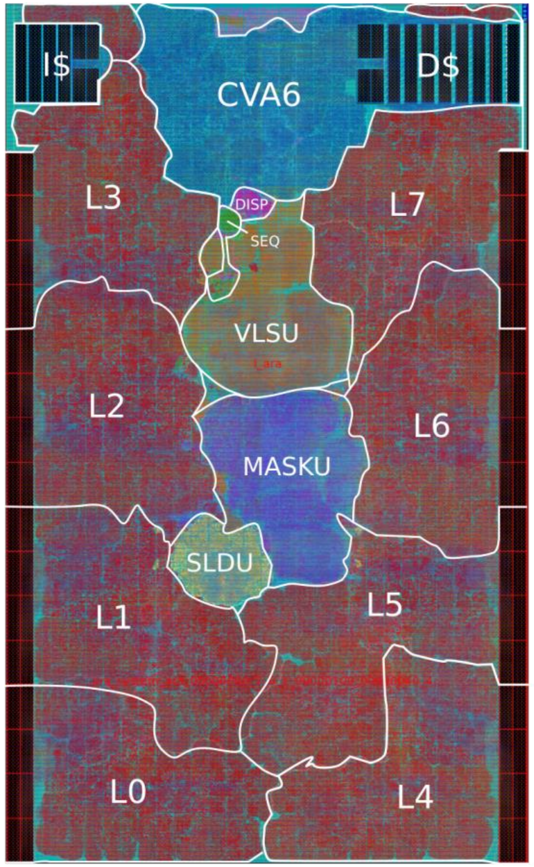
Physical Implementation



Physical Implementation – Scaling up

- Scale up with the number of lanes
- Flat flow for #L > 8





PPA Metrics

ARAV2 (ARAV1)	2 LANES	4 LANES	8 LANES	16 LANES*
SS Frequency [MHz]	950 (920)	960 (930)	940 (870)	860 (780)
TT Frequency [GHz]	1.35 (1.25)	1.35 (1.25)	1.35 (1.17)	1.26 (1.04)
Die Area (mm2)	0.588	0.953	1.876	4.47
Cell+Macro Area (mm2)	0.455 (0.443)	0.734 (0.683)	1.346 (1.174)	2.559 (2.136)
Macro Area (mm2)	0.111	0.153	0.235	0.4
Peak Efficiency	34.1 (35.6)	37.7 (37.8)	35.7 (39.9)	30.3 (40.8)

- *Simplified Mask Unit
- Improved SS, TT frequency
- Smaller die, slightly higher Cell+Macro area (with added functionality)
- Same efficiency up to 4 lanes, then the trend is different
- RVV 0.5 gains in efficiency until 16 lanes
- Efficiency peak at 4 lanes for RVV 1.0 design

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA Future Computing Laboratory

Area breakdown

Area [kGE]	2L	4L	8L	16L	16L*
CVA6	894	896 (1.0×)	906 (1.0×)	904 (1.0×)	904 (1.0×)
Lane	612	617 (1.0×)	626 (1.0×)	628 (1.0×)	573 (0.9×)
Dispatcher	16	17 (1.1×)	19 (1.1×)	23 (1.2×)	20 (1.1×)
Sequencer	14	15 (1.1×)	17 (1.2×)	29 (1.6×)	29 (1.6×)
MASKU	38	97 (2.5×)	300 (3.1×)	1105 (3.7×)	442 (1.5×)
ADDRGEN	35	36 (1.0×)	44 (1.2×)	59 (1.3×)	60 (1.4×)
VLDU	15	45 (3.0×)	212 (4.7×)	1286 (6.1×)	1135 (5.4×)
VSTU	8	21 (2.8×)	64 (3.1×)	332 (5.2×)	342 (5.3×)
New SLDU	24	48 (2.0×)	94 (2.0×)	196 (2.1×)	190 (2.1×)
Old SLDU	39	131 (3.4×)	577 (4.4×)	2900 (5.0×)	2860 (5.0×)

uArch optimization: Slide Unit

SLDU made scaling up to 16 lanes impossible

- Exponential growth after P&R

Area [kGE]	2L	4L	8L	16L
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
Old SLDU	39	131 (3.4×)	577 (4.4×)	2900 (5.0×)

All-to-all byte connections scale ~L^2

Do we need to support whichever slide amount?

uArch optimization: Slide Unit

SLDU made scaling up to 16 lanes impossible

- Exponential growth

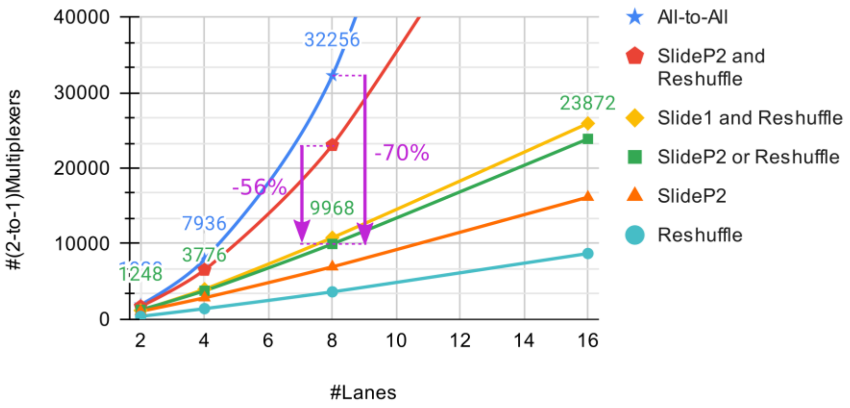
Area [kGE]	2L	4L	8L	16L
New SLDU	24	48 (2.0×)	94 (2.0×)	196 (2.1×)
Old SLDU	39	131 (3.4×)	577 (4.4×)	2900 (5.0×)

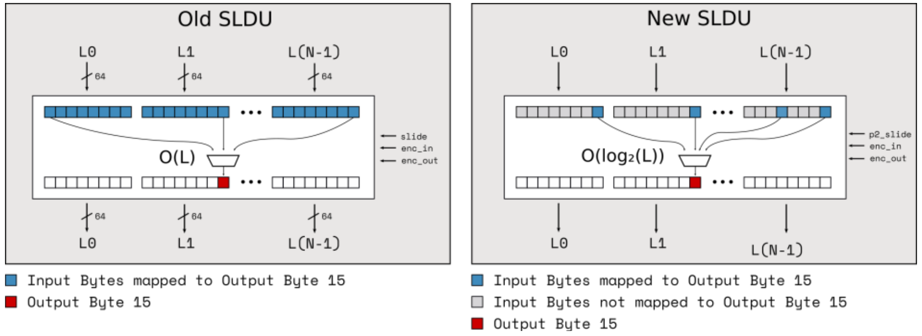
All-to-all byte connections scale ~L^2

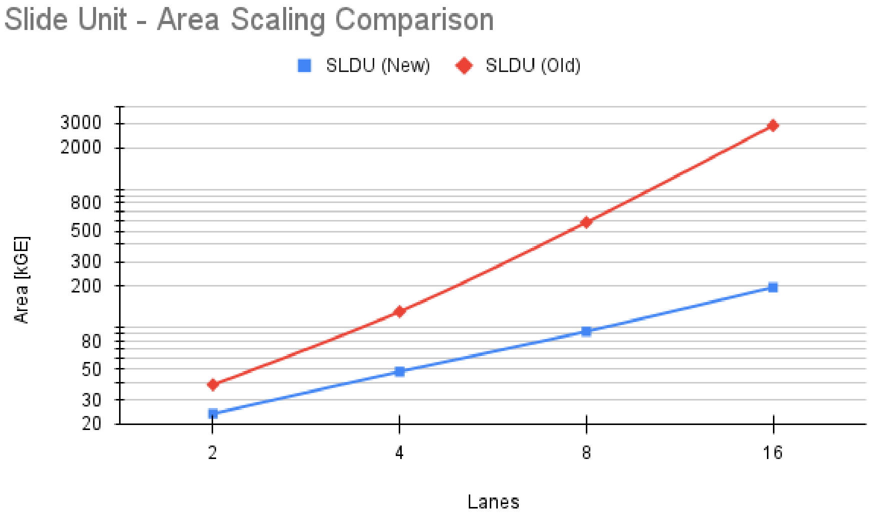
Do we need to support whichever slide amount?

Solution

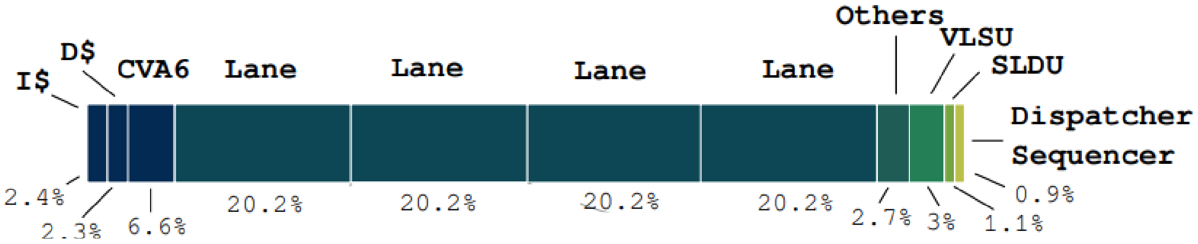
- Support only power-of-2 slide amounts
- Either reshuffle OR slide




Old SLDU
L0
L1
L(N-1)
64
O(L)
slide
enc_in
enc_out
■ Input Bytes mapped to Output Byte 15
■ Output Byte 15
New SLDU
O(log₂(L))
p2_slide
enc_in
enc_out
■ Input Bytes mapped to Output Byte 15
□ Input Bytes not mapped to Output Byte 15
■ Output Byte 15

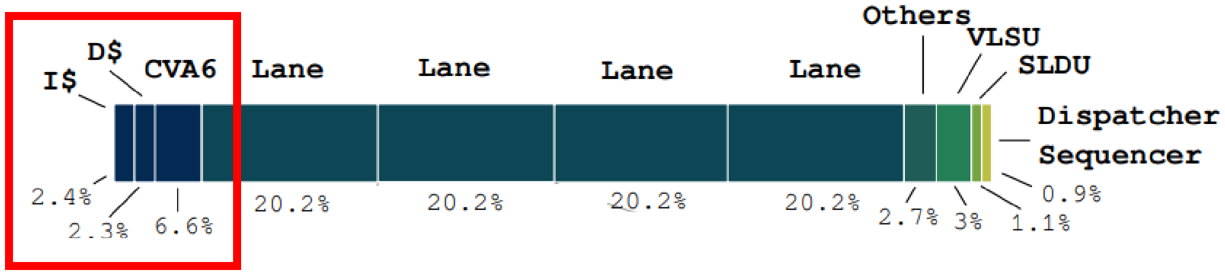


Power breakdown



- 4 Lanes
- FP-matmul computational loop

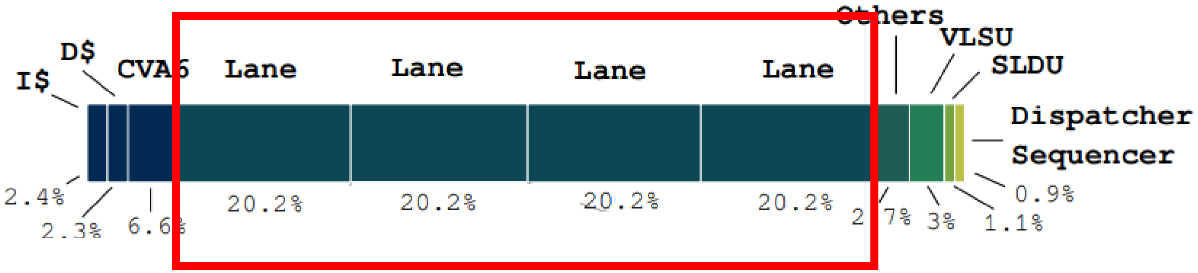
Power breakdown



11% power to CVA6

- 4 Lanes
- FP-matmul computational loop

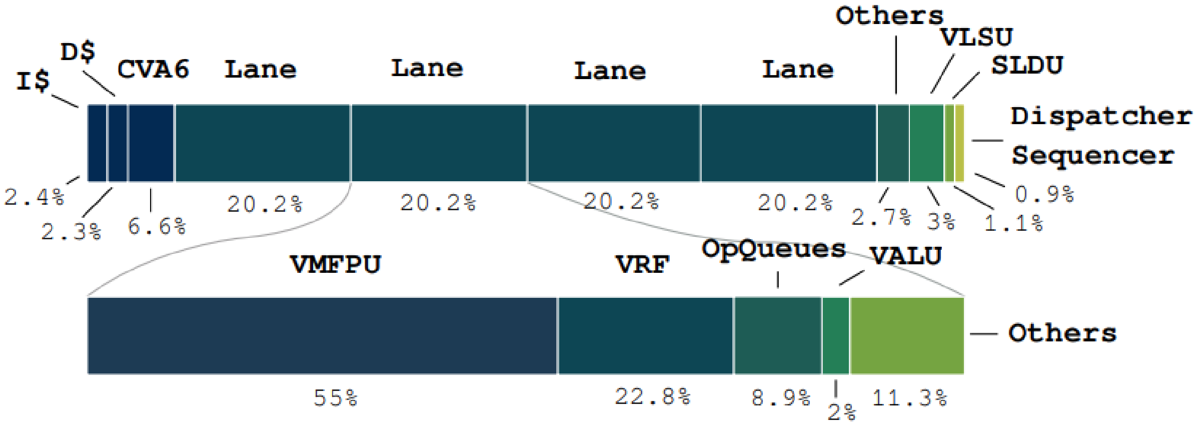
Power breakdown



>80% power to the Lanes

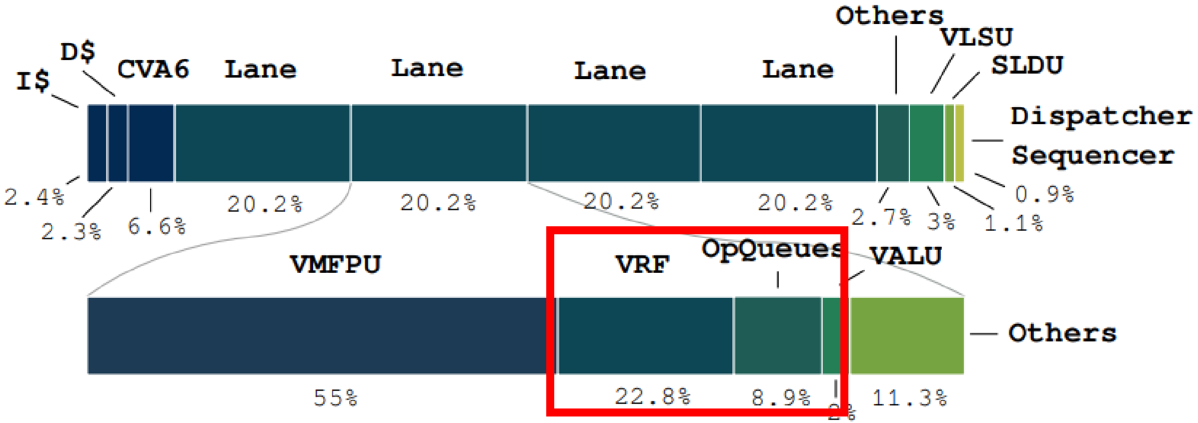
- 4 Lanes
- FP-matmul computational loop

Power breakdown



- 4 Lanes
- FP-matmul computational loop

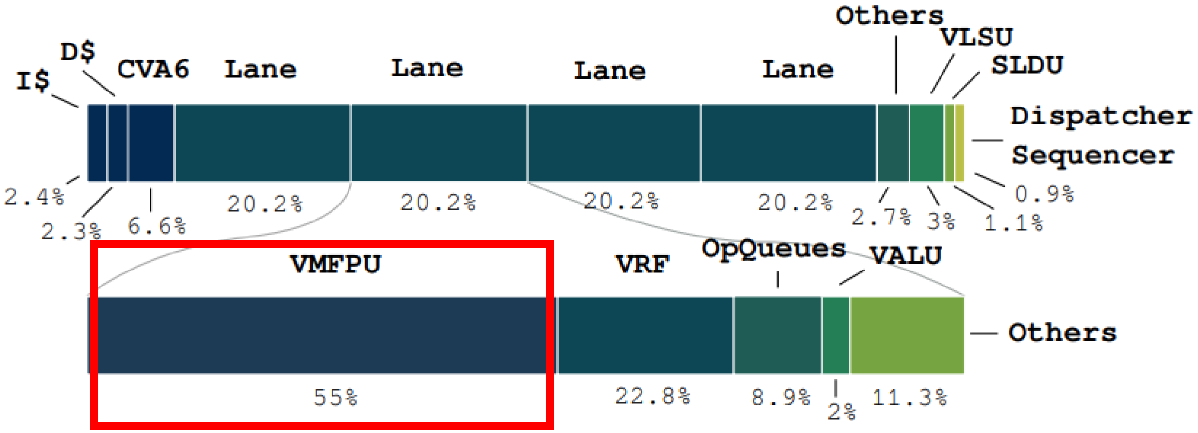
Power breakdown



~30% Buffering

- 4 Lanes
- FP-matmul computational loop

Power breakdown



55% Actual Computation

- 4 Lanes
- FP-matmul computational loop

Energy Efficiency

Boost energy efficiency

- Clock-gate unused CVA6 memory banks, VRF banks
- Operand-gate the SIMD multipliers
- Clock-gate the SIMD multipliers when not used

```
Ara 4 Lanes, fmatmul
A[128][128] x [128][1KiB] = C[128][1KiB]
```

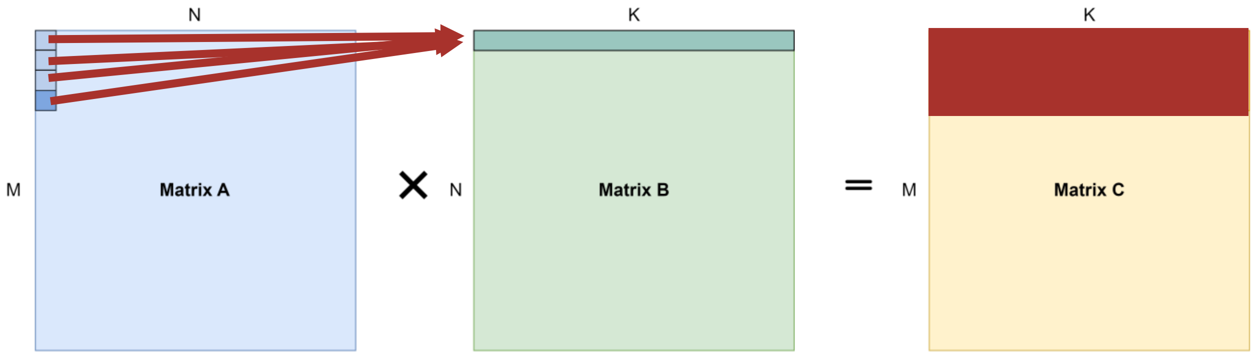
Fmt	Eff	
fp64	37.74	DP-FLOPS/W
fp32	90.04	SP-FLOPS/W
fp16	195.86	HP-FLOPS/W
int64	38.32	DP-GOPS/W
int32	85.26	SP-GOPS/W
int16	180.56	HP-GOPS/W
int8	376.04	BP-GOPS/W

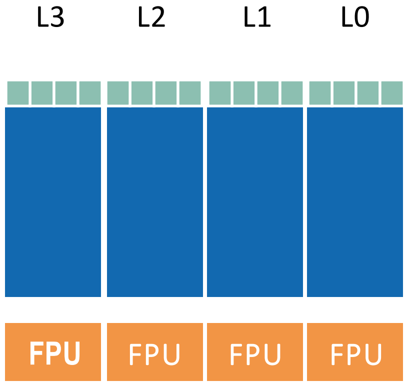
ISA extension – Transformers and small Matmul

Improve small-size matmul?

Default algorithm

- Example:
  `32x32x32 fp32-matmul`
  `(4*32)` output tile
  4-Lane Ara
- 4 scalar loads
  (D$ misses)
- 4 `vmacc` instructions
  (4 setup time, issue-rate limitation)
- 4 Elements/Lane -> 16 B/Lane



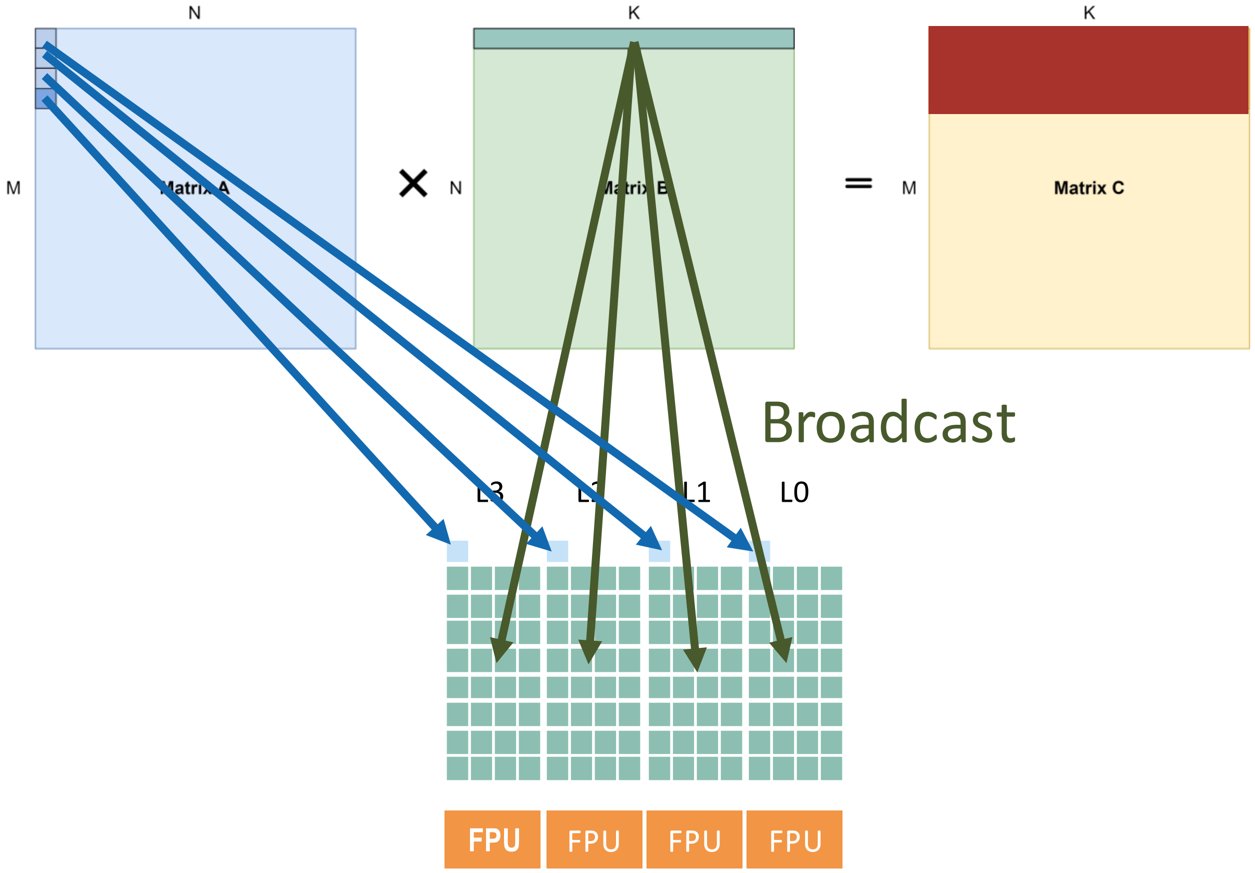


ISA extension – Transformers and small Matmul

Improve small-size matmul?

Matrix algorithm

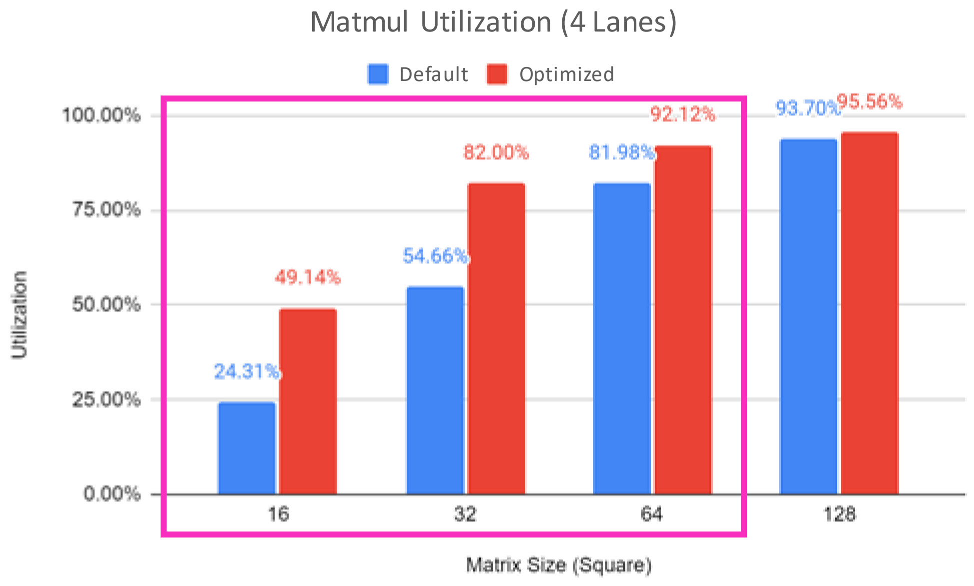
- Example:
  `32x32x32 fp32-matmul`
  `(4*32)` output tile
  8-Lane Ara
- 0 scalar loads
  (no D$ misses)
- 1 `mmacc` instruction
  (1 setup time)
- 32 Elements/Lane -> 64 B/Lane



ISA extension – Transformers and small Matmul

Improve small-size matmul?

Default algorithm vs. Optimized Matrix algorithm

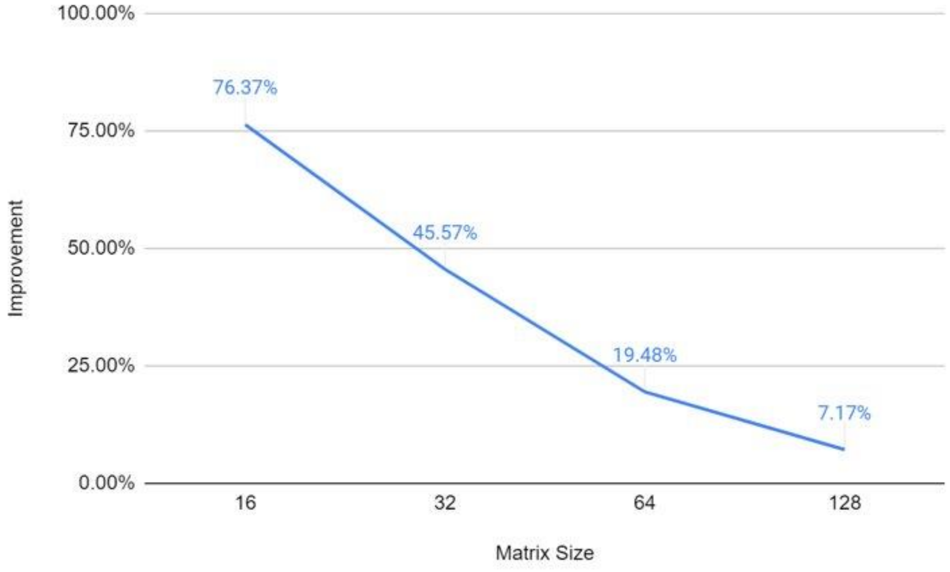


Peak of 2x performance for small problems!!

ISA extension – Transformers and small Matmul

Trial on 2, 4, 8 lanes – No frequency loss, negligible area impact

What about **Energy Efficiency**?
4-lane Ara example



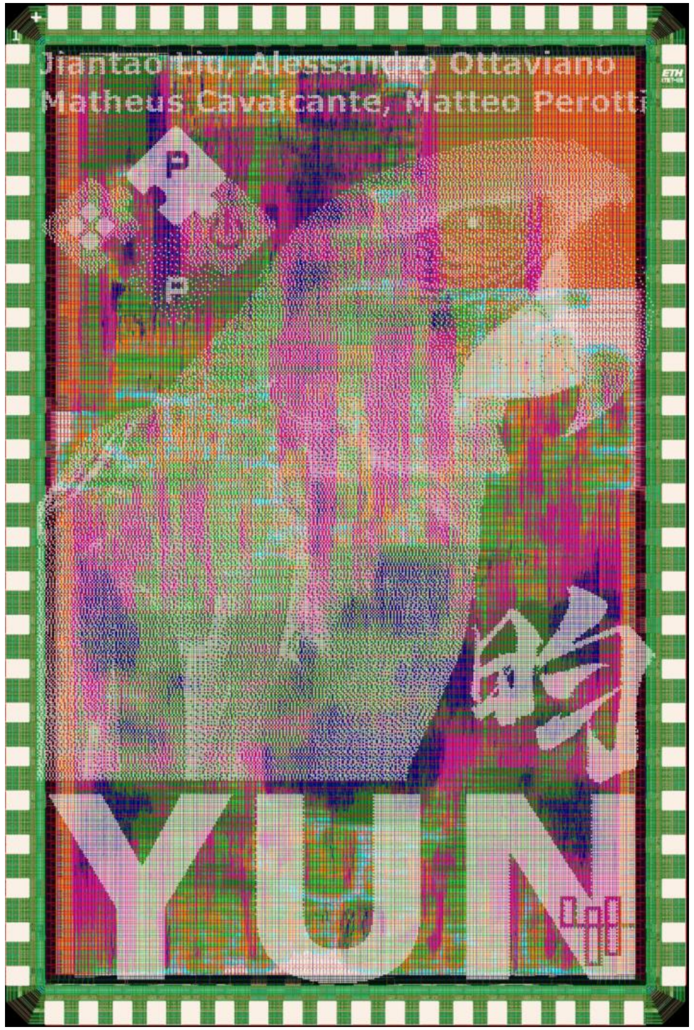
Peak of +75% EE improvement on small problems!

SoA Comparison

- Vitruvius+
  - 8-lane processor
  - ~ Max frequency for 8 Lanes
  - ~ IPC on jacobi2d, matmul, pathfinder
  - Unfair power comparison (no scalar core? No caches? How data are generated?)
- Vicuna
  - int-only RVV1.0
  - ~ utilization on matmul
  - FPGA only PPA metrics
- Hwacha
  - ~ utilization on matmul
  - Chip-level Power, hard to compare

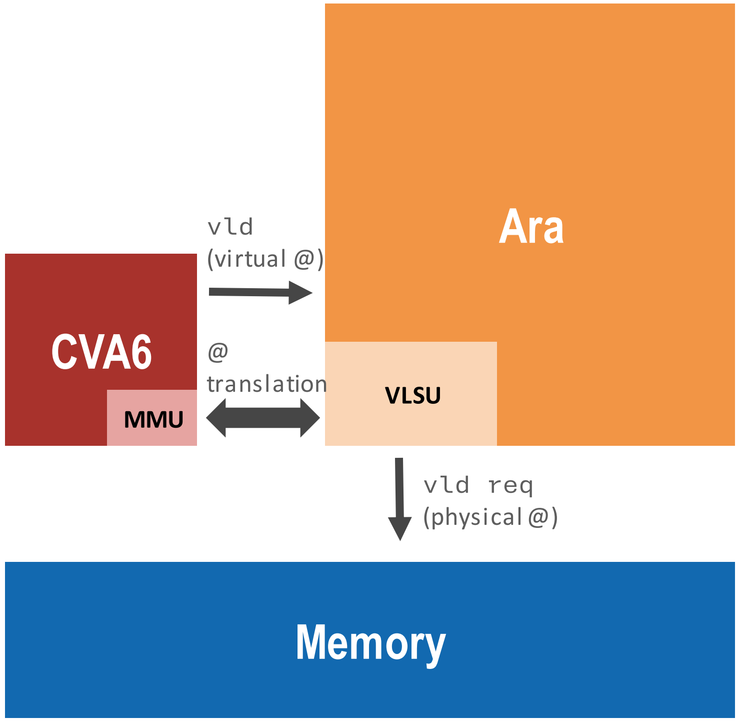
Yun: Ara’s in tsmc65

- Yun - tsmc65
- Ara – 4 Lanes
- SoC Architecture
  - Ara System
  - 64 KiB SRAM Memory
  - JTAG + Debug Module
  - Double-precision FPU
  - Die area: 6 mm2
  - Frequency: ~250 MHz
- **Already tested: it works!**


Jiantao Liu, Alessandro Ottaviano
Matheus Cavalcante, Matteo Perotti
YUN
ETH

What's further?

- OS support
  - Virtual memory
    - Re-use CVA6's MMU
    - Separate TLB for V accesses?
  - Context switch
    - Restrict V core to one process only
    - Faster context switch
  - Exceptions
    - Extensive debug

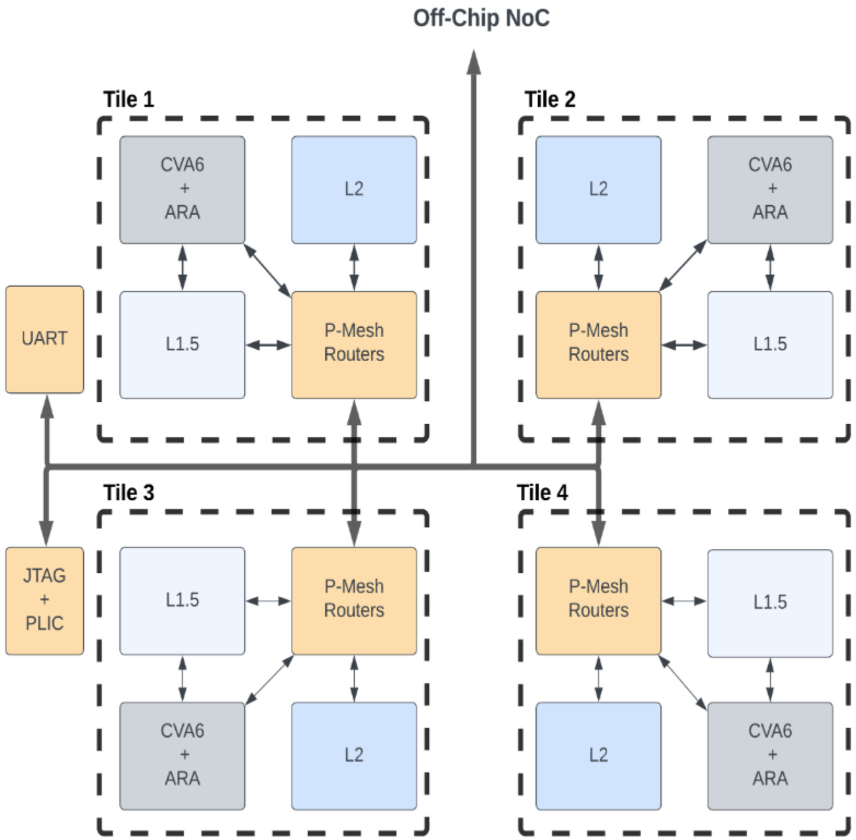


What's further?

- Matrix Extension
  - Load matrices instead of vectors
  - Decouple Scalar-Vector processors
    - Bypass D$!
  - Increase #operations/#instruction ratio
    - Pay less start-up time cycles
  - Reduce VRF accesses

What's further?

- Ara OpenPITON – PolAra
- Ara in real multi-core environment
- Optimizing matmul on L2-sparse memory hierarchy



https://github.com/openhwgroup/core-v-polara-apu/tree/openpiton_polara

Project goals

- ✓ Expand benchmark pool with heterogeneous applications for high-dimensional workloads
  - ➢ Extended benchmark pool with heterogeneous set of kernels and program to stress different units of Ara2
  - ➢ Performance analysis for each kernel and degree of ideality w.r.t. the maximum achievable
- ✓ Update Ara from RISC-V V v0.5 to the most recent RISC-V Vector ISA (v0.9)
  - ➢ Ara updated to RISC-V V v1.0 frozen
  - ➢ Added RVV features missing from AraV1 (e.g., int- and fp-reductions)
- ✓ Develop uArchitecture to improve IPC, Performance, Energy Efficiency
  - ➢ uArchitectural optimization (e.g., SLDU optimization) to scale up to 8 and 16 lanes
  - ➢ Exploration of new ISA extension to speed up transformer models
  - ➢ Transprecision FPU down to 16 bit, ready to host 16alt and 8-bit formats
  - ➢ Higher performance (frequency), same Energy Efficiency for smaller Ara instances
- ✓ Address coherence problem between Ara and Ariane
  - ➢ Added coherence and memory ordering engine
- ✓ HW/SW interface mixing intrinsics, hand-optimized macros, vector compilation
  - ➢ Compiler support with RVV intrinsics
  - ➢ Benchmark pool and optimized programs with intrinsics, hand-optimized macros, and possibly auto-vectorization

Publications

Published:

ASAP 2022 Conference

Title: A “New Ara” for Vector Computing:
An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design

Authors: Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Luca Benini

Link: https://ieeexplore.ieee.org/abstract/document/9912071

Work in progress:

IEEE Journal TCOMP

Thank you!

Q&A

EFCL Project Timeline, WPs, Milestones

Year 1

WP1: Workload analysis and benchmarking
Select target sensors and key algorithms - develop a **benchmark suite** with **performance targets** for **key application kernels** used in multi-dimensional sensor processing and fusion for consumer devices.

WP2: uArch design and exploration of instruction extensions
Moving from ARAv1 **develop the new micro-architecture** supporting the standard ISA and **evaluate the impact in HW cost** and **performance benefits** of instruction extensions targeted to the workloads in WP1.

Milestones: workload and benchmark analysis, version 0.1 of Ara2 with performance profiling

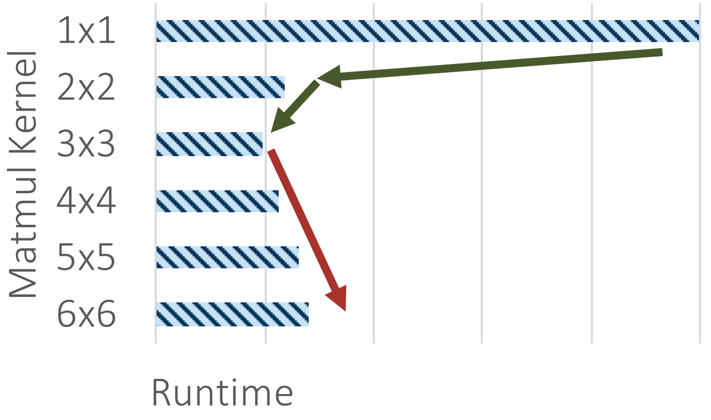
Year 2

WP3: uArch opt and implementation
Moving from version 0.1 the goal is to **develop an optimized microarchitecture**, design with a **target frequency in the GHz range** and support for the ISA extension explored in WP2. The design will **emphasize scalability** to large lane count **and high energy efficiency**. **PPA** of the resulting design will be **assessed**

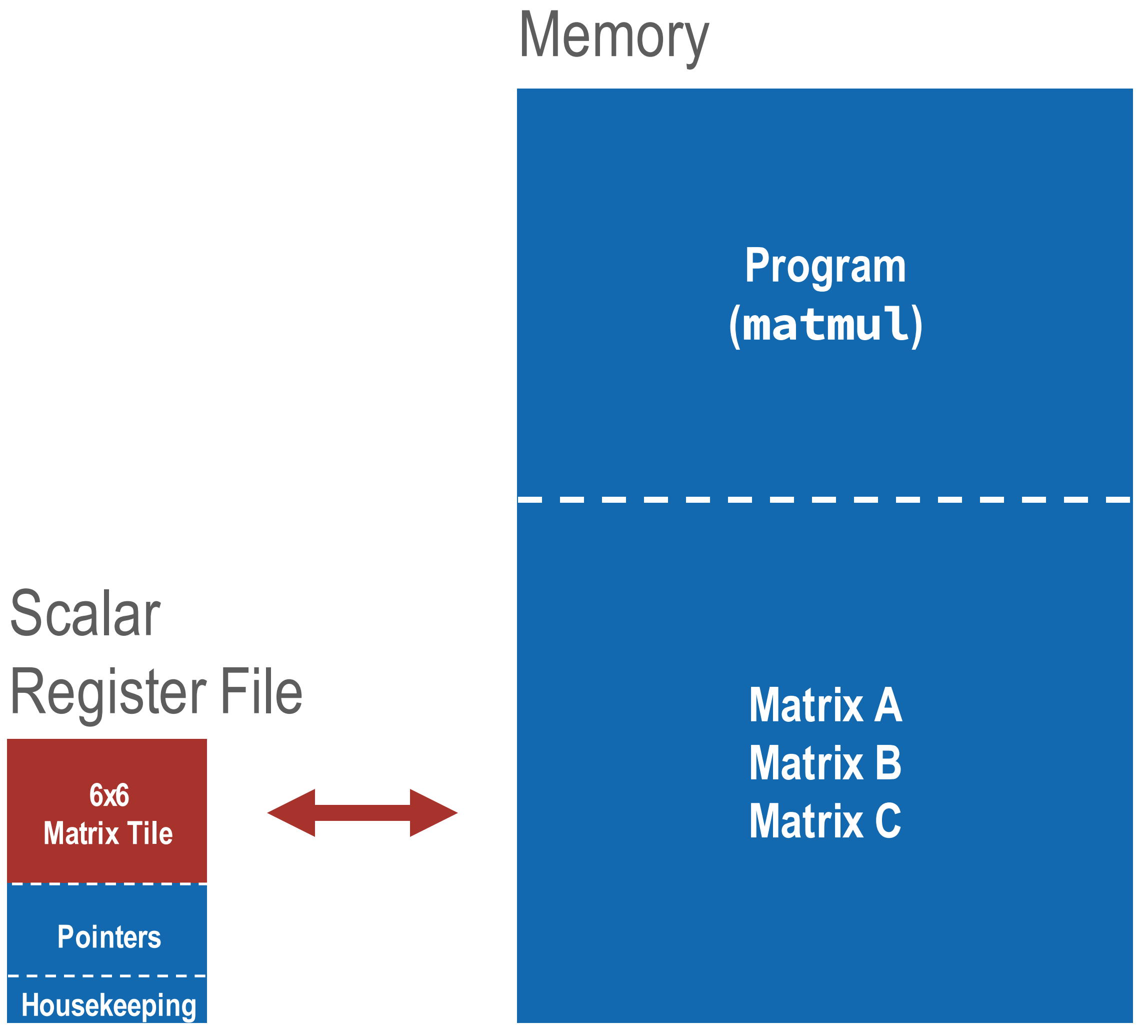
WP4: design space exploration, sw environment, open-source release
The main goal of this WP is to **verify the design** of VP4, provide **software support** (intrinsics, optimized libraries, preliminary support for vectorization).
An **open-source release of HW and SW** will be prepared with documentation.

Milestones: version 1.0 of Arav2 with instruction extensions and PPA analysis, open source release of HW and SW

Often need for more buffering space



The tile does not fit



Memory

Program
(`matmul`)

Matrix A
Matrix B
Matrix C

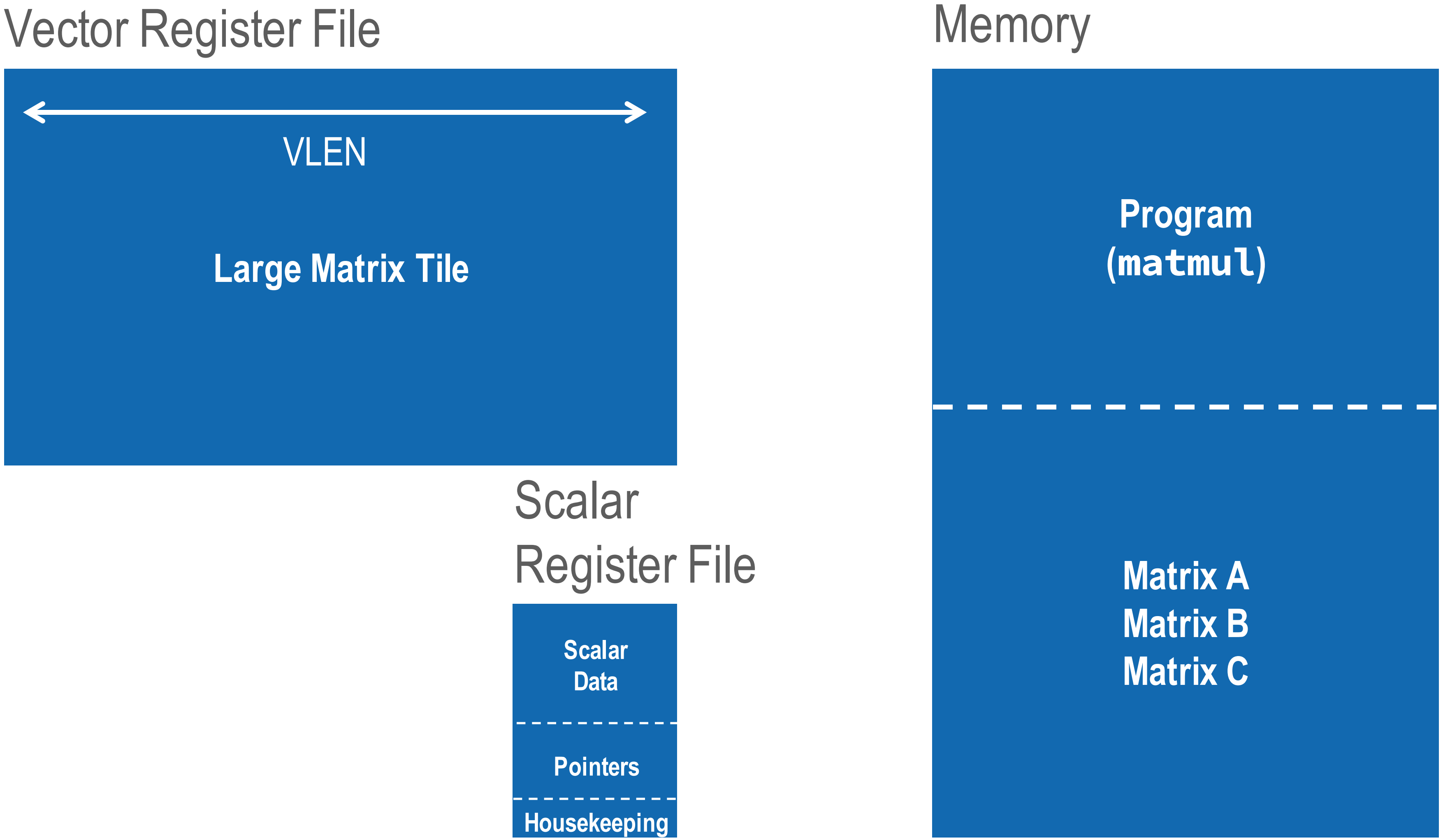
Scalar
Register File

6x6
Matrix Tile

Pointers

Housekeeping

Vector Register File (VRF) – L0 buffer



Vector Register File

VLEN

Large Matrix Tile

Scalar Register File

Scalar Data

Pointers

Housekeeping

Memory

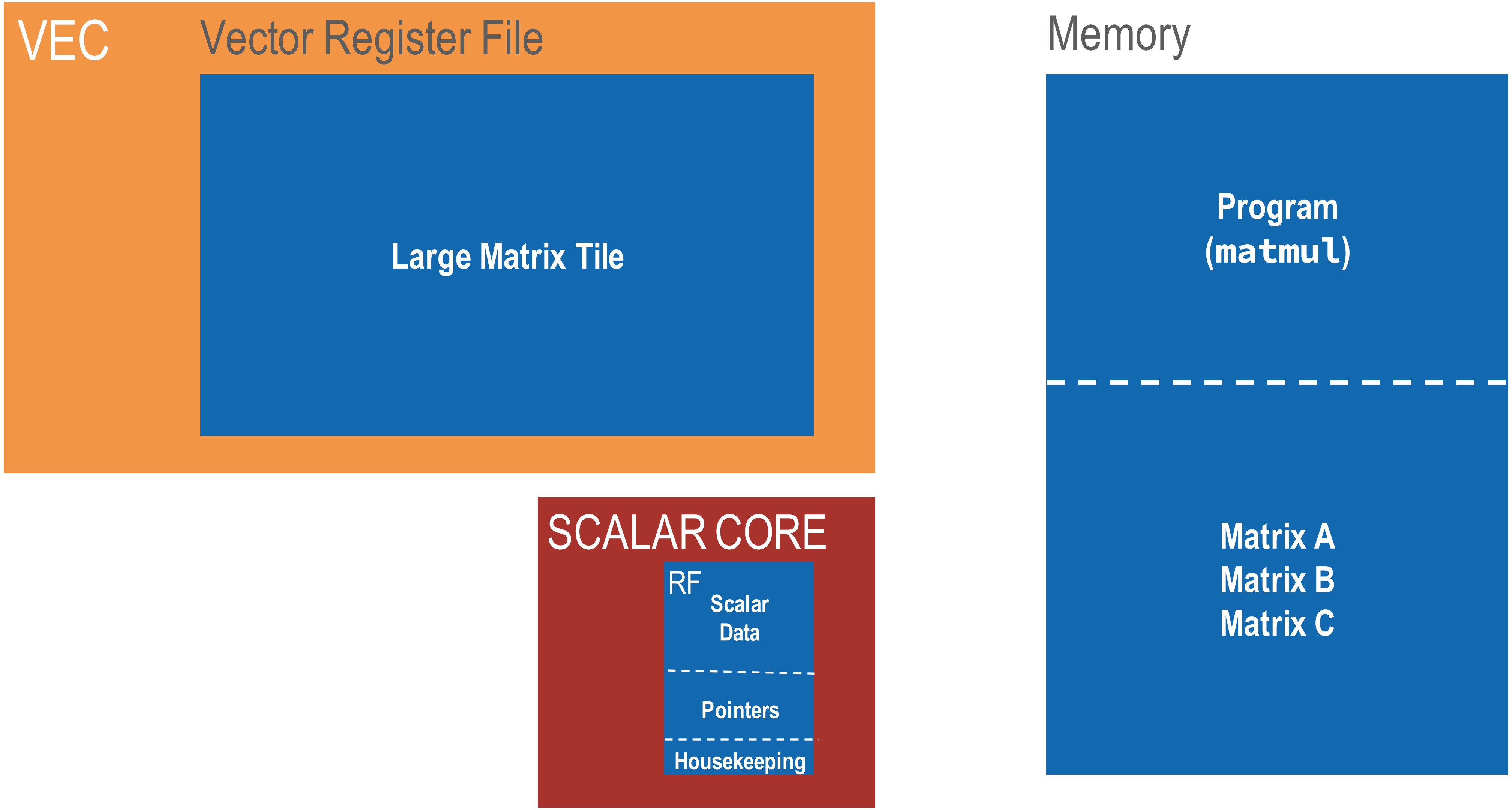
Program (`matmul`)

Matrix A

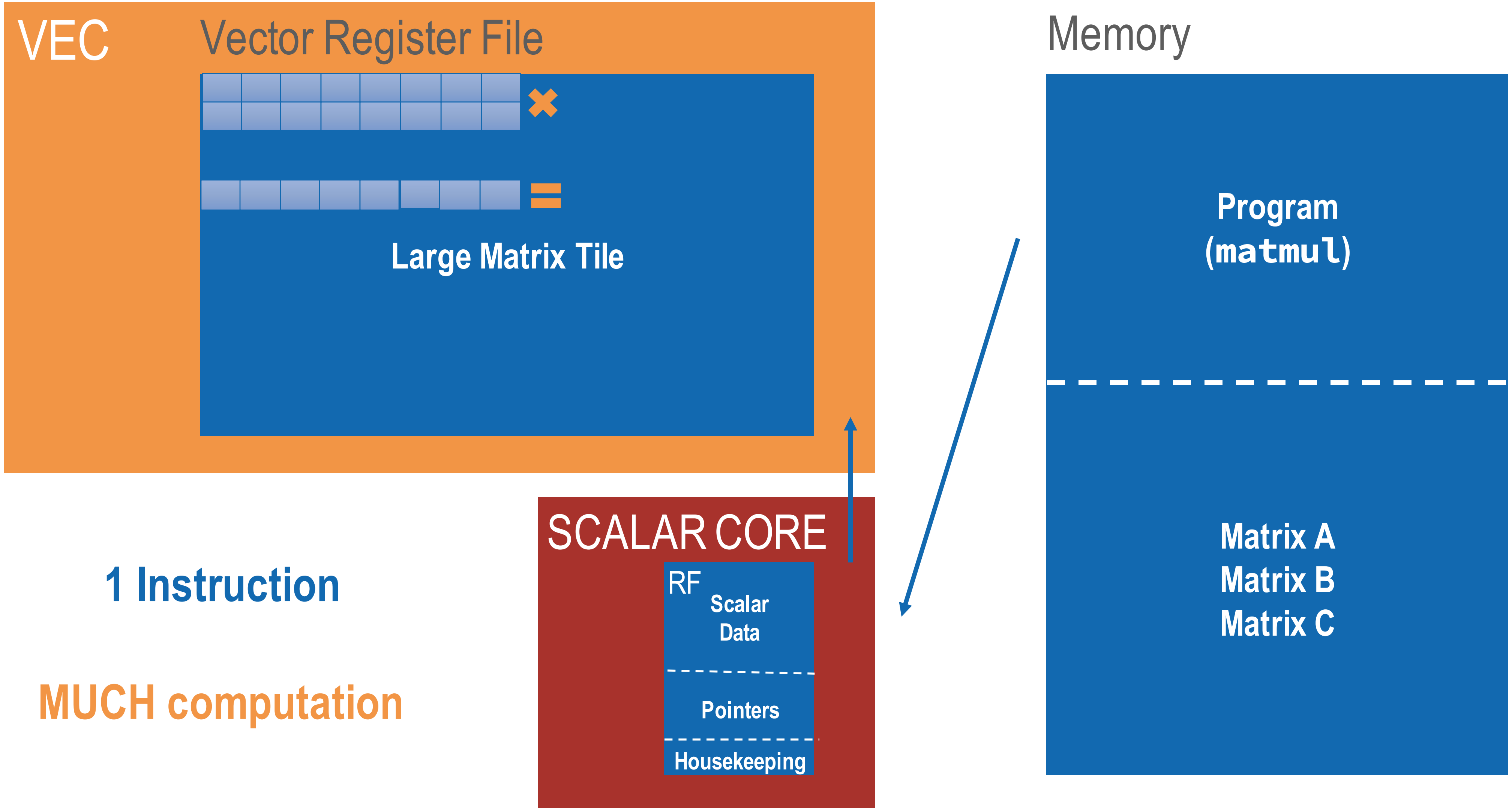
Matrix B

Matrix C

VRF – L0 buffer



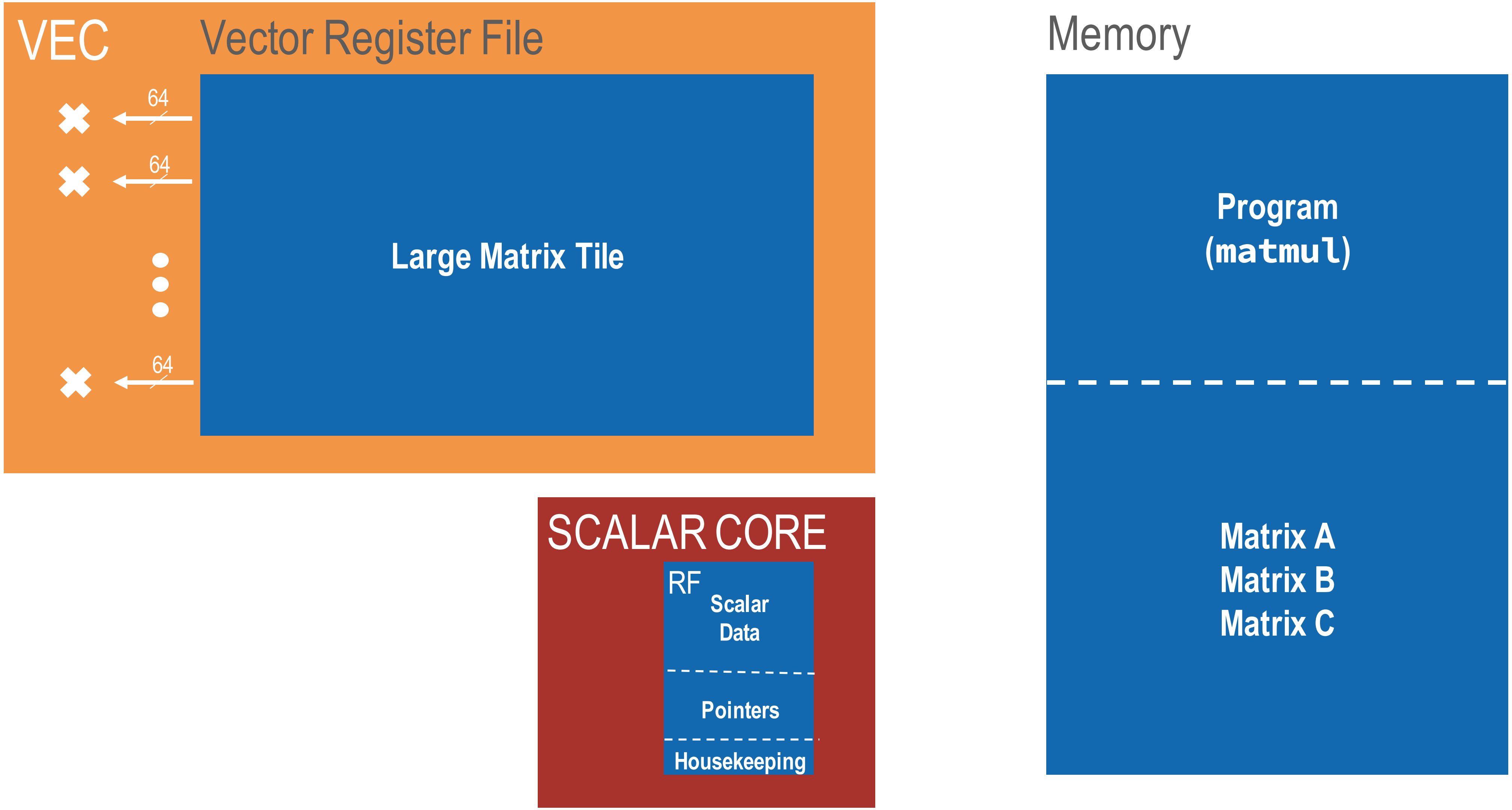
SIMD computation – tackle the VNB



1 Instruction

MUCH computation

SIMD computation – tackle the VNB



Vector Addition

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

Why Vector?

- Reduce Von-Neumann Bottleneck
- Higher buffering capabilities
- Vector length agnostic, reduced ISA

```
// Scalar code
for (int i = N; i > 0; --i) {
  ld a
  ld b
  add(c, a, b)
  st c
  // bump pointers
}
```

Vector Addition

Why Vector?

- Reduce Von-Neumann Bottleneck
- Higher buffering capabilities
- Vector length agnostic, reduced ISA

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

```
// Scalar code
for (int i = N; i > 0; --i) {
  ld a
  ld b
  add(c, a, b)
  st c
  // bump pointers
}


// Vector code
for (int vl = get_vl(N); N > 0; N -= vl) {
  vld va
  vld vb
  vadd(vc, va, vb)
  vst vc
  // bump pointers and get vl
}
```

Example of vector code

Why Vector?

- Reduce Von-Neumann Bottleneck
- Higher buffering capabilities
- Vector length agnostic, reduced ISA

Vector Addition

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

```
// Scalar code
for (int i = N; i > 0; --i) {
  ld a
  ld b
  add(c, a, b)
  st c
  // bump pointers
}
```

2N loads

N add

N stores

4N FETCHED, DECODED

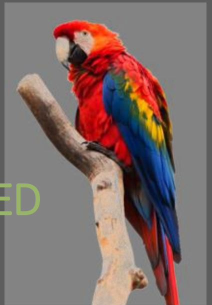
```
// Vector code
for (int vl = get_vl(N); N > 0; N -= vl) {
  vld va
  vld vb
  vadd(vc, va, vb)
  vst vc
  // bump pointers and get vl
}
```

2N/vl loads

N/vl add

N/vl stores

4N/vl FETCHED, DECODED



Example of vector code

Why Vector?

- Reduce Von-Neumann Bottleneck
- Higher buffering capabilities
- Vector length agnostic, reduced ISA

Lower pressure on memory!

Less power required!

Vector Addition

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

```
// Scalar code
for (int i = N; i > 0; --i) {
  ld a
  ld b
  add(c, a, b)
  st c
  // bump pointers
}
```

2N loads

N add

N stores

4N FETCHED, DECODED

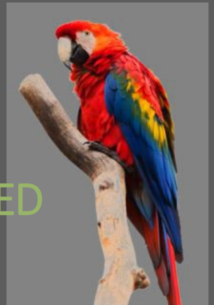
```
// Vector code
for (int vl = get_vl(N); N > 0; N -= vl) {
  vld va
  vld vb
  vadd(vc, va, vb)
  vst vc
  // bump pointers and get vl
}
```

2N/vl loads

N/vl add

N/vl stores

4N/vl FETCHED, DECODED

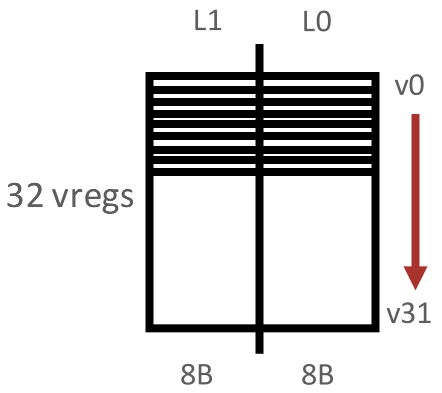


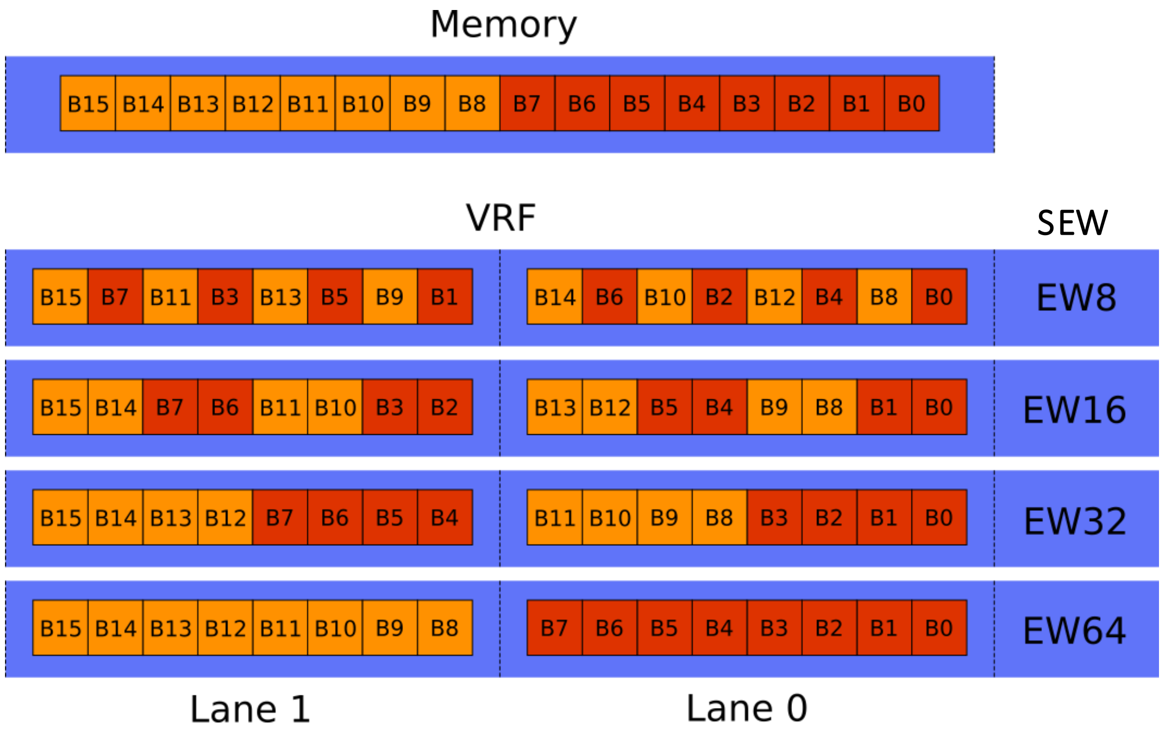
RVV < 1.0 - SLEN parameter

RVV < 1.0

- **VLEN**: size of a single vector register (bits)
- **SLEN**: size of a chunk of a vector register in a lane (bits)
- **SEW**: single element width (bits)
- SLEN = VLEN/#Lanes
- Lane-friendly
- Successive elements in successive lanes

Example:
VLEN = 128 bit
SLEN = 64 bit





Memory

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

VRF

Lane 1								Lane 0								SEW
B15	B7	B11	B3	B13	B5	B9	B1	B14	B6	B10	B2	B12	B4	B8	B0	EW8
B15	B14	B7	B6	B11	B10	B3	B2	B13	B12	B5	B4	B9	B8	B1	B0	EW16
B15	B14	B13	B12	B7	B6	B5	B4	B11	B10	B9	B8	B3	B2	B1	B0	EW32
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	EW64

RVV 1.0 - No SLEN parameter

RVV 1.0

- **VLEN**: size of a single vector register (bits)
- **SEW**: single element width (bits)
- VRF Byte layout == Memory Byte layout
- Non Lane-friendly

Widening instructions become a problem

- During a widening, elements would need to be moved across lanes

To ease widening instructions, we mimick an SLEN parameter.

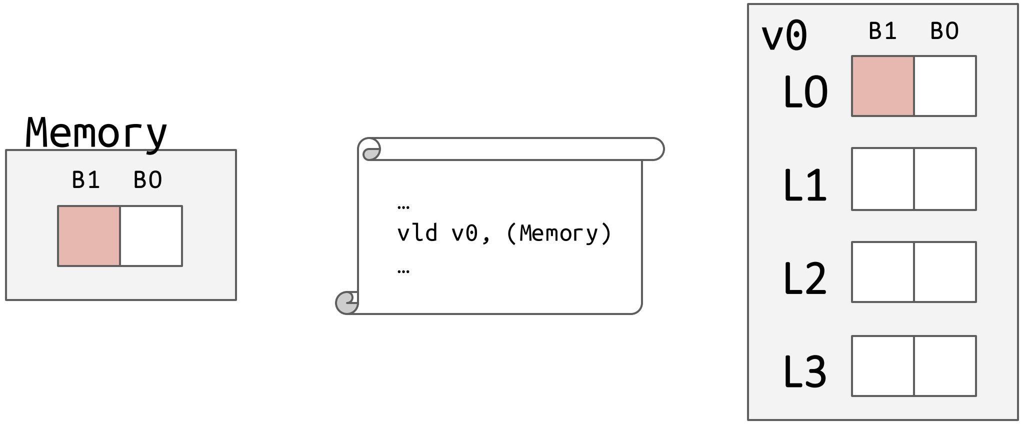
But we need to be able to reconstruct the vector as if it was buffered without SLEN!

Byte mapping (memory – VRF)

EW16

MAPPING

Memory(B1) - v0(B1, L0)

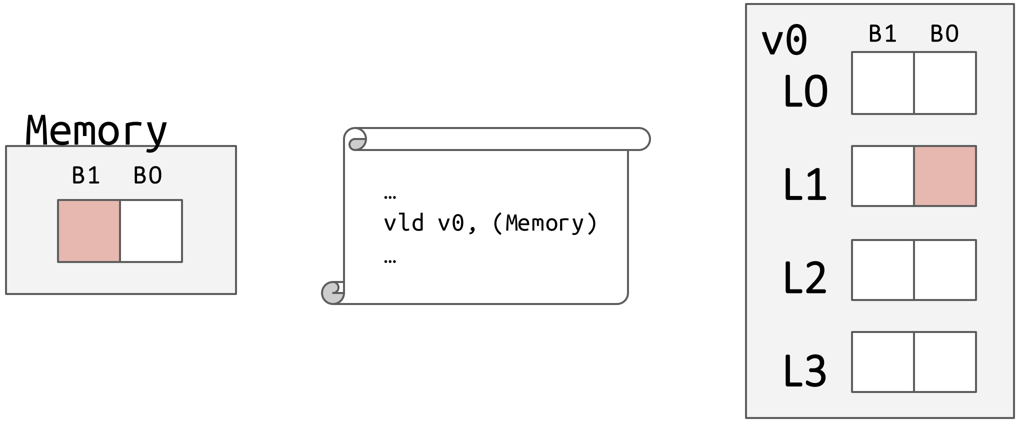


```
…
vld v0, (Memory)
…
```

EW8

MAPPING

Memory(B1) - v0(B0, L1)



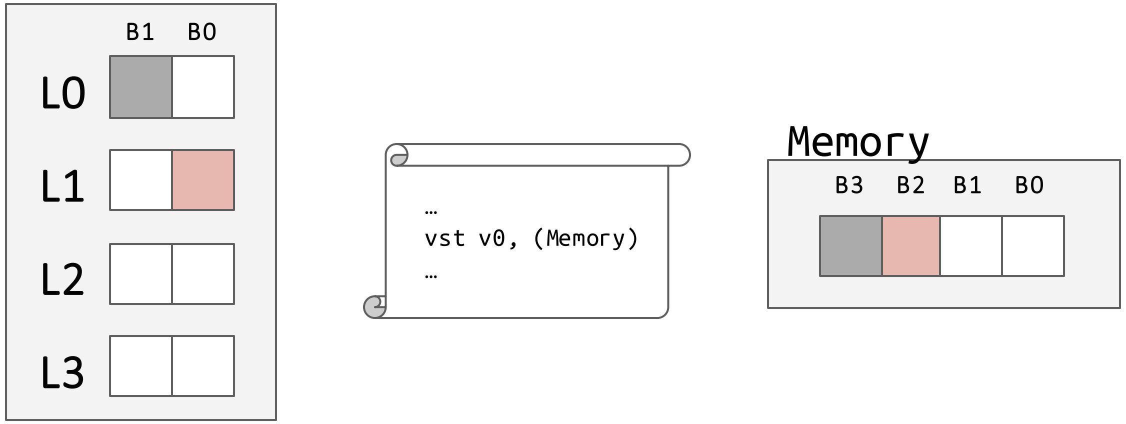
```
…
vld v0, (Memory)
…
```

Byte mapping (memory – VRF)

EW16

MAPPING

v0(B0, L1) -
Memory(B2)

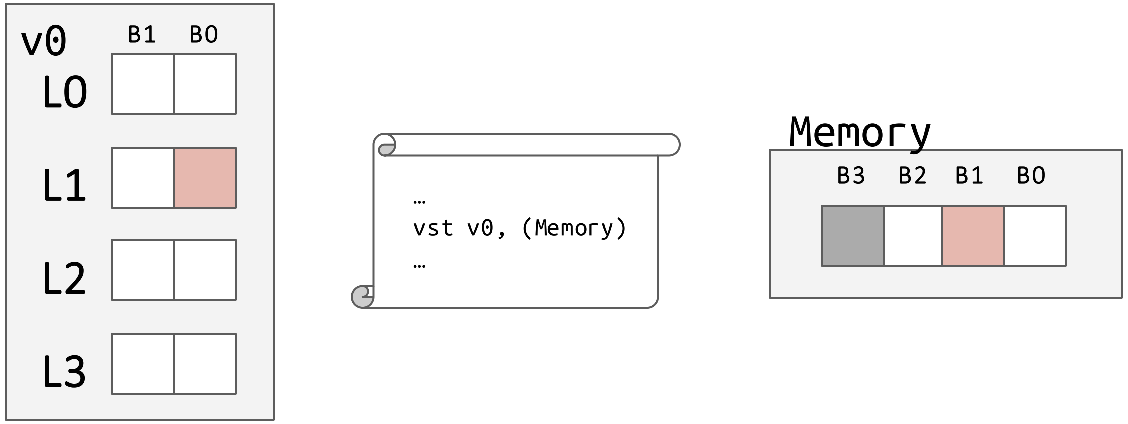


```
…
vst v0, (Memory)
…
```

EW8

MAPPING

v0(B0, L1) -
Memory(B1)



```
…
vst v0, (Memory)
…
```

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

MAPPING

v0 (EW16)

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

v0 (VRF, inside VU1.0)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

Ara VRF

- Vector with 1 elements
- 1 Byte/element

```
…
vld v0, (Memory)
…
```

MAPPING

v0 (EW8)

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
															A0

v0 (VRF, inside VU1.0)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

Ara VRF

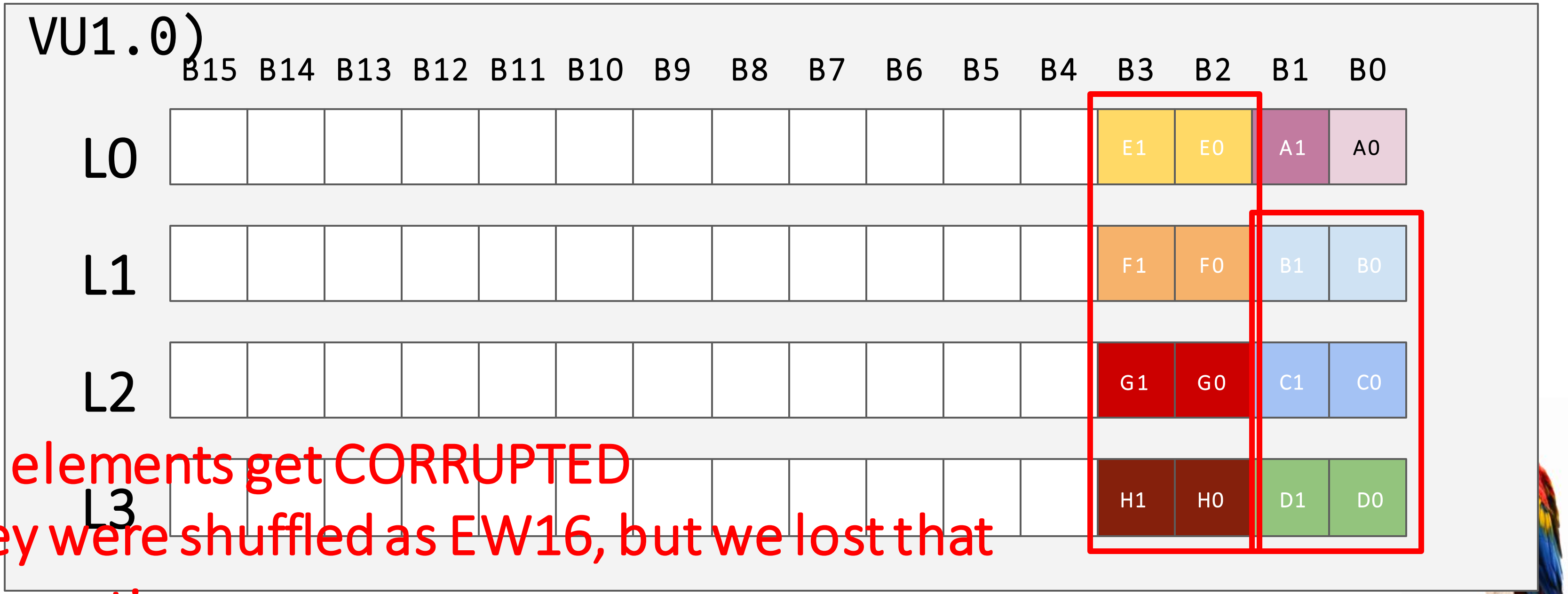
- Vector with 1 elements
- 1 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
								▨	▨	▨	▨	▨	▨	▨	A0

v0 (VRF, inside VU1.0)



	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

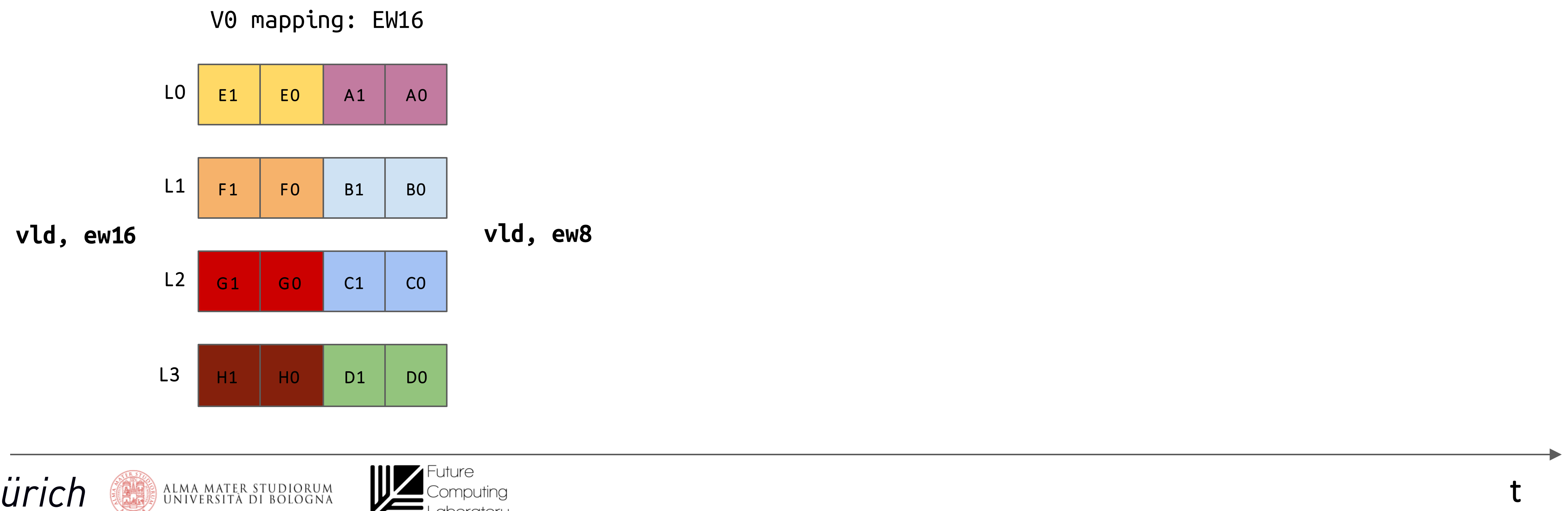
MAPPING

v0 (EW8)

Tail elements get CORRUPTED
They were shuffled as EW16, but we lost that information

Reshuffle (undisturbed policy)

- New data-width in the destination and tail elements exposed?
  - Reshuffle destination with new encoding!
  - Byte mapping is preserved!



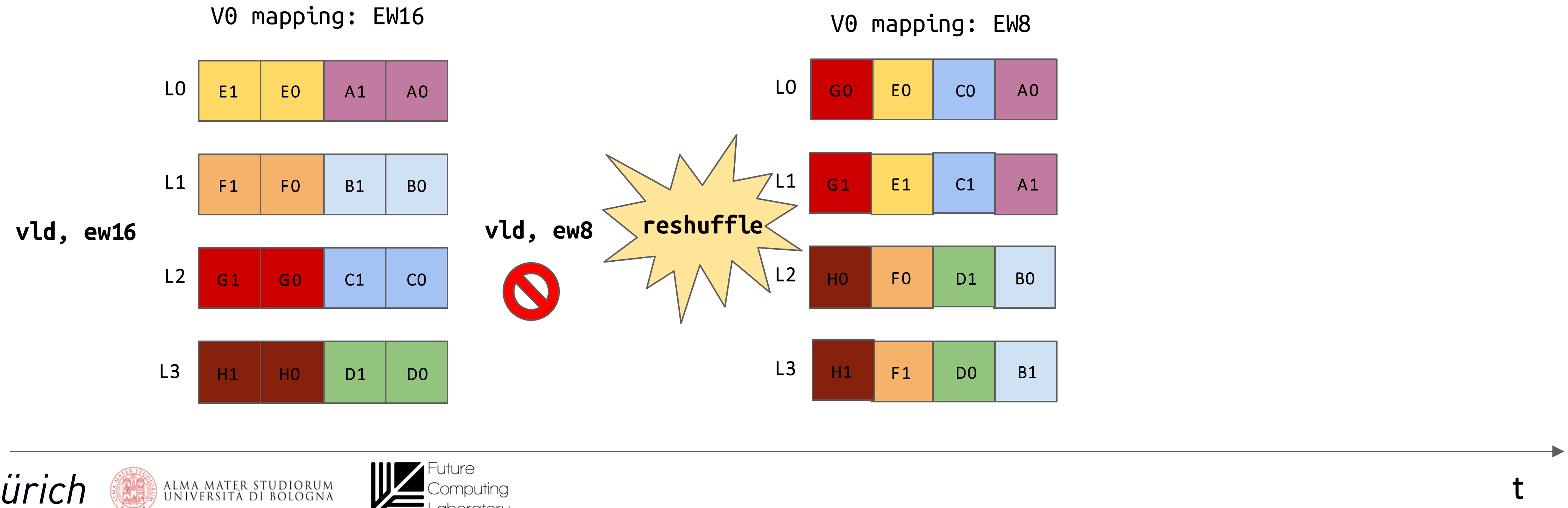
V0 mapping: EW16

L0	E1	E0	A1	A0
L1	F1	F0	B1	B0
L2	G1	G0	C1	C0
L3	H1	H0	D1	D0

Reshuffle (undisturbed policy)

- New data-width in the destination and tail elements exposed?
  - Reshuffle destination with new encoding!
  - Byte mapping is preserved!

- Reshuffle injected when needed



V0 mapping: EW16

L0	E1	E0	A1	A0
L1	F1	F0	B1	B0
L2	G1	G0	C1	C0
L3	H1	H0	D1	D0

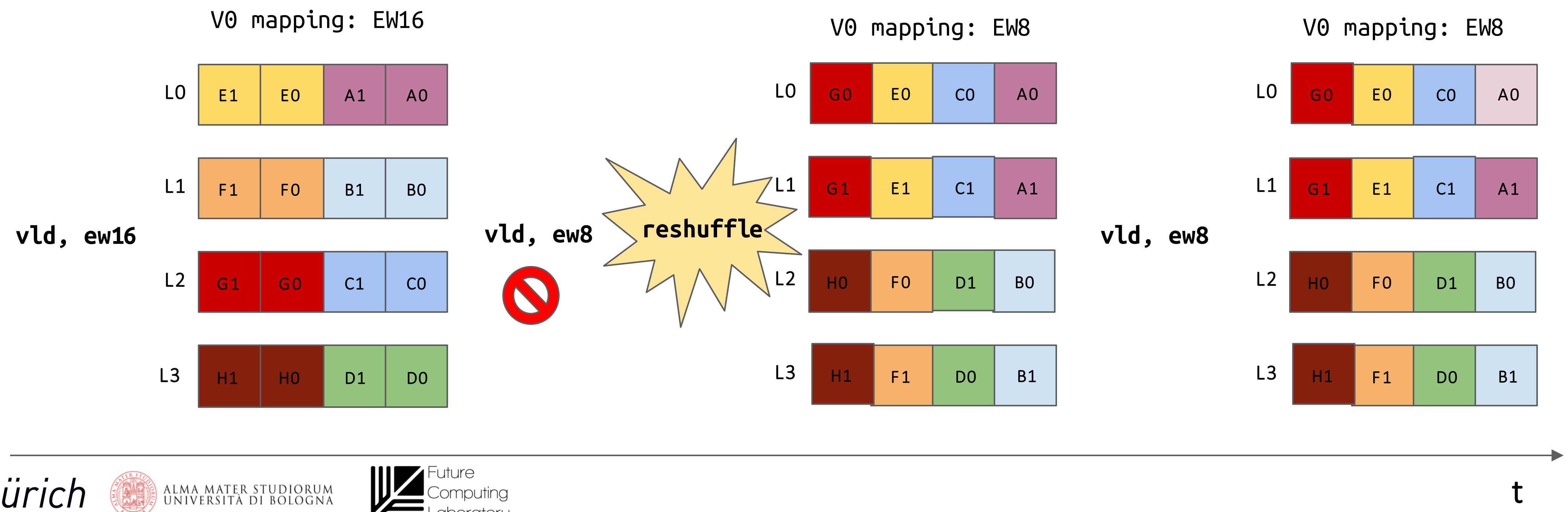
V0 mapping: EW8

L0	G0	E0	C0	A0
L1	G1	E1	C1	A1
L2	H0	F0	D1	B0
L3	H1	F1	D0	B1

Reshuffle (undisturbed policy)

- New data-width in the destination and tail elements exposed?
  - Reshuffle destination with new encoding!
  - Byte mapping is preserved!

- Reshuffle injected when needed



V0 mapping: EW16

L0	E1	E0	A1	A0
L1	F1	F0	B1	B0
L2	G1	G0	C1	C0
L3	H1	H0	D1	D0

V0 mapping: EW8

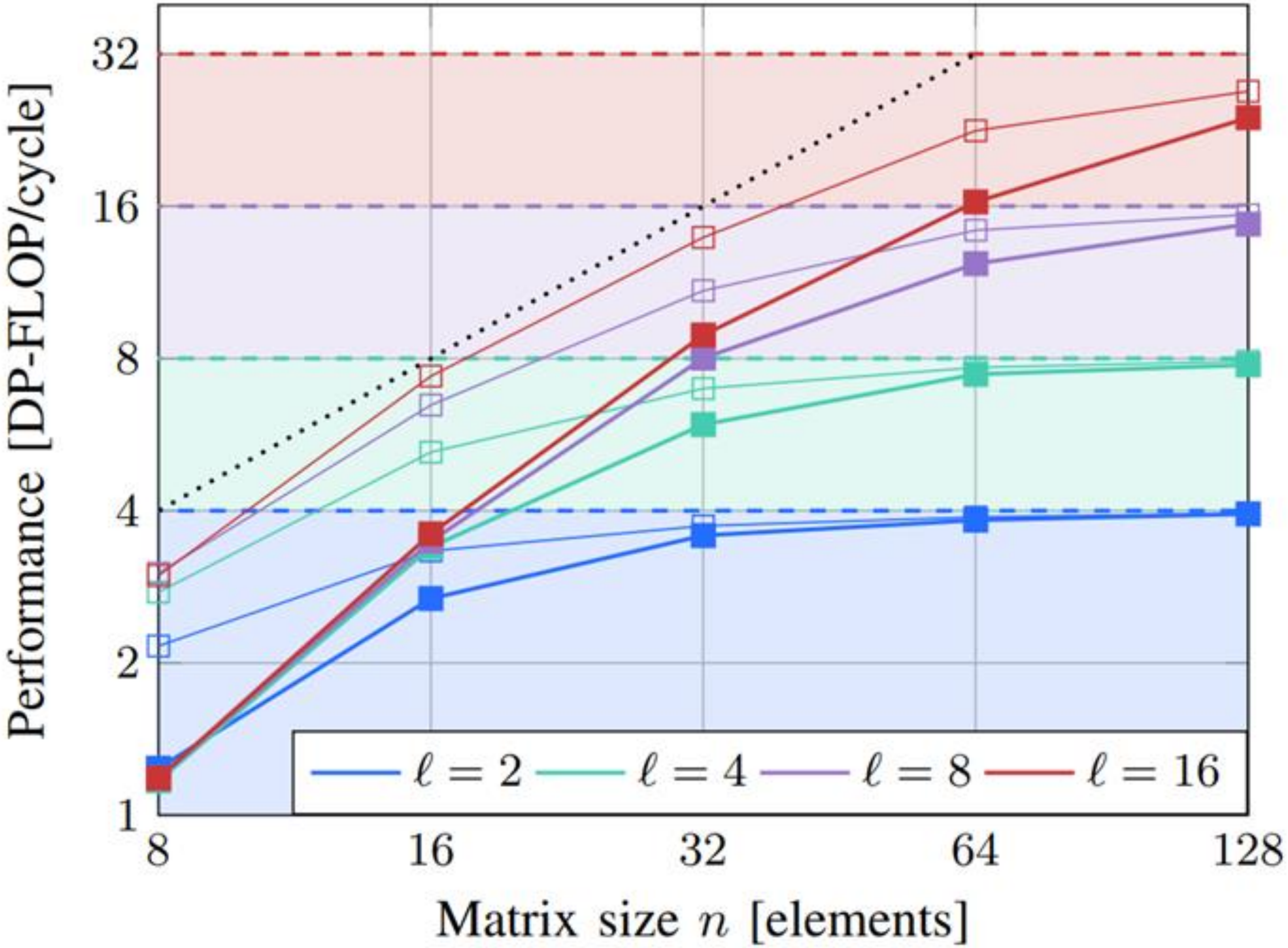
L0	G0	E0	C0	A0
L1	G1	E1	C1	A1
L2	H0	F0	D1	B0
L3	H1	F1	D0	B1

V0 mapping: EW8

L0	G0	E0	C0	A0
L1	G1	E1	C1	A1
L2	H0	F0	D1	B0
L3	H1	F1	D0	B1

Scalar Core + Caches effect on performance

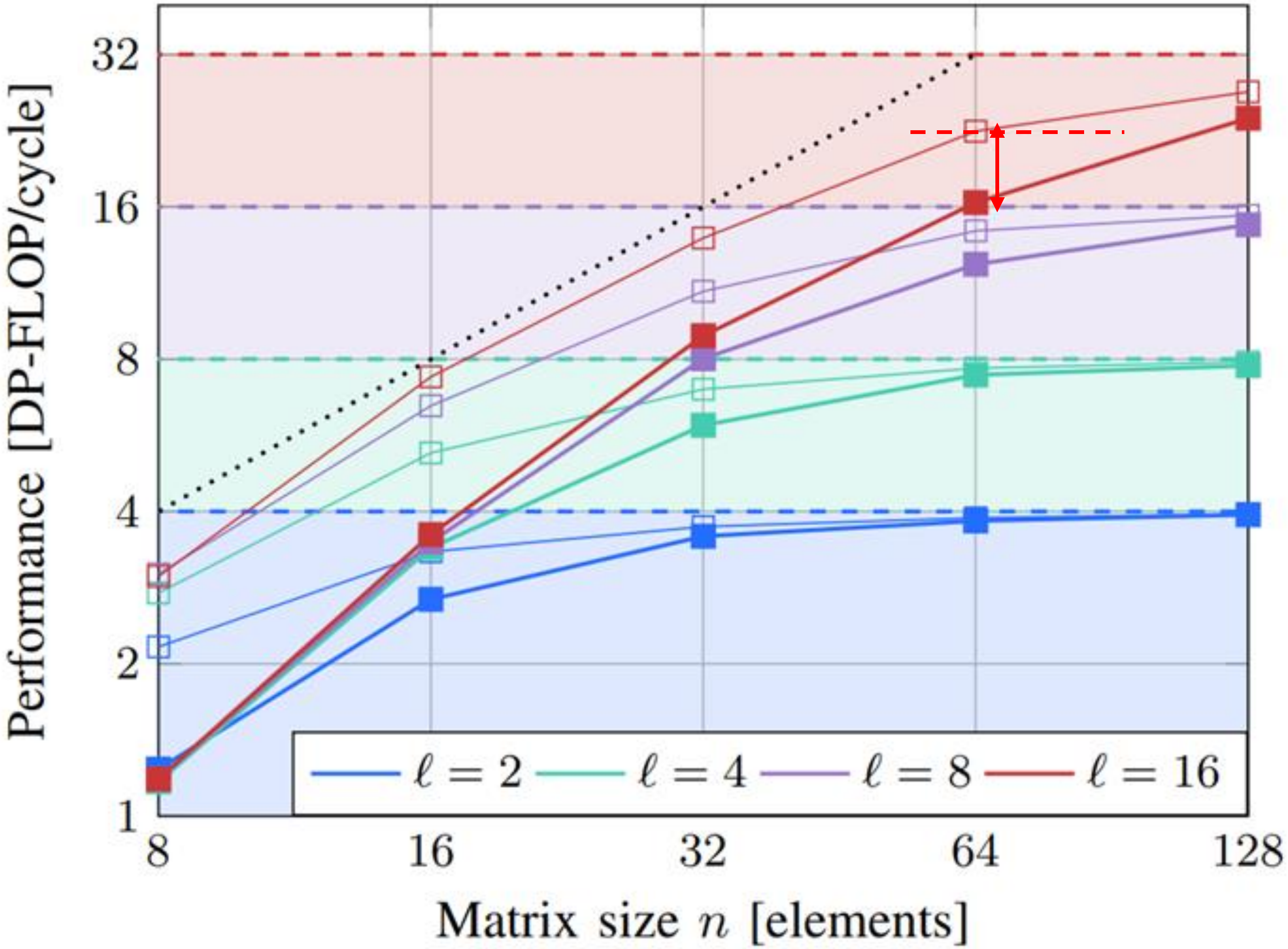
- **Replace CVA6** with **ideal dispatcher**
  - FIFO with V instructions
  - Ideal issue rate
  - Narrow lines on plot



Scalar Core + Caches effect on performance

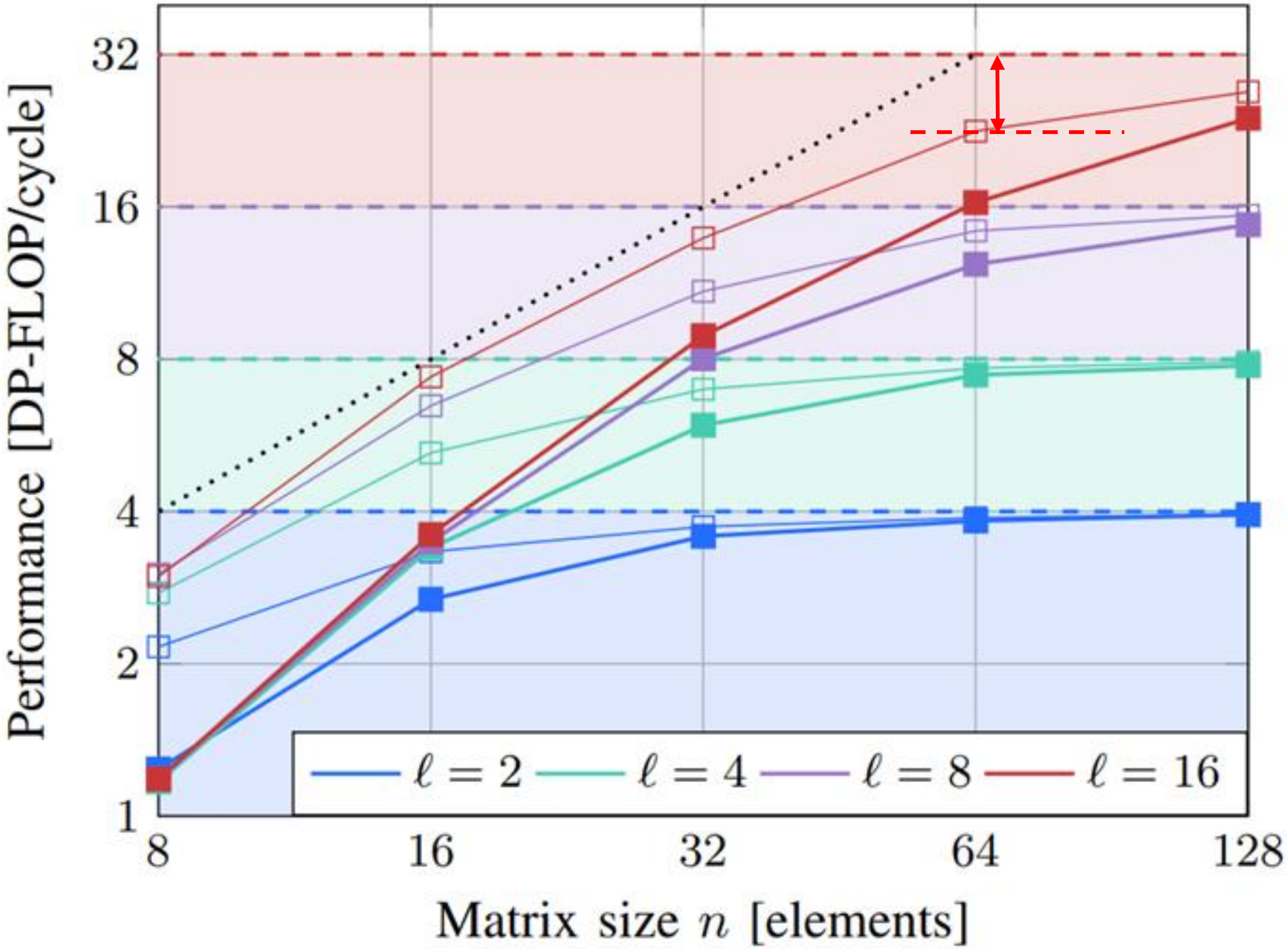
- **Replace CVA6** with **ideal dispatcher**
  - FIFO with V instructions
  - Ideal issue rate
  - Narrow lines on plot

- Show **CVA6 + $ non-idealities**



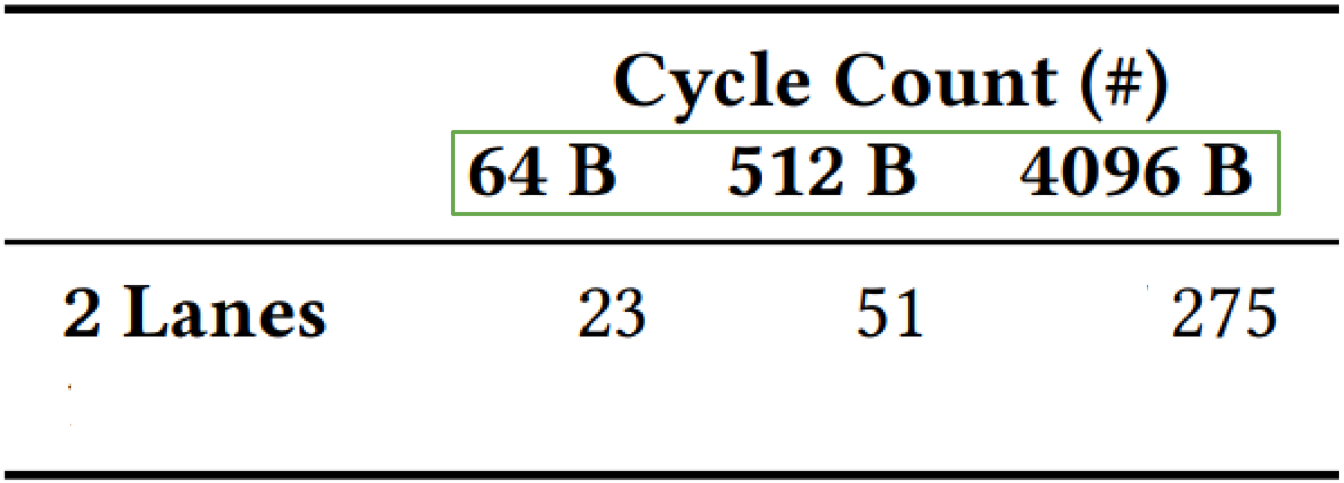
Scalar Core + Caches effect on performance

- **Replace CVA6** with **ideal dispatcher**
  - FIFO with V instructions
  - Ideal issue rate
  - Narrow lines on plot
- Show **CVA6 + $ non-idealities**
- Show **VU1.0 non-idealities**



Reduction performance

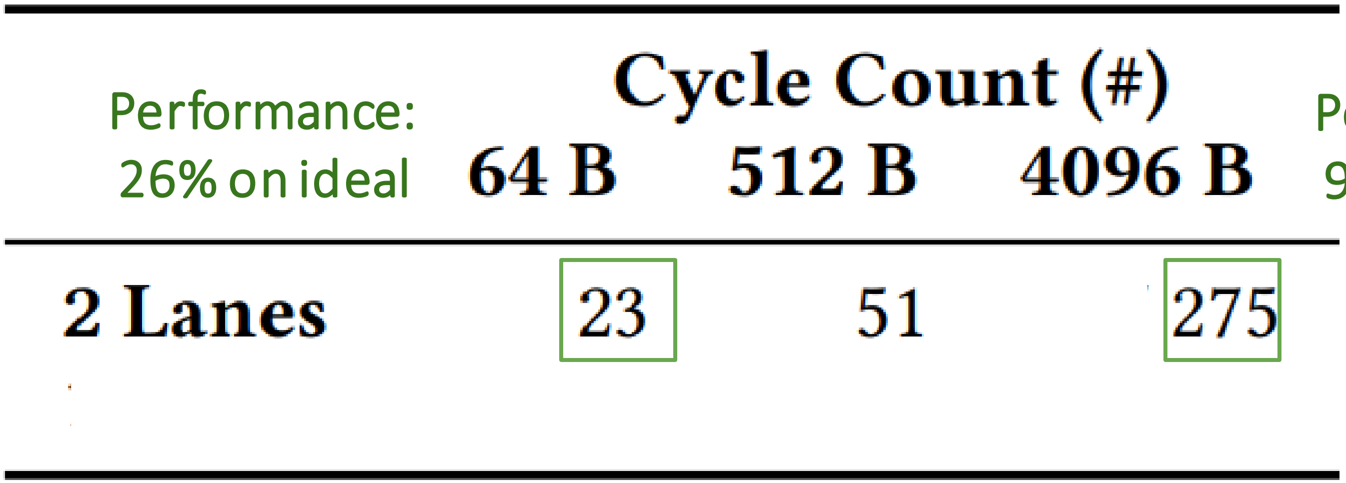
- **`dotp`** benchmark (chained `vmul`, `vredsum`)
  - Varying vector length in Byte



	Cycle Count (#)		
	64 B	**512 B**	**4096 B**
2 Lanes	23	51	275

Reduction performance

- **`dotp`** benchmark (chained `vmul`, `vredsum`)
  - Varying vector length in Byte



	Cycle Count (#)		
	64 B	512 B	4096 B
2 Lanes	23	51	275

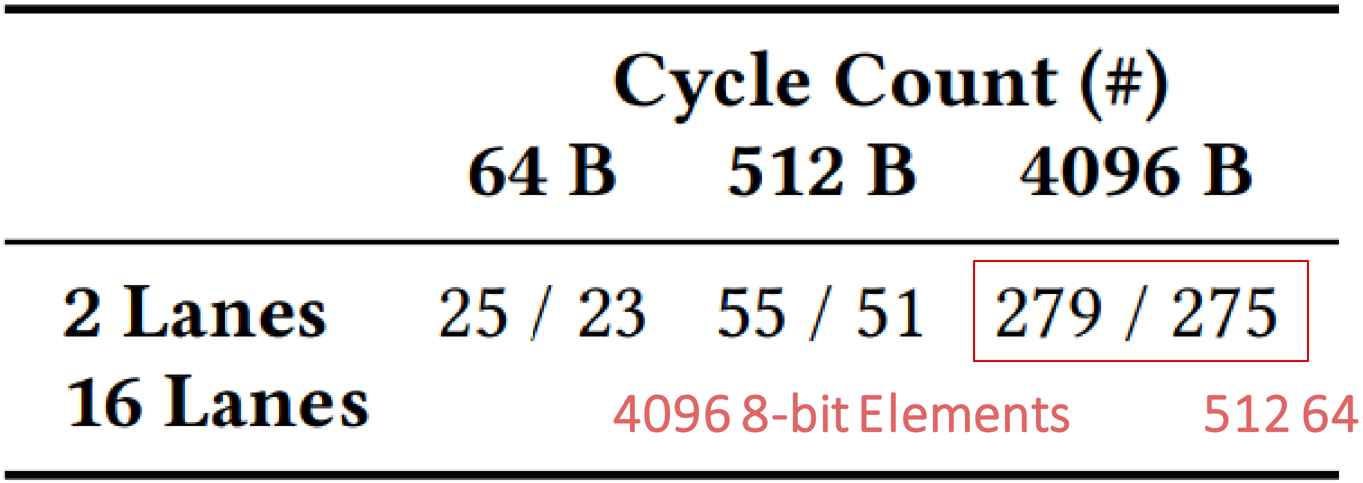
Performance: 26% on ideal

Performance: 94% on ideal

Longer vectors == Higher Efficiency!!

Reduction performance

- **`dotp`** benchmark (chained `vmul`, `vredsum`)
  - Varying vector length in Byte
  - Varying vector element size: *8-bit / 64-bit*



	Cycle Count (#)		
	64 B	**512 B**	**4096 B**
2 Lanes	25 / 23	55 / 51	279 / 275
16 Lanes			

4096 8-bit Elements

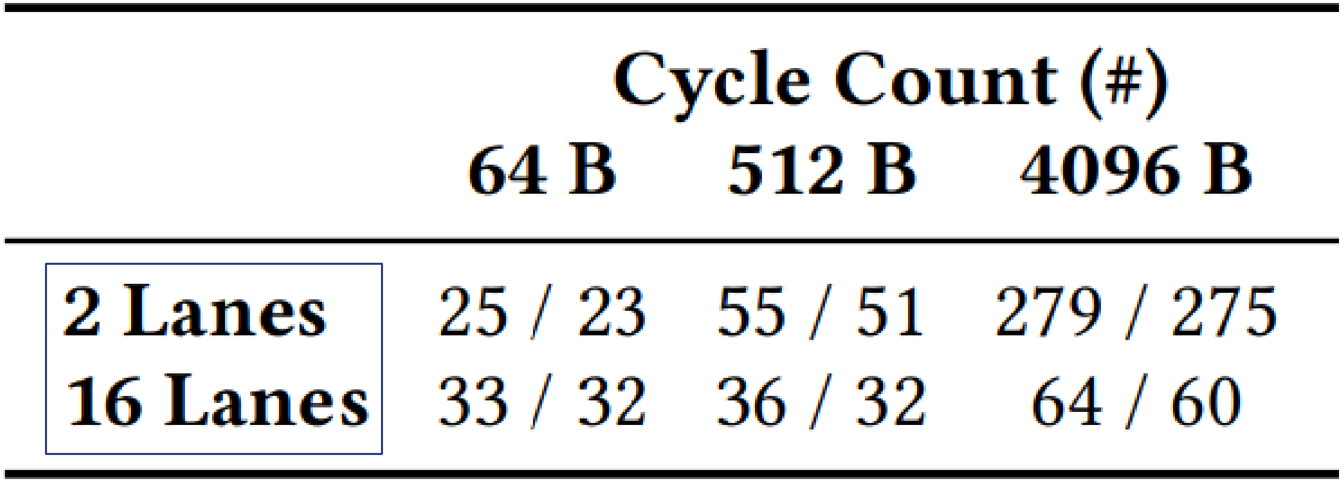
512 64-bit Elements

Cycle count almost independent on element size

8-bit elements == ~8x the throughput of 64-bit elements

Reduction performance

- **`dotp`** benchmark (chained `vmul`, `vredsum`)
  - Varying vector length in Byte
  - Varying vector element size: *8-bit / 64-bit*
  - Varying number of lanes

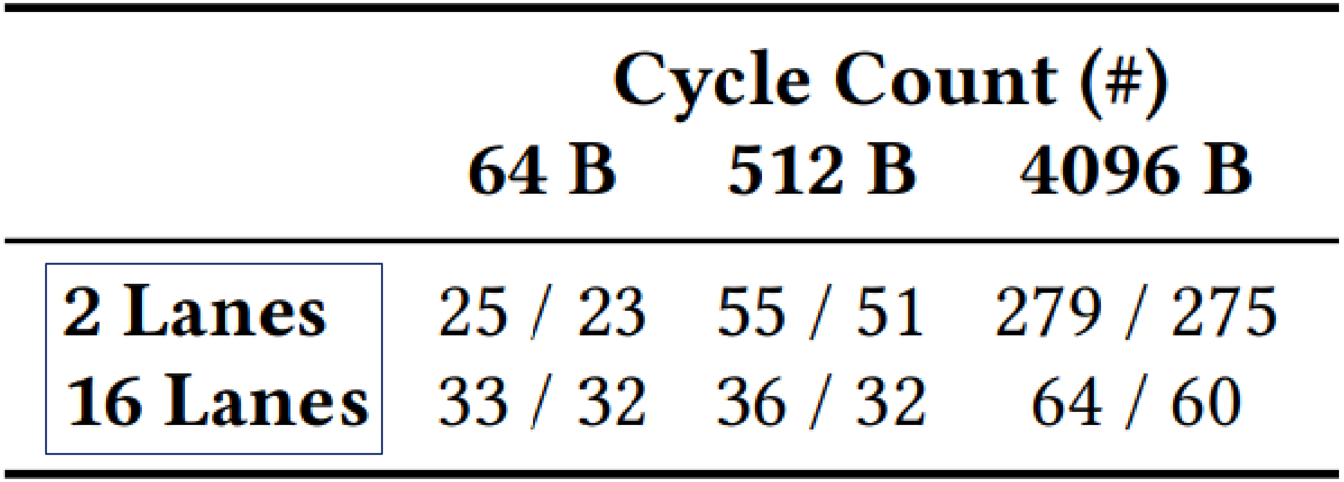


	Cycle Count (#)		
	64 B	**512 B**	**4096 B**
2 Lanes	25 / 23	55 / 51	279 / 275
16 Lanes	33 / 32	36 / 32	64 / 60

High overhead by inter-lanes reduction phase for shorter vectors or more lanes

Reduction performance

- **`dotp`** benchmark (chained `vmul`, `vredsum`)
  - Varying vector length in Byte
  - Varying vector element size: *8-bit / 64-bit*
  - Varying number of lanes



	Cycle Count (#)		
	64 B	**512 B**	**4096 B**
2 Lanes	25 / 23	55 / 51	279 / 275
16 Lanes	33 / 32	36 / 32	64 / 60

High overhead by inter-lanes reduction phase for shorter vectors or more lanes

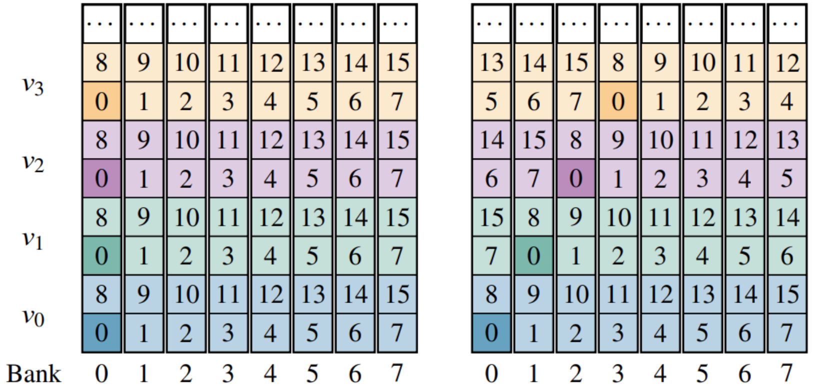
Barber's Pole layout

Ara RVV 0.5
Barber's Pole layout to speed up operand fetch

Ara RVV 1.0
No Barber's Pole

e.g., on matmul

- Small positive effect on short/medium vectors
  - No bank conflicts upon first operands fetch
- Small negative effect on medium/long vectors
  - More bank conflicts on different instructions



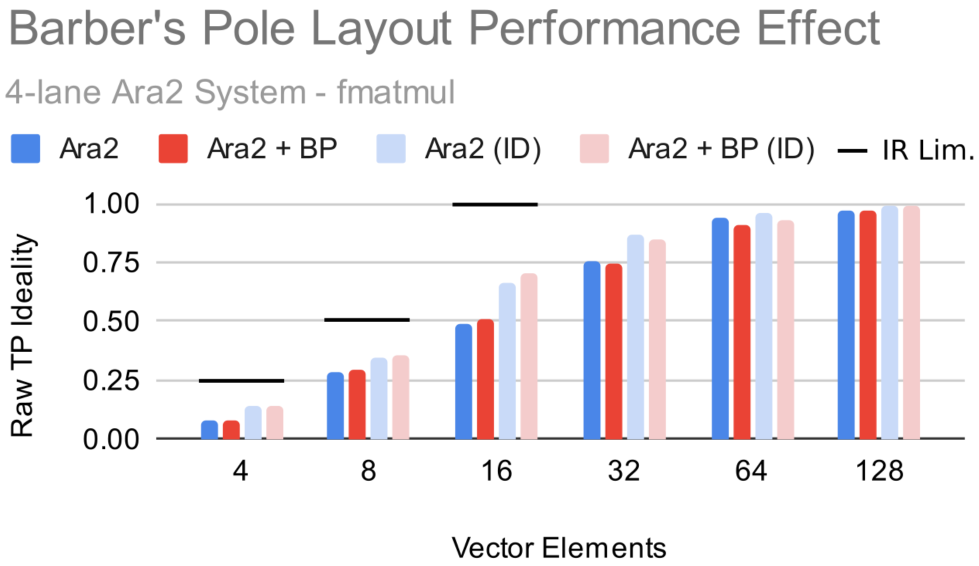
	...	...	...	...	...	...	...	...
	8	9	10	11	12	13	14	15
v_3	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
v_2	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
v_1	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
v_0	0	1	2	3	4	5	6	7
Bank	0	1	2	3	4	5	6	7

(a) Without "barber's pole" shift.

...	...	...	...	...	...	...	...
13	14	15	8	9	10	11	12
5	6	7	0	1	2	3	4
14	15	8	9	10	11	12	13
6	7	0	1	2	3	4	5
15	8	9	10	11	12	13	14
7	0	1	2	3	4	5	6
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

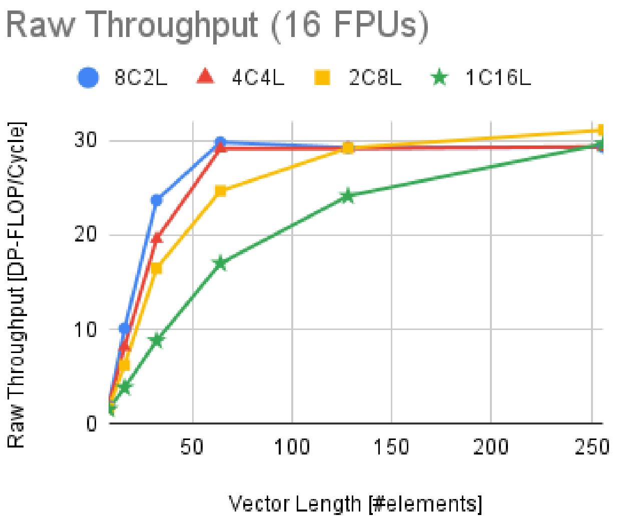
(b) With "barber's pole" shift.

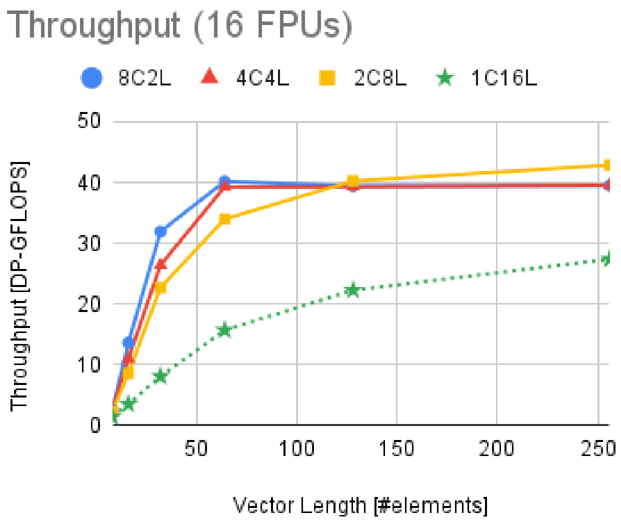
M. Cavalcante et. Al, "Ara: A 1 GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI"

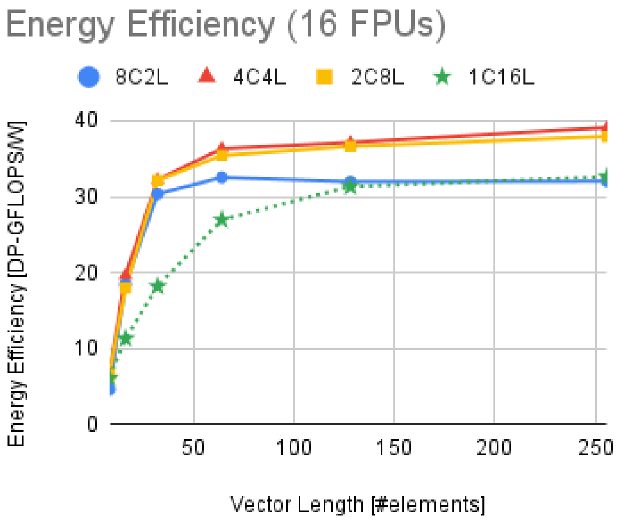


Multi-core performance, throughput, efficiency

- System with 16 FPUs
- Example: 8C2L = 8 (Ara+CVA6) instances


Raw Throughput (16 FPUs)


Throughput (16 FPUs)


Energy Efficiency (16 FPUs)

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA Future Computing Laboratory