ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

HW/SW Co-design of Energy Efficient RISC-V Edge AI Platforms

ACACES 2026, Fiuggi (Italy), 12-17 July 2026

Part 1: The RISC-V ISA and Edge Processors

Angelo Garofalo angelo.garofalo@unibo.it
agarofalo@iis.ee.ethz.ch

pulp-platform.org
@pulp_platform
company/pulp-platform
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

The PULP Team

Wide research team spread across University of Bologna and ETH Zurich
Led by Prof. Luca Benini
https://pulp-platform.org/team.html

Research focused on Open-source Energy-Efficient Computing Systems

Covering a vertical stack, from SW down to Chip Design

In 12 years PULP team has designed more than 60 chips

RISC-V and open-source hardware have been instrumental in our success

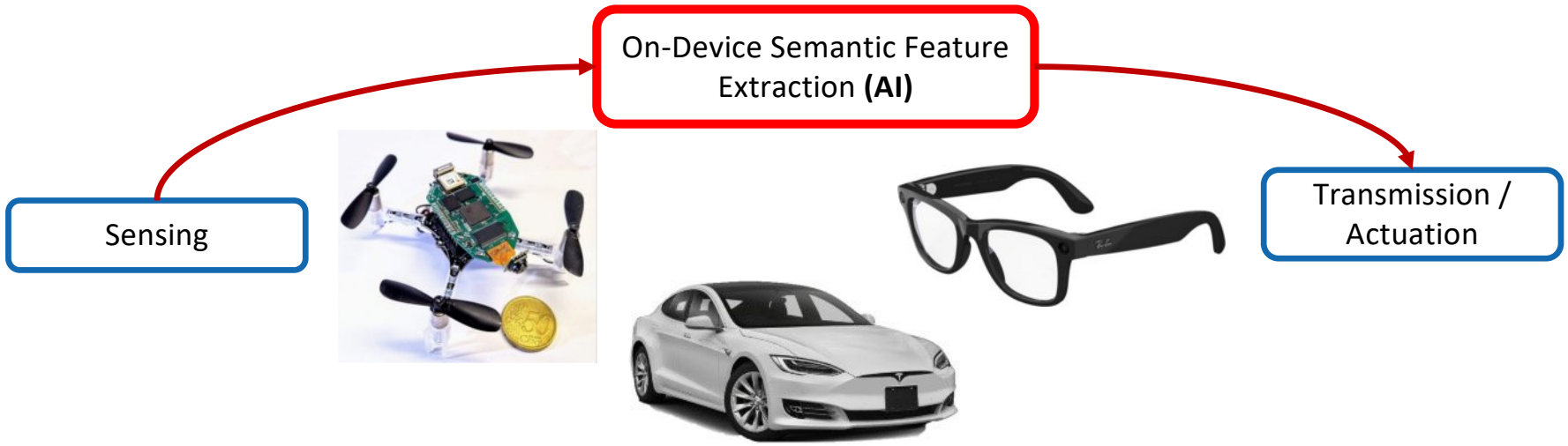
What is this Course About?

- **Part1: The RISC-V ISA and Edge Processors**
  - Introduction to RISC-V and processors' implementation
- **Part 2: Domain-Specialized RISC-V Processors for DSP and AI**
  - Methodology of ISA extensions; Examples on DSP and AI use-cases
- **Part 3: Parallel Ultra Low Power Cluster and Vector Lockstep Execution Modes**
  - Parallel Ultra Low-Power Cluster: Architectures, Optimizations and Limitations
- **Part 4: Addressing the Von Neumann Bottleneck with Heterogeneous Clusters and Streaming Clusters**
  - Extending the PULP cluster with accelerators: digital & analog
- **Part 5: RISC-V Vector Processors: Advantages and Drawbacks for Edge AI**
  - RISC-V Vector ISA and energy-efficient micro-architectural implementations

Outline of Part1

- **Introduction to the Edge AI Challenges**
  - The quest for flexible AI processors
  - The high power-performance proportionality goal
- **Introdution to RISC-V**
  - The advantages of RISC-V and the ecosystem
  - The RISC-V Extensions as composable computer architectures
- **How to build RISC-V Processors**
  - Low-end micro-architecture for control applications
  - Mid-end micro-architecture for DSP/TinyML
- **Summary**

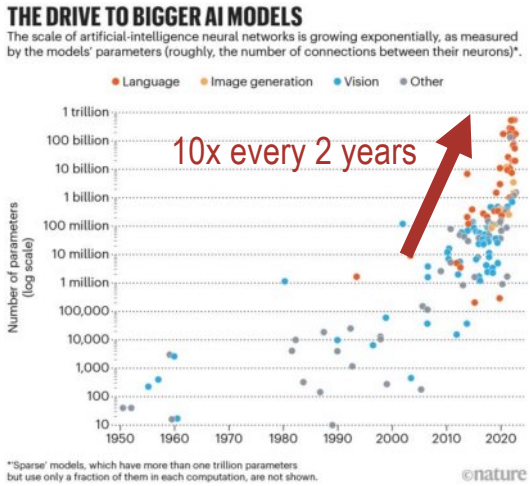
Energy-Efficient Embodied AI

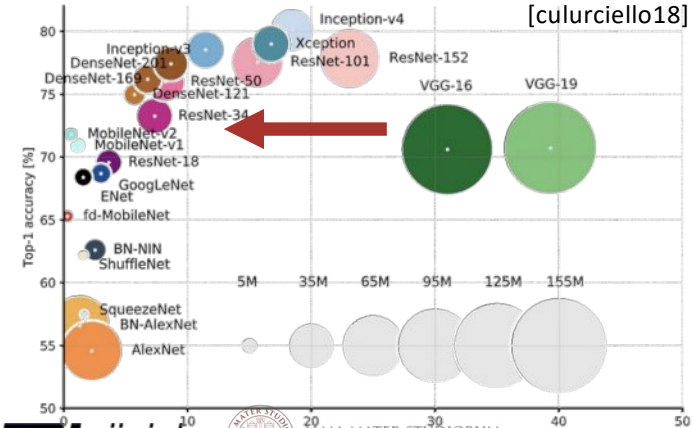


- **Computing requirements/challenges for Edge AI**
  - AI capabilities under tight power (passive cooling, 100mW-1Ws of budget)
  - Real-time constraints (Latency as important as Throughput)
  - Limited area budget (affects on chip memory)
  - Not only about AI execution (workload heterogeneity, pre-processing, control tasks)

TinyML → low-power, embeddable, low-cost edge AI Processors (not necessarily “small DNNs”)

Optimizing AI Deployment for Edge Devices



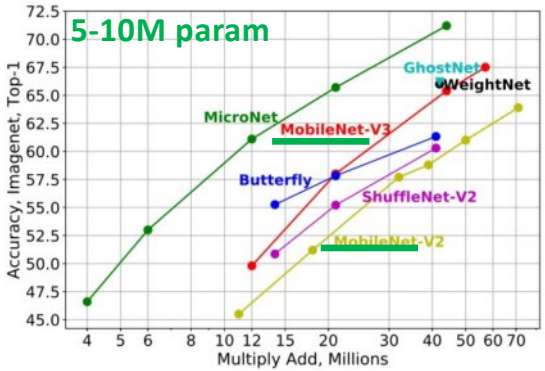


- **DNNs/AI** deployed at the edge **must be designed specifically** for low-power, constrained environments (lightweight AI models, quantization, pruning, …)
- Efficient edge deployment requires **specialized hardware** architectures optimized to run these lightweight networks efficiently.
- Effective deployment depends on comprehensive **Hardware-Software Co-design**, ensuring performance, energy efficiency, and reliability at the edge.

A Fast Evolving Model Zoo

Tiny Devices (IoT) → More Powerful edge Systems

Quantized Neural Networks (QNNs)

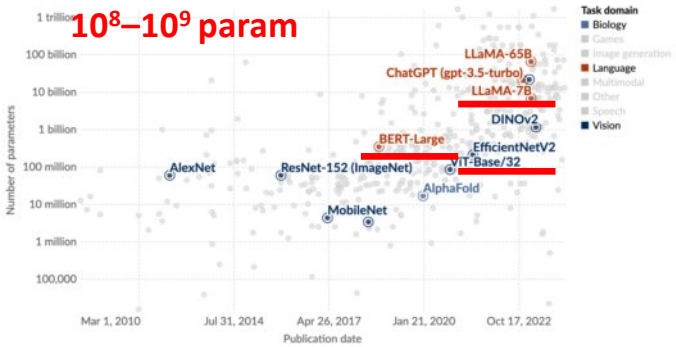


High OP/B ratio
Massive Parallelism
MAC-dominated

- Low precision is OK
- 8-/4-/2-b **integer** arithmetic
  - Including mixed-precision

[Rusci et al., PML&S (2020)]

Edge Transformers



- **BF16, FP16, FP8** is the standard
- Micro-scaling fmts for reduced footprint at no accuracy loss

[Rouhani et al., arxiv23]
[Yang et al., ARITH25]

Edge AI SoCs must provide: High Performance-Power Proportionality & Flexibility

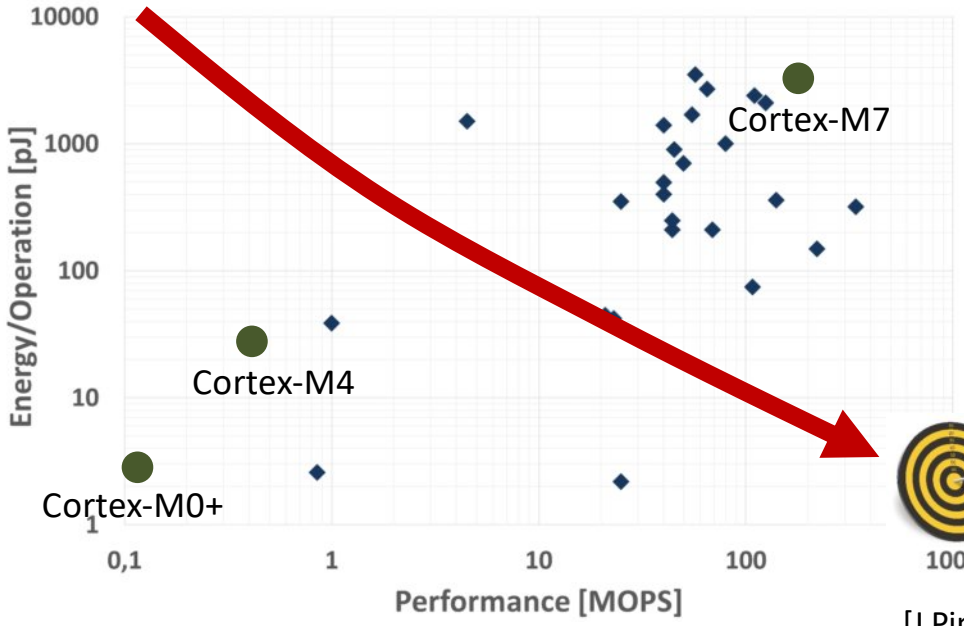
The Edge Performance-Power Proportionality Goal

High-Performance MCU

High-end MCUs like Cortex-M7/55/85

Low-Power MCU

Low-end MCUs like Cortex-M3/4

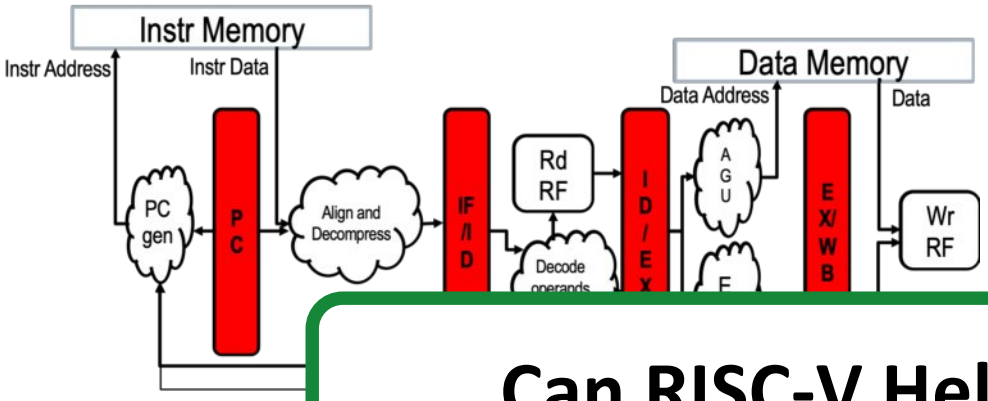


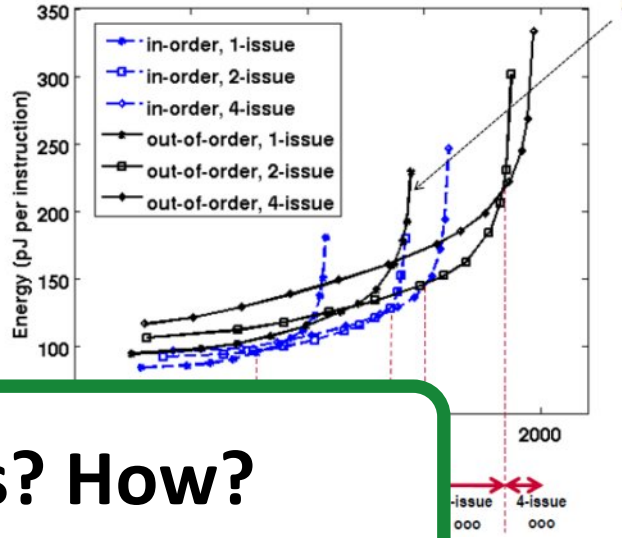
How do we get there?

[J Pineda, NXP + Updates]

1pJ/OP @ GOPS is the challenge

More Performance at Core-Level?





[AziziISCA10]

Can RISC-V Helps Us? How?

- **"Classical"**
  - Faster CLK → deeper pipeline → **IPC drops**
  - Recover IPC → superscalar → **ILP bottleneck** (dependencies)
  - Mitigate ILP bottlenecks → OOO → **huge power, area cost**!
- ..but this does not help with edge ML workload (highly parallel and resilient to low-precision arith)
  - → work on more semantically dense instructions for **better power-performance** proportionality

RISC-V

What is an Instruction Set Architecture?

Interface between HW and SW
List of supported instructions that your CPU must support

```
void matrixAdd(int32_t *  A, int32_t *  B, int32_t *  C, int N, int M)
{
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            C[i*N+j] = A[i*WIDTH+j] + B[i*WIDTH+j];
        }
    }
}
```

Compiler

```
start_loop:
    lw  a0,0(a5)
    lw  t1,0(a7)
    addi    a5,a5,4
    addi    a7,a7,4
    add a0,a0,t1
    sw  a0,0(a6)
    addi    a6,a6,4
bne a5,t3,start_loop
```

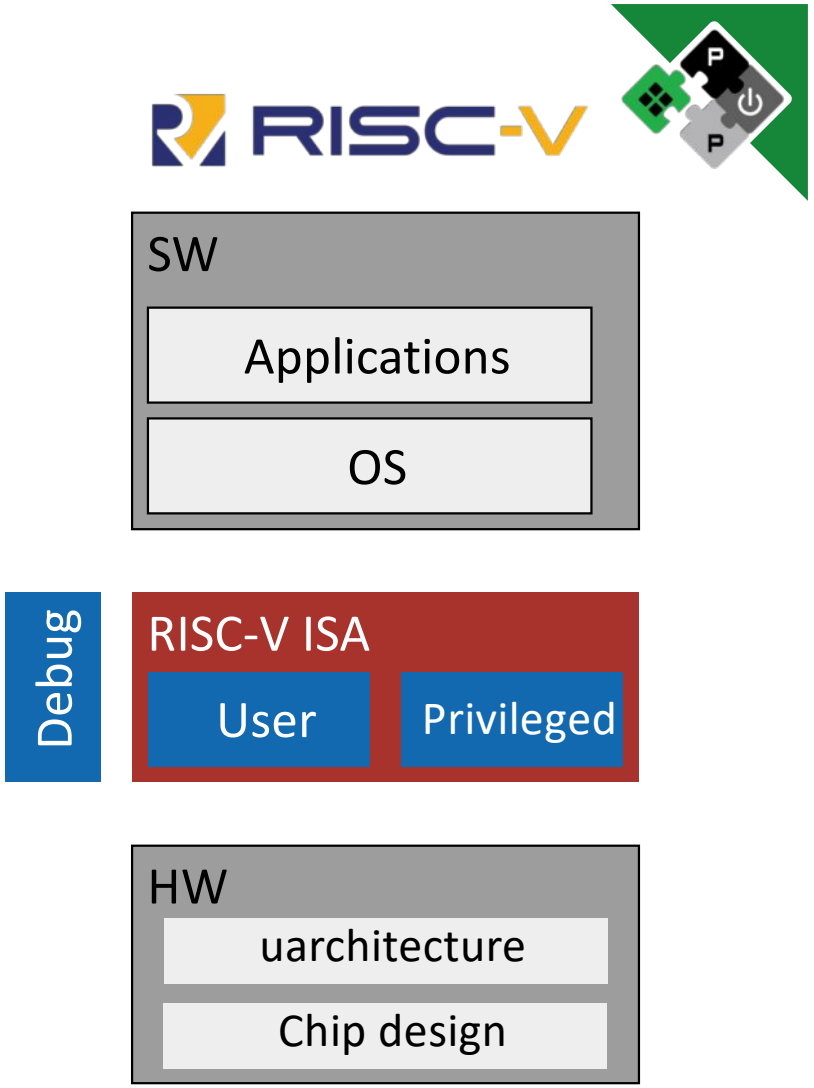
Processor Design (HW)

```
@18c0 ->    0007a503
@18c4 ->    0008a303
@18c8 ->    00478793
@18cc ->    00488893
@18d0 ->    00650533
@18d4 ->    00a82023
@18d8 ->    00480813
@18dc ->    ffc792e3
```

In principle, any compiled code for a given ISA runs on any processor that implements that ISA

RISC-V Instruction Set Architecture

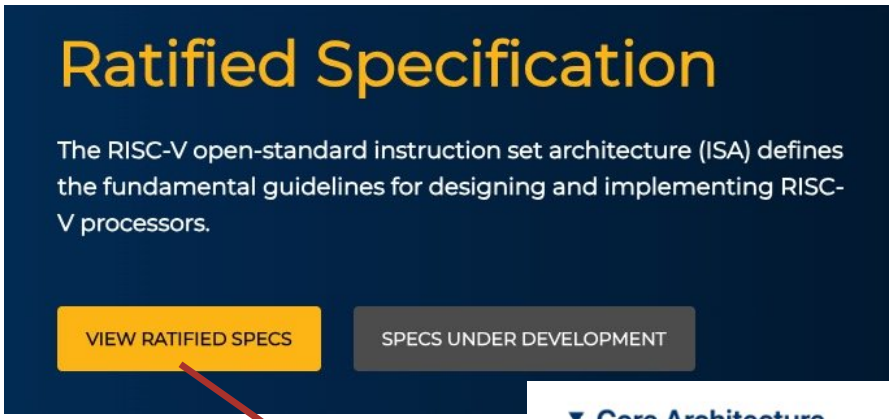
- **Started by UC-Berkeley in 2010**
- **Contract between SW and HW**
  - Partitioned into user and privileged spec
  - External Debug
- **Standard governed by RISC-V foundation**
  - ETHZ is a founding member of the foundation
  - Necessary for the continuity
- **Defines 32, 64 and 128 bit ISA**
  - **No implementation, just the ISA**
  - Different implementations (both open and close source)
- **Workshops and conferences**
  - Very active community, especially in EU

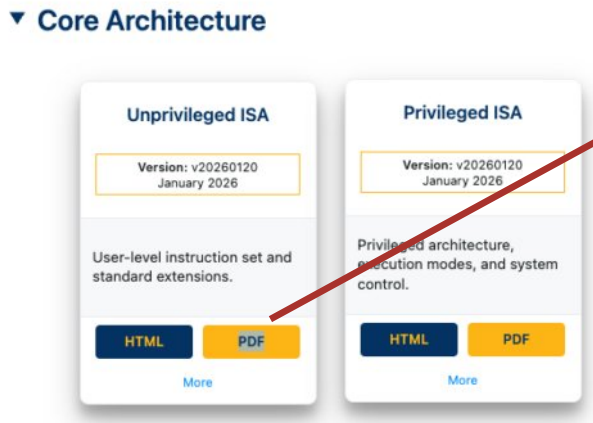


RISC-V Foundation Members

Maintainance of a (Very Important) Document

URL: riscv.org/specifications





In the PDF almost every choice is justified! So you can also get a lot of architectural insights/knowledge by reading it

RISC-V Ecosystem is Growing

- **Binutils – upstream**
- **GCC – upstream**
- **LLVM – upstream**
- **Simulator:**
  - ”Spike” - reference
  - QEMU, Gem5
- **OpenOCD**
  - Bridges HW and SW debuggers (e.g., JTAG and GDB)
- **OS**
  - Linux, sel4, freeRTOS, zephyr, rtems
- **Runtimes**
  - Jikes, Ocaml, Go
- **SW maintained by different parties**
  - Binutils and GCC by Sifive a Berkeley start-up

What's Nice About RISC-V?

- **Open-Source (and free)**
  - Everyone can implement and use its ISA (e.g. to design RISC-V processors)
- **Simple and Modern Design**
  - Far smaller than other commercial ISAs
  - No need to be compatible with older design of same ISA (like ARM, Intel, .. which are around from 20+ years)
- **Clean-state design**
  - Clear separation between user and privileged ISA
- **A modular ISA**
  - Small standard base ISA + Multiple standard extensions
- **Designed for extensibility/specialization**
  - Vast opcode space available for instruction-set extensions

The Freedom in RISC-V is in the Implementation

- **You can access <u>all ISAs</u> without (many) restrictions**
  - SW tools need to be developed so that they can generate code for that ISA
- **Most ISAs are closed. Only specific vendors can implement it**
  - To use a core that implements an ISA, you have to license/buy it from vendor
  - Open source SW (for the ISA) is possible but **building HW is not allowed**

RISC-V

Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP

ARM

C2.9 ADD

Add without Carry.

Syntax

```
ADD{S}{cond} {Rd}, Rn, Operand2
ADD{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only
```

Are RISC-V Processors Better than XYZ?

- **Actual performance depends on the implementation**
  - RISC-V does not specify implementation details (on purpose)
- **RISC-V is a modern design → should deliver comparable performance**
  - If implemented well, it should perform as well as other modern ISA implementations
  - In our experiments, we see **no major weaknesses** when compared to other ISAs
  - It also is **not magically 2x better**
- **High-end processor performance is not only about ISA**
  - Implementation "details" like microarchitecture, memory hierarchy, target technology, power management are more important.

What is Missing about RISC-V?

- **Still in development, but fast progress**
  - Some “advanced” extensions, including **Matrix Extensions**, still being refined, adjusted
  - Tools and development environment needs to catch up.

- **No canonical implementation (“*the*” RISC-V core)**
  - It is free to implement, so many people did so, resulting in many cores

- **Higher end (out of order, superscalar) cores not yet mature**
  - Some RISC-V based Linux laptops available (experimental), but not wide-spread.
  - New RISC-V profile (RVA23) boosted the development of new HPC processors to be used in datacenters (mostly from Asia and US, Europe is catching up)

RISC-V is a Load/Store Architecture

- **All operations are on internal registers**
  - Can not manipulate data in memory directly
- **Load instructions to copy from memory to registers**
- **R-type or I-type (compute) instructions to operate on them**
- **Store instructions to copy from registers back to memory**
- **Branch and Jump instructions**
- **1/3 ALU utilization if operands are from/to memory (LD, ALU, ST)**

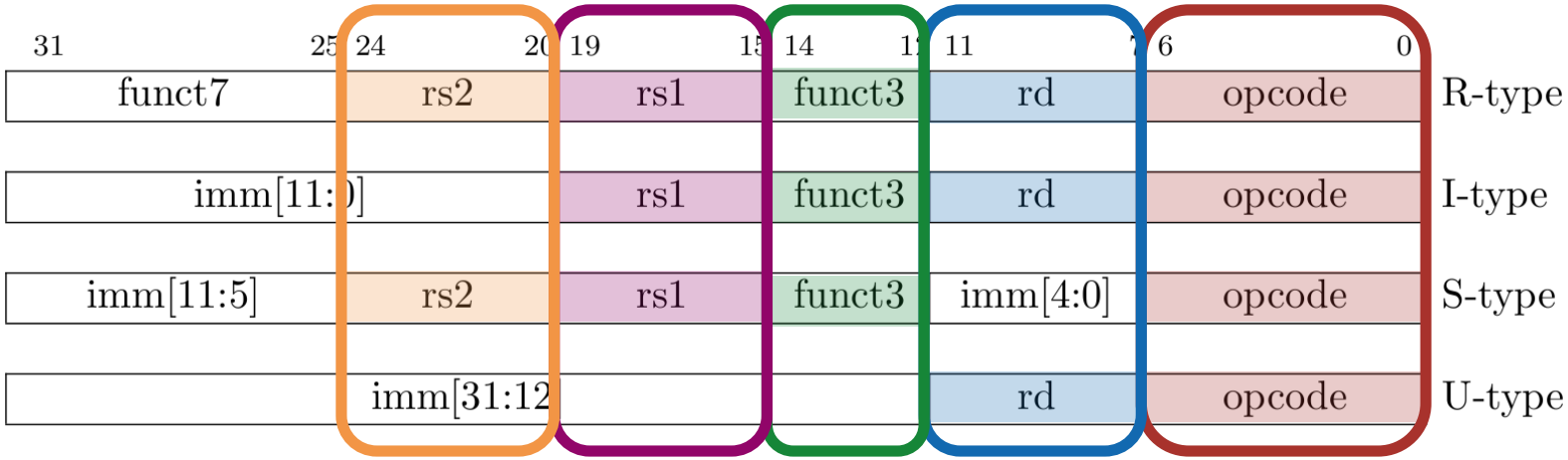
RISC-V Architectural State

- **There are 32 regs, 32/64/128 bits long**
  - Named x0 to x31
  - x0 is hard wired to zero
  - There is a standard '**E**' extension that uses only 16 registers (RV32E)
- **In addition one program counter (PC)**
  - Byte based addressing, program counter increments by 4/8/16
- **For FP operation 32 additional FP regs**
  - "Zfinx" extension to use "integer" ones
- **Additional Control Status Registers (CSRs)**
  - Encoding for up to 4'096 registers are reserved. Not all are used.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

RISC-V Instructions: Four Basic Types

- **R** register to register operations
- **I** operations with immediate/constant values
- **S / SB** operations with two source registers
- **U / UJ** operations with large immediate/constant value

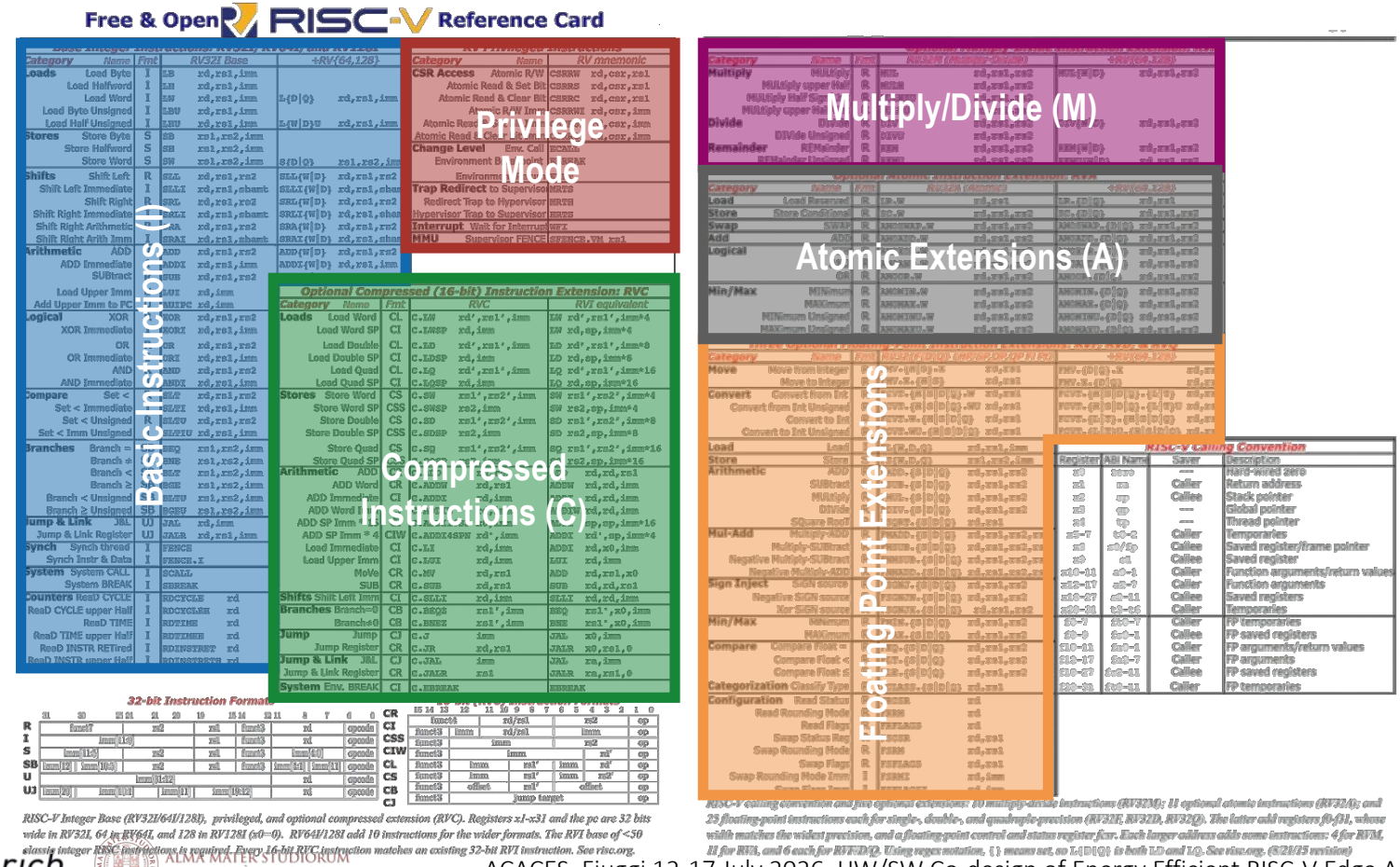


RISC-V Baseline and Extensions

- **User-level ISA Spec**
  - Defines the normal insns for computation
  - Separated into extensions, only **I** is mandatory
- **Kept very simple and extendable**
  - Wide range of applications from IoT to HPC
- **RV + word-width + extensions**
  - RV32IMC: 32bit, integer, multiplication, compressed
  - RV64GC: 64bit, G=I+M+A+F+D+csr+fence and compressed
  - RV32IMCXpulp: RV32IMC + custom extensions “pulp”
- **Privileged Specification:**
  - Governs OS functionality: Exceptions, Interrupts
  - Virtual Addressing, Privilege Levels

I	Integer instructions (frozen)
E	Reduced number of registers
M	Multiplication and Division
A	Atomic instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
C	Compressed Instructions (frozen)
B	Bit manipulation instructions
V	**Vector instructions (additional state!)**
X	**Non Standard Extensions**

Reduced Instruction Set: All in One Page



Work Continues on New RISC-V Extensions

- **Foundation members work in task-groups**
- **Dedicated task-groups (TGs)**
  - Formal specification
  - Memory Model
  - Marketing
  - External Debug Specification
- **Dedicated Special Interests Groups (SiGs)**
  - Space
  - Security
  - Others..
  - To tackle specific challenges that can be addressed at architectural level

Q	Quad-precision Floating-Point
L	Decimal Floating Point
T	Transactional Memory
P	Packed SIMD
J	Dynamically Translated Languages
N	User-Level Interrupts
	Matrix Extensions (IME, VME, AME)

Extending ISA for ML → RISC-V as an enabler

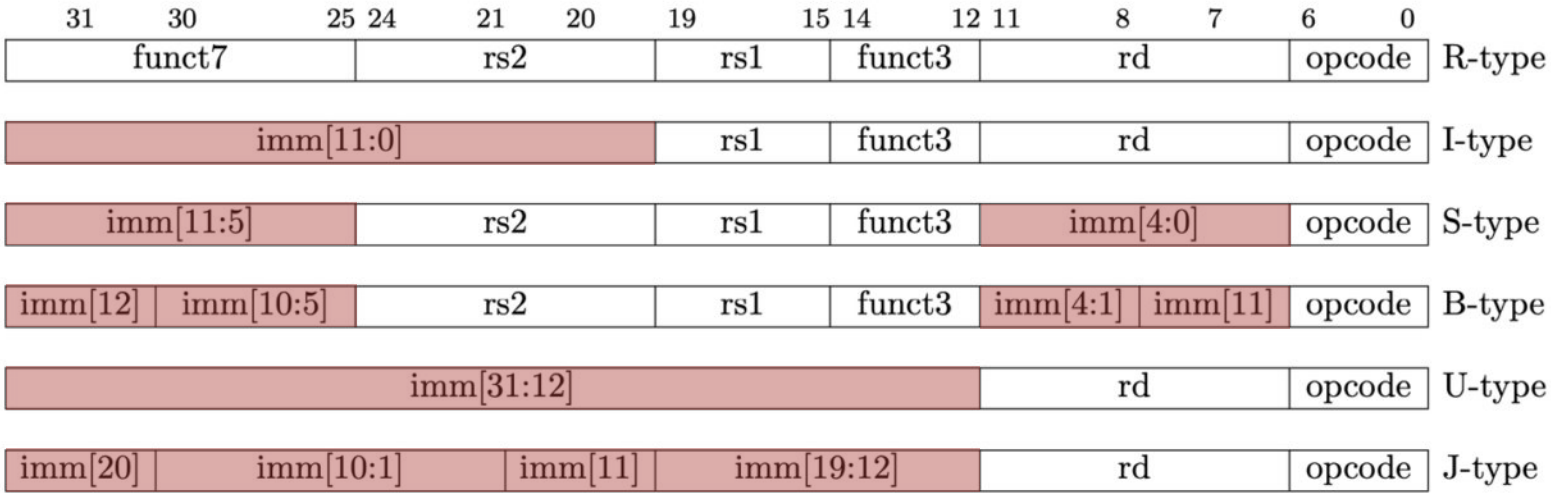
- Main opcodes are 7-b (actually 5-b due to Compressed C insns)
- **Reserved** opcodes for standard extensions
- Rest of opcodes free for **custom** implementations
- **Standard extensions will be frozen/not change in the future**

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Extensibility is integral to RISC-V ISA design!

Constants (Immediates) in Instructions

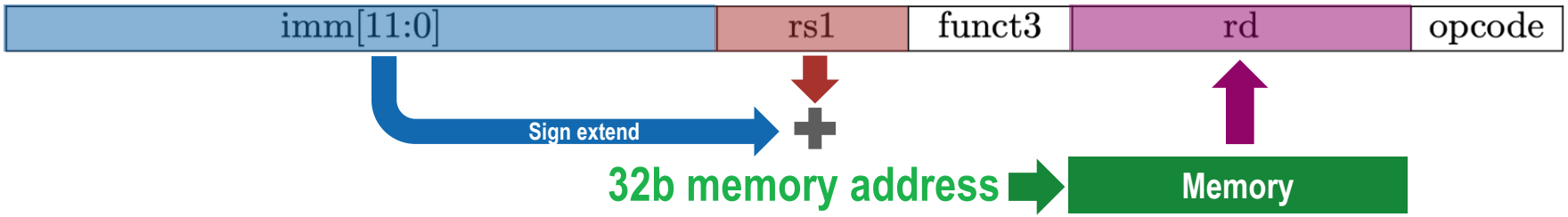
- **In 32-bit instructions, not possible to have 32-bit constants**
  - Constants are distributed in instructions, and then sign extended
  - The **L**oad **U**pper **I**mmediate (`lui`) instruction to assemble/push constants
- **Instruction types according to Immediate encoding**



Load From Memory (ld): How Immediates Work

- **Not possible to fit a 32b address in 32b encoding directly**
  - Take the content in source (**rs1**), add the immediate (**imm**) to it. This is the **address**
  - Read from this **address** in the memory and load into the destination (**rd**) register
- **RISC-V tries to minimize number of instructions**
  - The `ld` instruction seems overly complicated, but you can use this for everything

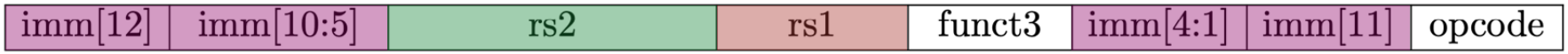
```
ld x9, 64(x22)
```



Branching: How Addresses Come Togheter

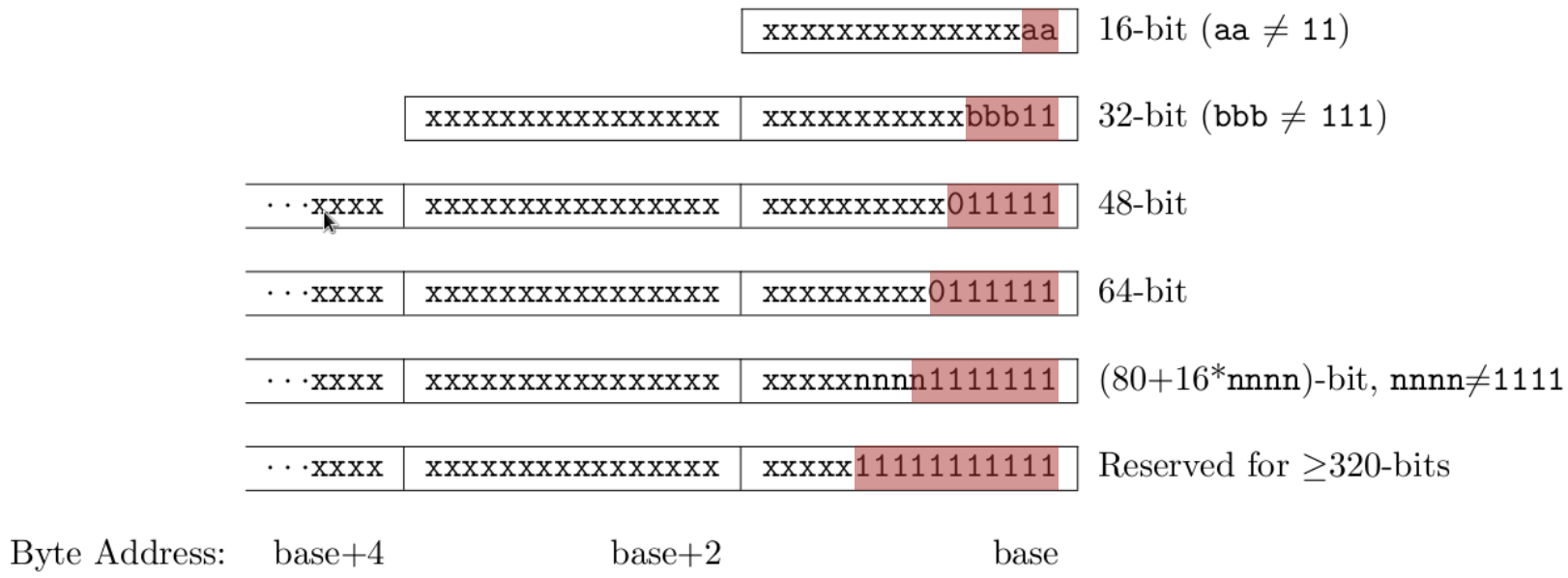
- **Similar problem, how to encode jump address in branches**
  - Branch on Equal (**beq**) and Branch on Not Equal (**bne**)
  - They use B type operations, need two source registers
- **Jumps are relative to Program Counter (PC)**
  - The immediate (constant) shows how far we have to jump (PC-relative addressing)
  - Works addresses within ± 4096. To branch further, we need several instructions.

```
bne x10, x11, 2000 // if x10 != x11, jump 2000 ahead
```



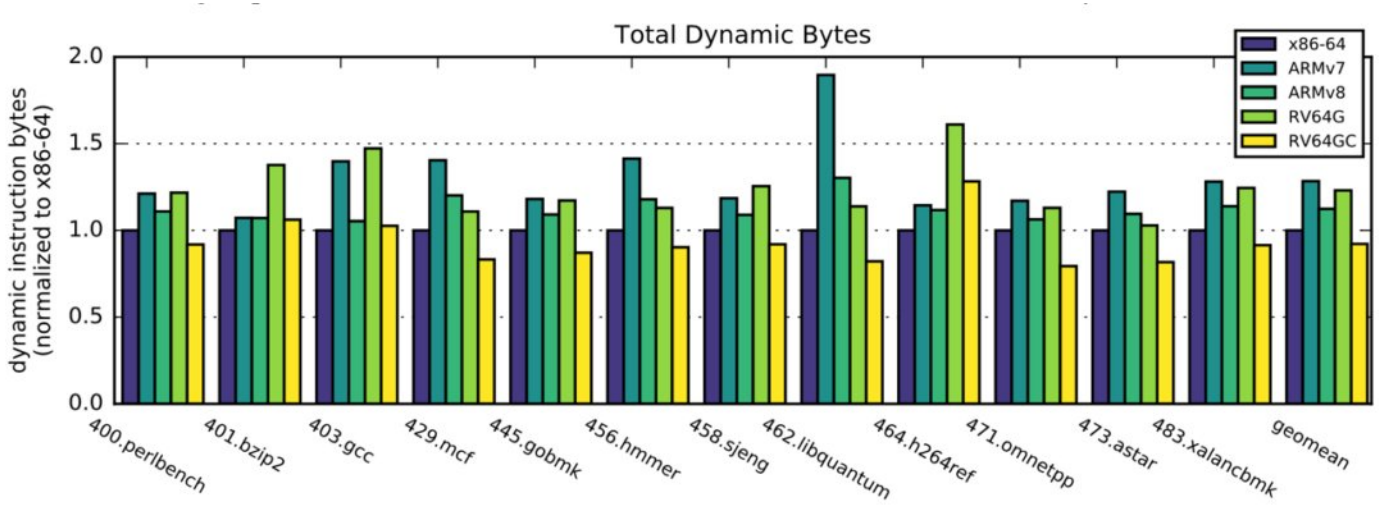
RISC-V Instruction Length is Encoded

- **Supports instructions of 16, 32, 48, 64, 80, 96, … , 320 bit**
  - Allows RISC-V to have Compressed instructions
- **LSB of the instruction tells how long the instruction is**



Compressed Instructions ‘C’

- **Use 16-bit instructions for common (and simple) operations**
  - Performance of operations won’t change but..
  - Code size reduction by 34%
  - Compressed instructions increase fetch-bandwidth



x86-64: 3.71 bytes / instruction **RV64IC**: 3.00 bytes / instruction

So..How To Build RISC-V Cores?

- **RISC-V ISA tells you the function**
  - You know which instructions are supported/want to support
  - How they are encoded
  - What they are supposed to do
- **It does not tell you any implementation details**
  - Pipeline stages, memory hierarchy, computation units, in-order or out–of order
  - Everyone is free to figure out how to best implement these
- **Need to come up with a micro-architecture to implement it**
  - Determine which standard extensions are supported, how
    - RV32I? (small) – RV32IMCFX..? (bigger but more powerful) – RV64GC (Linux!)
  - Choose a micro-architecture that fits performance requirements
    - Which metrics to take into the account for the design?

Performance Metrics

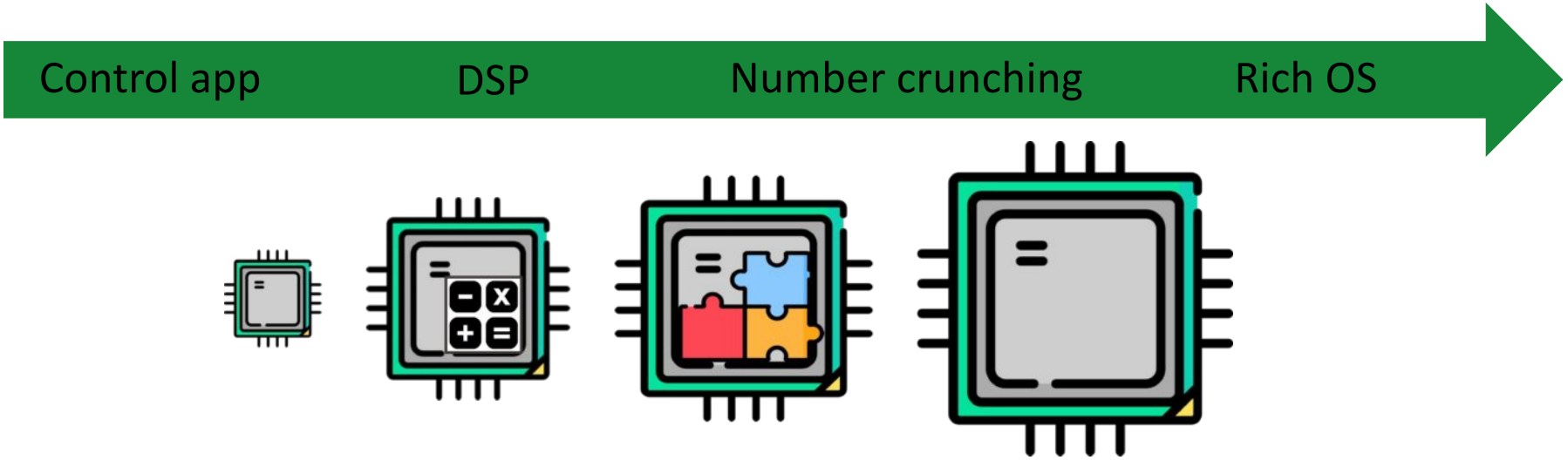
- **Area**
  - in kGE equivalent (# of simple logic gates) or mm^2 (technology dependent)
- **Frequency:**
  - Depends on # of gates on longest path
- **Power:**
  - Strongly depends on the above metrics
  - **Leakage**: dissipated even when not working (Area)
  - **Dynamic Power**: dissipated on logic transitions (frequency and area)

$$P = ACV^2f + \tau AVI_{short}f + VI_{leak}$$

Dynamic Short-Circuit **Leakage**

- **CPU Design**:
  - IPC (Instructions per cycle)
    - IPC implicitly measured in commonly used benchmarks (Coremark, Dhrystone, SpecInt)
  - Energy Efficiency: OPs/Joule
- **Hardware Designer**
  - Tries to find a good balance
  - Application dependent
    - IoT and HPC have different requirements
  - **One size does not fit all**

Different Workloads, Different Cores



- Simple ISA (RV32I)
- Area/power are primary design goals

- Richer ISAs (Xcustom)
- Energy-efficiency is the primary design goal

- Very Rich ISA (at least **G**)
- Optimize micro-archi to increase IPC on OS kernel execution

RISC-V Cores Designed in Our Group

32 bit			64 bit
Low Cost Core	**DSP Enhanced Core**	**Streaming Compute Core**	**Linux capable Core**
▪ **Zero-riscy** ▪ RV32-ICM ▪ **Micro-riscy** ▪ RV32-CE	▪ **RI5CY** ▪ RV32-ICMFX ▪ SIMD ▪ HW loops ▪ Bit manipulation ▪ Fixed point	▪ **Snitch** ▪ RV32-ICMDFX	▪ **Ariane** ▪ RV64GCH ▪ Full privileged spec ▪ Hypervisor Extension

Ibex by LowRISC

CV32E40P by OpenHW

CV6A by OpenHW

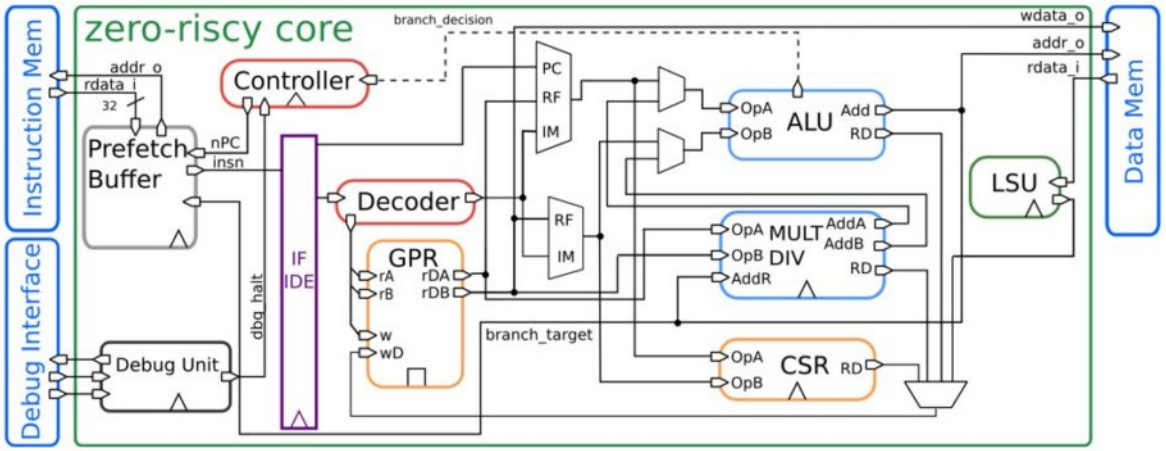
lowRISC

OPENHW GROUP

OPENHW GROUP

Zero-riscy → Ibex

- **2-stage pipeline**
- **Optimized for area**
  - Area:
    19 kGE (Zero-riscy)
    12 kGE (Micro-riscy)
  - Critical path:
    ~ 30 logic levels
- **New name: Ibex**
  - LowRISC has taken over Zero/Micro-Riscy in 2019

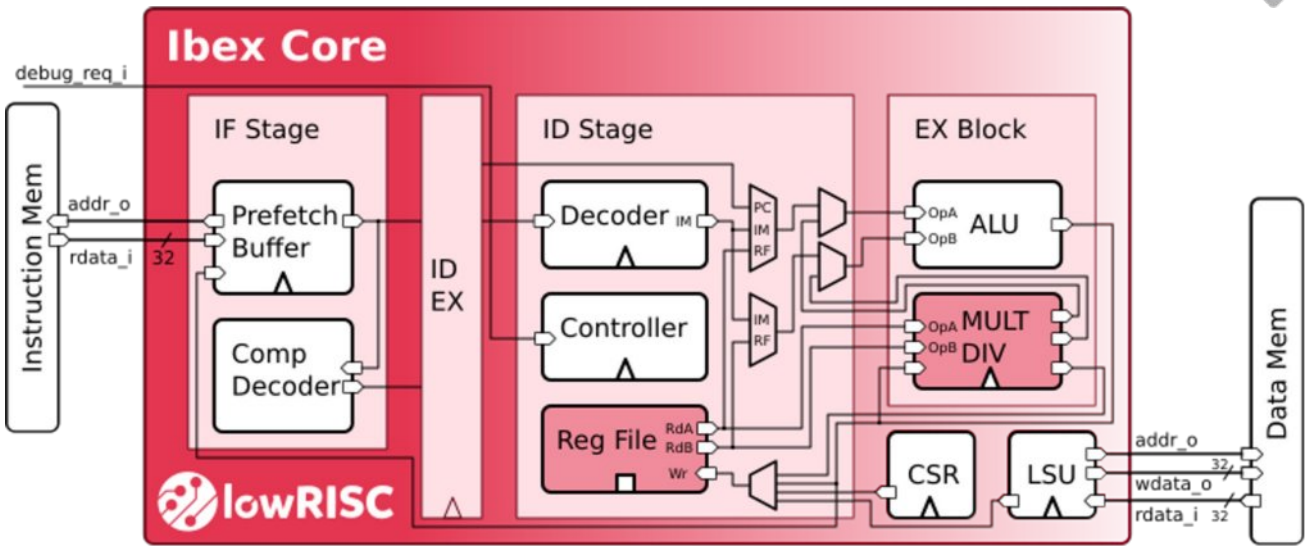


- **Two Configurations:**
  - **Zero-riscy**: RV32IMC (2,44 Coremark/MHz)
    - 32 registers, hardware multiplier
  - **Micro-riscy** : RV32EC (0,91 Coremark/MHz)
    - 16 registers (E), software emulated multiplier

P. Davide Schiavone *et al.*, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1-8.

Ibex continues to grow with LowRISC

40+	Contributors
680	Pull Requests
314	GitHub Issues



- *Ibex is a **small** and efficient, **32-bit**, in-order RISC-V core with a **2-stage** (or optionally 3-stage) pipeline that implements the **RV32IMCB** instruction set architecture.*
- *Main processor of the **OpenTitan HW Root of Trust** → https://opentitan.org/*

Growth of Ibex measured with Coremark/MHz

Config	"micro"	"small"	"maxperf"	"maxperf-pmp-bmfull"
Features	RV32EC	RV32IMC, 3 cycle mult	RV32IMC, 1 cycle mult, Branch target ALU, Writeback stage	RV32IMCB, 1 cycle mult, Branch target ALU, Writeback stage, 16 PMP regions
Performance (CoreMark/MHz)	0.904	2.47	3.13	3.13
Area - Yosys (kGE)	16.85	26.60	32.48	66.02
Area - Commercial (estimated kGE)	~15	~24	~30	~61
Verification status	Red	Green	Green	Green

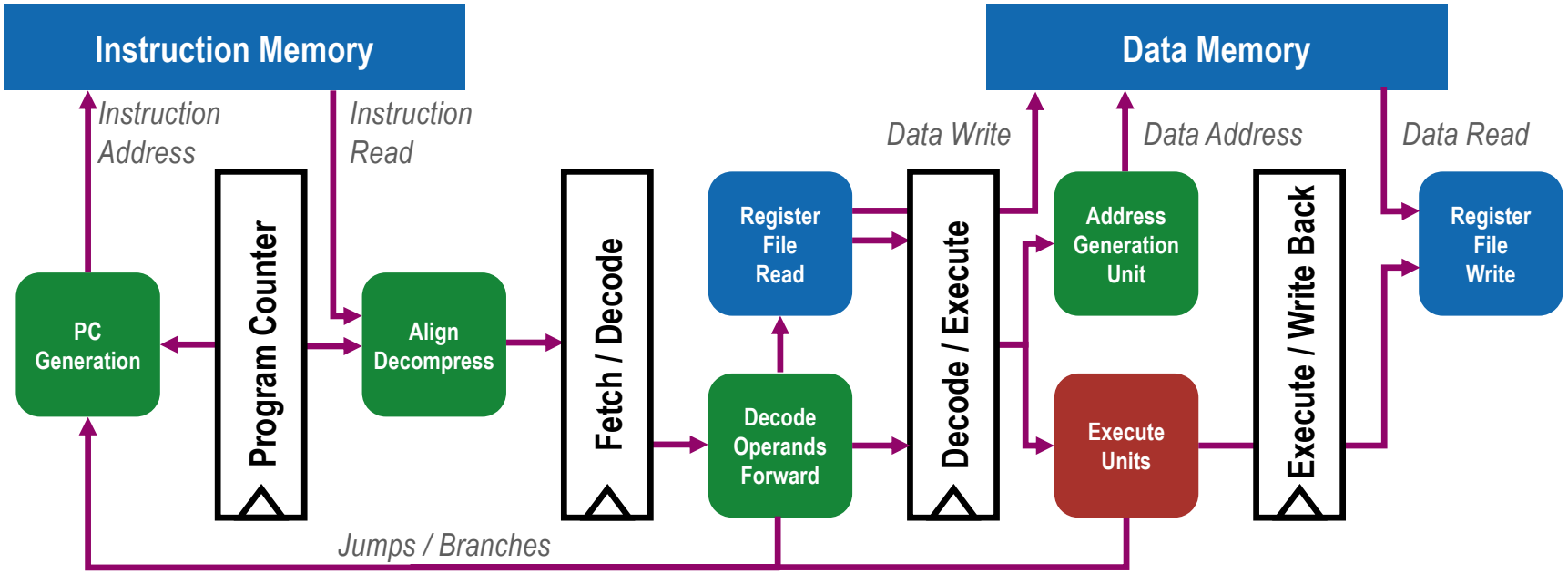
https://github.com/lowrisc/ibex

lowRISC

RI5CY/CV32E40P: Our Main 32b RISC-V Core for DSP/AI

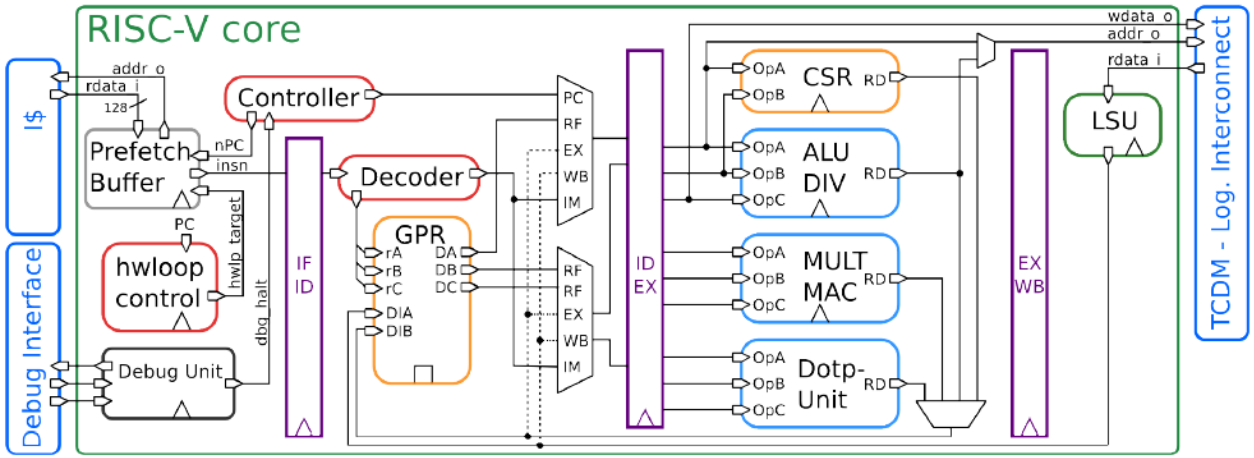
- **Zero-riscy / Ibex is suitable for simple applications**
  - Control applications, book-keeping
- **For number crunching, we need more capable cores**
  - Mainly used in clusters for signal processing / machine learning applications
- **Tuned for energy efficiency**
  - Not necessarily lowest power
- **Make use of *custom extensions***
  - The Xpulp extensions enhance the computing capabilities on arithmetic operations
  - Several Xpulp extensions in discussions for ratification
    - **B** (bit manipulation) ratified → based/inspired by instructions in Xpulp
    - **P** (packed SIMD) towards ratification → based on DSP instructions in Xpulp

Simplified Pipeline for RI5CY/CV32E40P



RI5CY → CV32E4

- **4-stage pipeline**
  - >40 kGE
    - **~2x bigger than Ibex**
  - Coremark/MHz 3.19
- **Includes Xpulp extensions**
  - SIMD
  - Fixed point
  - Bit manipulations
  - HW loops



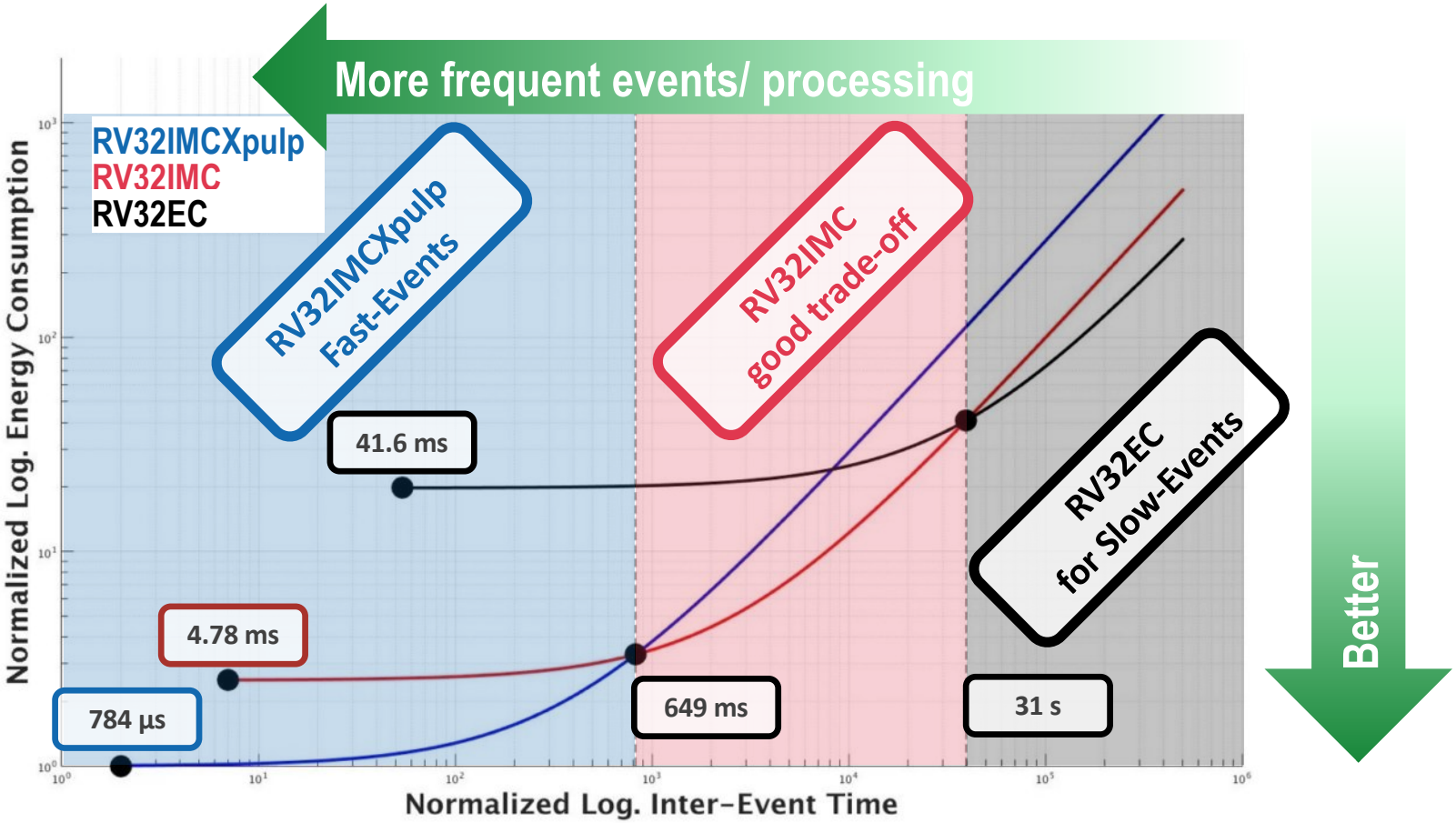
- **Different Options:**
  - **FPU**: IEEE 754 single precision
    - Including hardware support for `FDIV`, `FSQRT`, `FMAC`, `FMUL`
  - **Privilege support**:
    - Supports privilege mode **M** and **U**

M. Gautschi *et al.*, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700-2713, Oct. 2017.

RISC-V has Space for Custom Instructions (X)

- **There is a reserved decoding space for custom instructions**
  - Allows **everyone** to add new instructions to the core
  - The address decoding space is **reserved**, it will not be used by future extensions
  - Implementations supporting custom instructions will be compatible with standard ISA
    - Code compiled for standard RISC-V will run without issues
  - The user has to provide support to take advantage of the additional instructions
    - Compiler that generates code for the custom instructions
- **We use a lot this degree of freedom**
  - Great tool for exploring
  - The goal is to help ratify interesting extensions as standards through working groups

Energy Efficiency 2D Convolution @55MHz, 0.8V



Summary

- **RISC-V specifies the ISA, not the implementation**
  - Many composable extensions
  - Customization of the ISA possible through X extensions (powerful exploration tool)
  - Processor performance depends on the implementation (left to HW designers)
- **RISC-V can be a key enabler to address the edge AI challenges**
  - Flexible, yet energy-efficient (and domain-specialized), TinyML and Edge processors
- **RISC-V Edge Processors**
  - Zero-riscy suitable for tiny control applications
  - RI5CY suitable for DSP/AI through custom ISA extensions (Xpulp)
- **Next**
  - Methodology to design ISA extensions (and how to use them in SW)
  - Xpulp and Xpulpnn use-cases

Thank you!

ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

HW/SW Co-design of Energy Efficient RISC-V Edge AI Platforms

ACACES 2026, Fiuggi (Italy), 12-17 July 2026

Part 2: Domain-Specialized RISC-V Processors for DSP and AI

Angelo Garofalo angelo.garofalo@unibo.it
agarofalo@iis.ee.ethz.ch

pulp-platform.org
@pulp_platform
company/pulp-platform
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

What is this Course About?

- Part1: The RISC-V ISA and Edge Processors
  - Introduction to RISC-V and processors' implementation
- **Part 2: Domain-Specialized RISC-V Processors for DSP and AI**
  - **Methodology of ISA extensions; Examples on DSP and AI use-cases**
- Part 3: Parallel Ultra Low Power Cluster and Vector Lockstep Execution Modes
  - Parallel Ultra Low-Power Cluster: Architectures, Optimizations and Limitations
- Part 4: Addressing the Von Neumann Bottleneck with Heterogeneous Clusters and Streaming Clusters
  - Extending the PULP cluster with accelerators: digital & analog
- Part 5: RISC-V Vector Processors: Advantages and Drawbacks for Edge AI
  - RISC-V Vector ISA and energy-efficient micro-architectural implementations

Outline Part 2

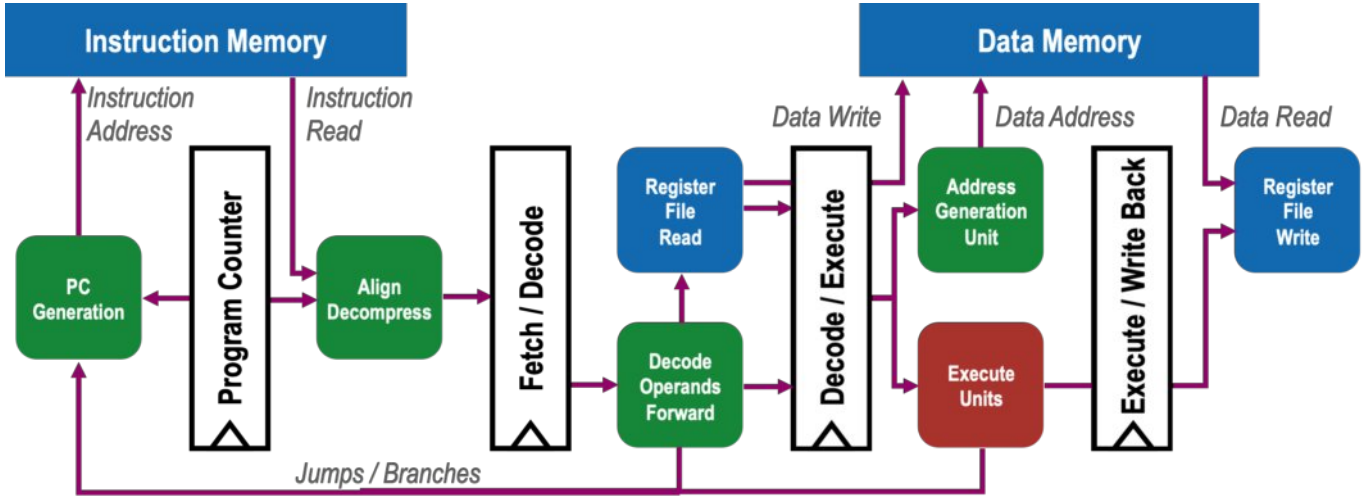
- Why do we expect higher energy efficiency through domain-specialization of RISC-V processors?
- RISC-V ISA extensions for DSP/AI (Xpulp): step-by-step methodology
- How to exploit specialized ISA in SW: The convolution case study
- Improving operation efficiency of AI kernels through XpulpNN ISA and mixed-precision support
- Analysis of benefits and costs of ISA-extended RISC-V processors
- Summary

Instruction Specialization & Energy Efficiency

- Battery has a given energy budget **(mAh @ V)**
- Reducing Energy consumption → longer device lifetime
- We care about **latency** & **Energy:** $\boldsymbol{E = P_{avg} \cdot latency}$
- More semantically dense ISA instructions helps reduce instruction count to execute a workload
- Direct advantages on latency and energy reductions if:
  - Timing of the processor (max op. freq.) not affected
  - Limited Power overhead to support new instructions

A Typical Processing Core for TinyML

- 4-stage RISC-V **RV32IMC**
- 32-bit wide datapath
- 32 32-bit regs in the RF



Single issue, in order is the most energy efficient

- Memory is always critical: 1 full stage for fetch, 2 full stages for load+writeback
- Single stage decode
- Singles stage execute/WB

Is RV32IM Processor Enough for AI/DSP?

FIR Inner Loop Exec. in RV32IM core → this is part of a larger piece of code

```
for(int j=0; j<coeff_len; j++) {
  sum += arr[i+j] * coeff[j];
}
```

ANSI C

Lower-level view

```
for(int j=0; j<coeff_len; j++) {
  int8 arr_int   = arr[i+j];
  int8 coeff_int = coeff[j];
  arr_int = arr_int * coeff_int;
  sum = sum + arr_int;
}
```

ANSI C

Assembly code on RV32IM Architecture

```
Lstart:
lb t2, 0x0(t1)
lb a4, 0x0(t3)
mul a4, a4, t2
add t0, t0, a4
addi t3, t3, 0x1
addi t1, t1, 0x1
bne t3, a7, Lstart
```

RV32IM ASM

per iteration:
1 MAC (multiply-accumulate) operation
7 instructions
Compute efficiency = 1/7 = 0.14 ☹

We want to minimize instructions that are not DOTP/MAC in the innermost loop of an arithmetic-intensive kernel

Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
  1. **Fuse Multiplication and Addition in a single instruction/operation → Multiply-and-Accumulate (MAC) operation**
  2. Exploit regular access patterns of the workload → Load/Store with Post-increment of the pointer
  3. Boost Loop Control in HW to reduce branch penalties → HW Loops

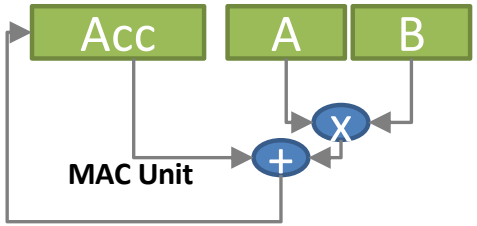
- **DSP**
  4. Exploit Low-bitwidth of DSP/AI operands and data → Packed-SIMD ALU operations & dot-products

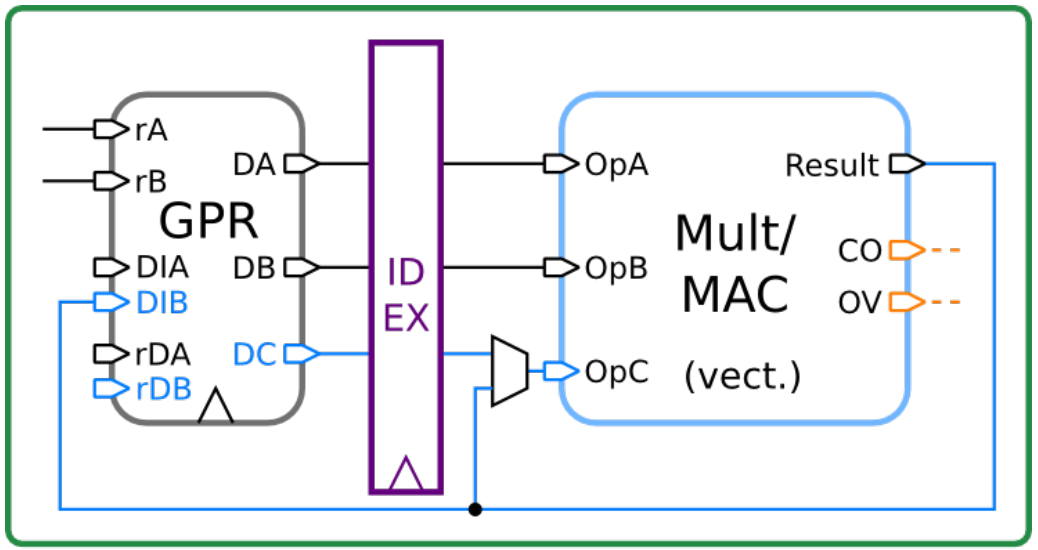
Multiply Accumulate

- **Integrate the MAC unit with the register file**
- **Accumulator becomes also a source operand of the insn**
- **p.mac rD, rs1, rs2**

Micro-archi requirements

- **Ex unit performs MUL and ADD in the same clock cycle**
  - Multiplier followed by an adder
- **rD encoded to be also an input operand**
  - Modifications to the Decoder
  - Additional read port to the GPR
- **Pro:**
  - Faster access to mac accumulation
  - Single cycle mult/mac
- **Cons:**
  - Additional read port on the register file
  - used for pre/post increment with register





Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
  1. Fuse Multiplication and Addition in a single instruction/operation → Multiply-and-Accumulate (MAC) operation
  2. **Exploit regular access patterns of the workload → Load/Store with Post-increment of the pointer**
  3. Boost Loop Control in HW to reduce branch penalties → HW Loops

- **DSP**
  4. Exploit Low-bitwidth of DSP/AI operands and data → Packed-SIMD ALU operations & dot-products

Post Increment Load/Store

- Standard RISC-V requires explicit updates of memory pointers (**addi**)
- Solution: Automatic address update
  - Perform memory access with e.g., **x10**
  - In the same cycle, update **x10** with the updated pointer for next mem access
    - **x10 ← (x10 + 0x1)**
- **Micro-architectural requirements**
  - LSU for the memory access (already there)
  - ALU must be active for address update
  - additional write port to RF
- **Costs**
  - 12% area overhead w.r.t. RV32IMC

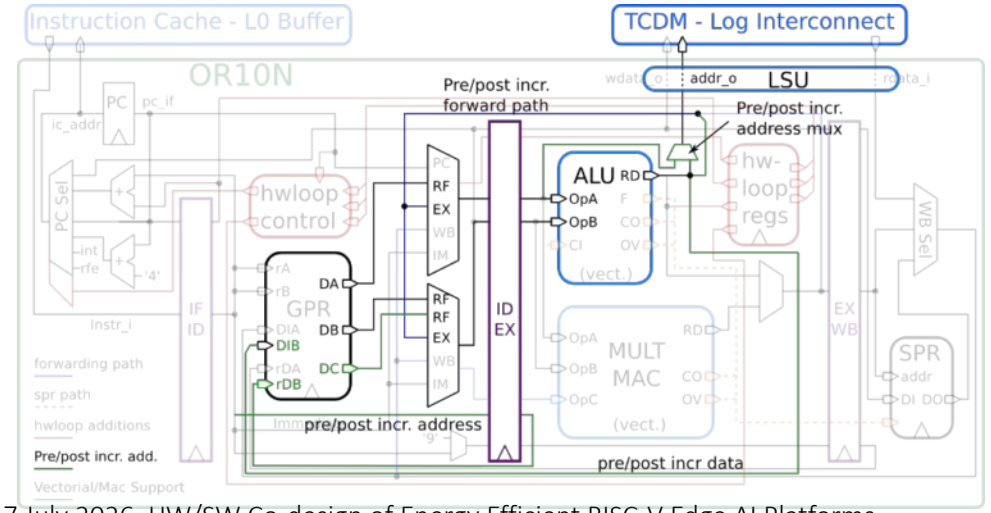
Original RISCV

```
lb x2, 0(x10)
lb x3, 0(x12)
addi x10, x10, 1
addi x12, x12, 1
```

With auto-increment

```
p.lb x2, 0(x10!)
p.lb x3, 0(x12!)
```

You save two instructions!



FIR Inner Loop Exec. in RV32IM + MAC + pointer post-incr.

```
for(int j=0; j<coeff_len; j++) {
  sum += arr[i+j] * coeff[j];
}
```

ANSI C

Lower-level view

```
for(int j=0; j<coeff_len; j++) {
  int8 arr_int   = arr[i+j];
  int8 coeff_int = coeff[j];
  arr_int = arr_int * coeff_int;
  sum = sum + arr_int;
}
```

```
a7 = coeff_len
```

```
Lstart:
p.lb t2, 0x0(t1!)
p.lb a4, 0x0(t3!)
p.mac t0, a4, t2
bne t3, a7, Lstart
```

RV32IMXpulp ASM

Implicit update after load operation:
t1 = t1 + 0x1 (Byte)

per iteration:
1 MAC (multiply-accumulate) operation
4 instructions
efficiency = 1/4 = 0.25 ☹

one instruction is spent to control the loop
→ Can I delegate this operation to Hardware?

Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
  1. Fuse Multiplication and Addition in a single instruction/operation → Multiply-and-Accumulate (MAC) operation
  2. Exploit regular access patterns of the workload → Load/Store with Post-increment of the pointer
  3. **Boost Loop Control in HW to reduce branch penalties → HW Loops**

- **DSP**
  4. Exploit Low-bitwidth of DSP/AI operands and data → Packed-SIMD ALU operations & dot-products

Hardware (Zero-Overhead) Loops

- Loops setup beforehand through three custom instructions (outside hot loop)
  - **Lp.start** → sets a SP reg with start addr. of the loop
  - **Lp.end** → sets a SP reg with end addr. of the loop
  - **Lp.count(lp.counti)** → sets # of iterations
  - → no restrictions on start/end address
- Fast instruction
  - **lp.setup, lp.setupi**
  - → start_addr = PC+4
  - → end_addr = start + offset
  - → counter = IMM or GP-REG

```
c = 0;
for(i=0;i<100;i++)
    c = c + a[i]*b[i];
```

Original RISC-V

```
mv   x4, 100
mv   x5, 0
Lstart:
addi  x4, x4, -1
lw x8, 0(x9)
lw x10, 0(x11)
add x9, x9, 4
add x11, x11, 4
mul x8, x8, x10
add x5, x5, x8
bne  x4, x0, Lstart
```

w/ HW Loops

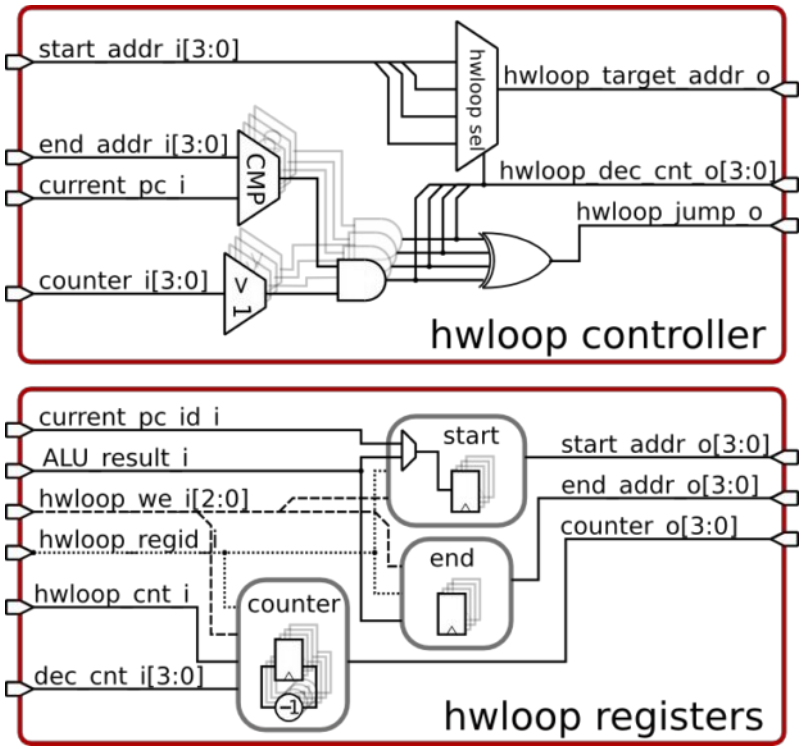
```
mv   x5, 0
lp.setupi 100, Lend
lw x8, 0(x9)
lw x10, 0(x11)
add x9, x9, 4
add x11, x11, 4
mul x8, x8, x10
Lend:  add x5, x5, x8
```

No explicit instruction to ctrl the loop!

Hardware Loop Implementation

Small HW engine automatically handles the loop

- **Hwloop registers: two sets of of three regs**
  - Start, end addresses of the loop + counter of # of iterations
  - Two sets are needed to accelerate nested loops
  - More sets would require higher micro-archi overhead and bring no significant benefits
- **Hwloop controller: logic unit that automatically decrements the counter:**
  - Once # of iterations reached → The controller will generate a "jump" for the processor
- **Area cost**
  - **5% area overhead**
- **Performance**
  - **Speedup can be a factor of 2x!**
  - Small loops benefit more!



FIR Inner Loop Exec. in RV32IM + MAC + pointer post-incr. + HW Loo

```
for(int j=0; j<coeff_len; j++) {
  sum += arr[i+j] * coeff[j];
}
```

ANSI C

Lower-level view

```
for(int j=0; j<coeff_len; j++) {
  int8 arr_int   = arr[i+j];
  int8 coeff_int = coeff[j];
  arr_int = arr_int * coeff_int;
  sum = sum + arr_int;
}
```

a7 = coeff_len

```
lp.setup t3,a7,Lend
Lstart:
p.lb t2, 0x0(t1!)
p.lb a4, 0x0(t3!)
Lend:
p.mac t0, a4, t2
```

RV32IMXpulp ASM

per iteration:
1 MAC (multiply-accumulate) operation
3 instructions
efficiency = 1/3 = 0.33 ☺

one element at time (1 Byte) is loaded and then processed. However, my processor datapath has a 32-bit wide parallelism → How to exploit it?

Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
  1. Fuse Multiplication and Addition in a single instruction/operation → Multiply-and-Accumulate (MAC) operation
  2. Exploit regular access patterns of the workload → Load/Store with Post-increment of the pointer
  3. Boost Loop Control in HW to reduce branch penalties → HW Loops

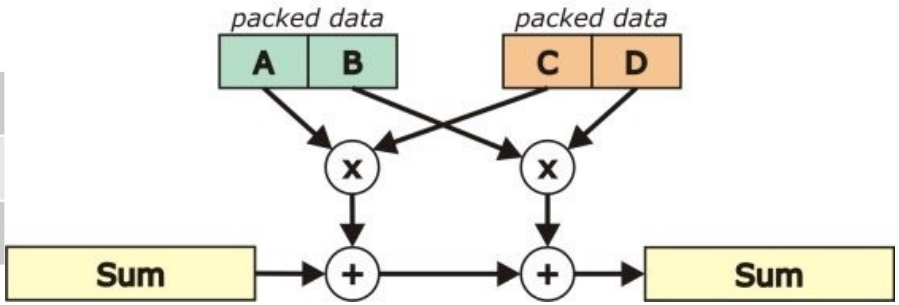
- **DSP**
  4. **Exploit Low-bitwidth of DSP/AI operands and data → Packed-SIMD ALU operations & dot-products**

Packed-SIMD

Remember: DSP/TinyML inference are OK with low-bitwidth operands

- **Packed-SIMD extensions**
  - Well known idea from classic DSP → Widely used (ARM, x86)
  - Efficient usage of hardware resources to target low-bitwidth arithmetic (DSP, QNNs)
  - Target embedded systems → RV Vector ISA more for high performance
- **SIMD in 32bit machines**
  - Vectors are either 4 8bits-elements or 2 16bits-elements

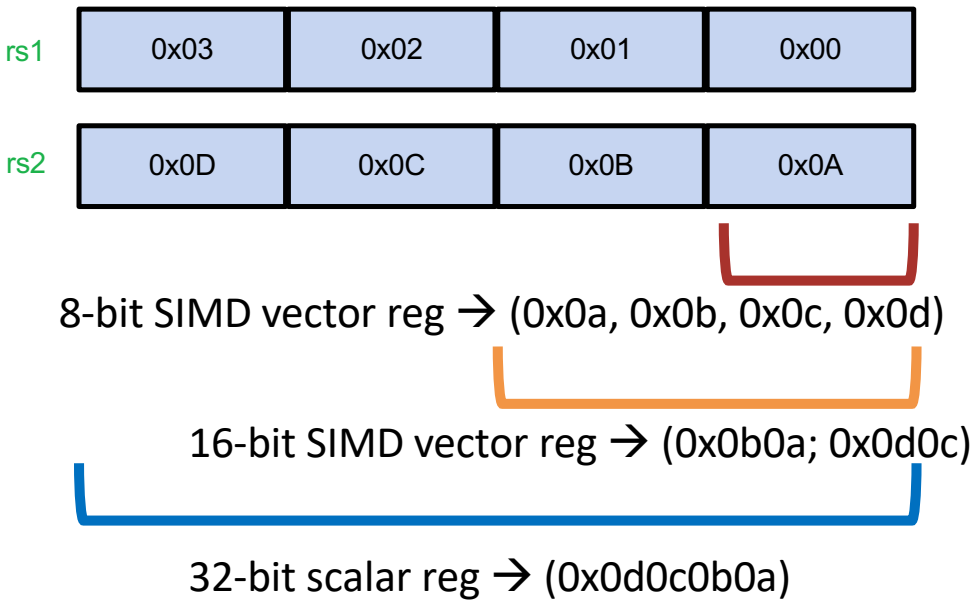
Computation	add, sub, shift, avg, abs, dot product
Compare	min, max, compare
Manipulate	extract, pack, shuffle



Packed-SIMD ALU Instructions

- **Same Register-file**
  - The instruction encodes how to interpret the content of the register

add rD, rs1, rs2	rD = 0x03020100 + 0x0D0C0B0A
add.h rD, rs1, rs2	rD[0] = 0x0100 + 0x0B0A rD[1] = 0x0302 + 0x0D0C
add.b rD, rs1, rs2	rD[0] = 0x00 + 0x0A rD[1] = 0x01 + 0x0B rD[2] = 0x02 + 0x0C rD[3] = 0x03 + 0x0D



Packed-SIMD Vectorial Adder

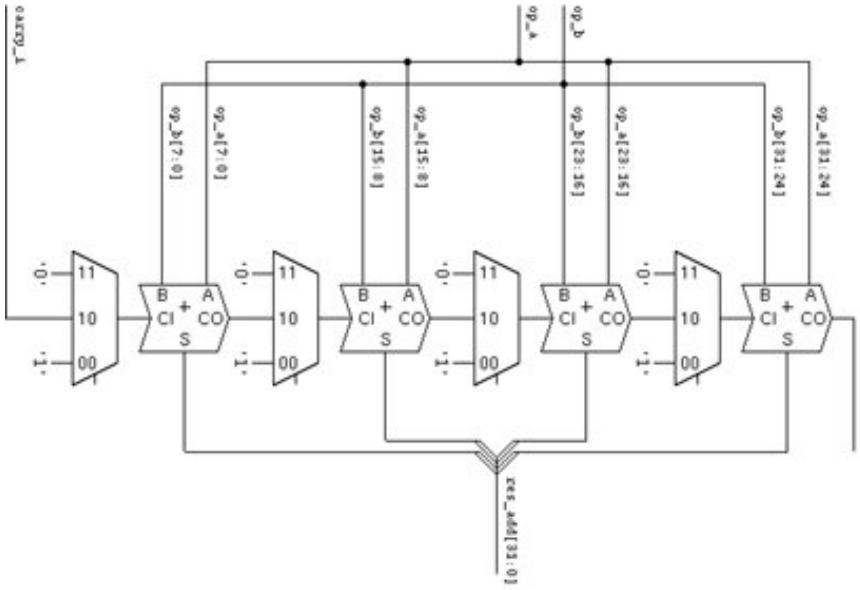
- We reuse the scalar adder
- Supported mode: word**, half, byte**
- We just play some tricks with the carry chain! (32b adder → 35b adder)

word carry
- A[34:0] = {A[31:24], **1'b1**, A[23:16], **1'b1**, A[15:8], **1'b1**, A[7:0]}
- B[34:0] = {B[31:24], **1'b0**, B[23:16], **1'b0**, B[15:8], **1'b0**, B[7:0]}

halfword carry
- A[34:0] = {A[31:24], **1'b1**, A[23:16], **1'b0**, A[15:8], **1'b1**, A[7:0]}
- B[34:0] = {B[31:24], **1'b0**, B[23:16], **1'b0**, B[15:8], **1'b0**, B[7:0]}

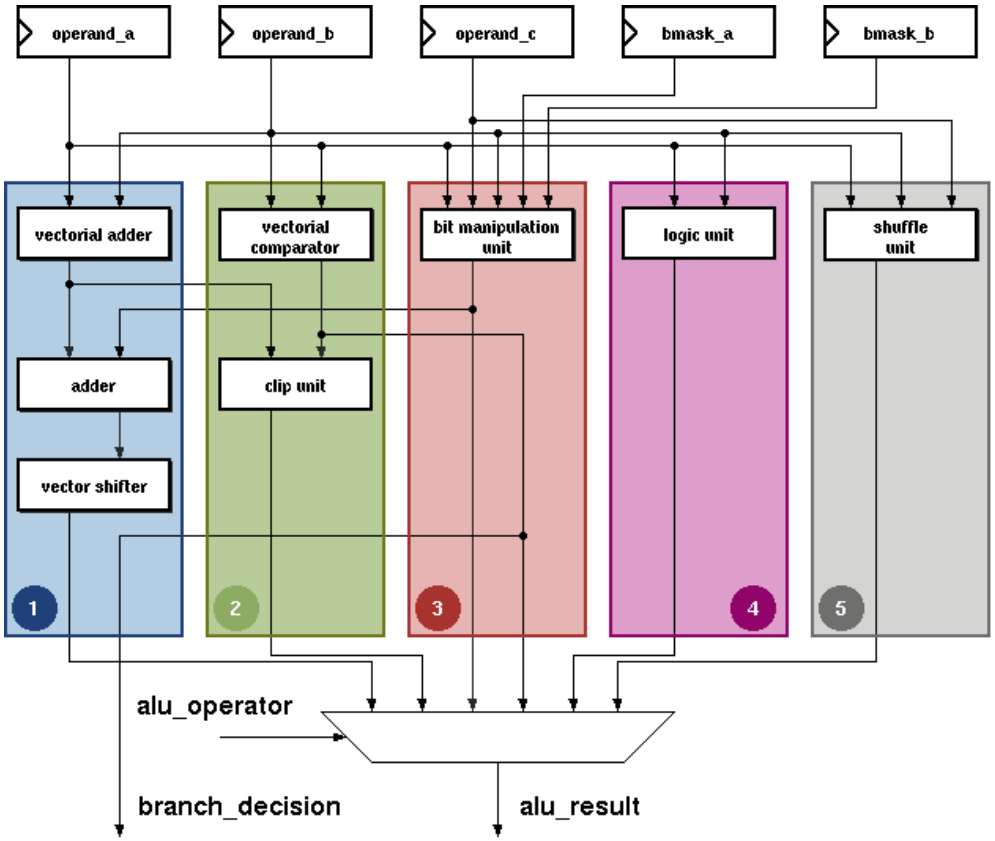
byte carry
- A[34:0] = {A[31:24], **1'b0**, A[23:16], **1'b0**, A[15:8], **1'b0**, A[7:0]}
- B[34:0] = {B[31:24], **1'b0**, B[23:16], **1'b0**, B[15:8], **1'b0**, B[7:0]}

RESULT
- C[34:0] = A+B; ***RES[31:0] = {C[34:27],C[25:18],C[16:9],C[7:0]}***



xPULP ALU Architecture

- Advanced ALU for Xpulp extensions
- Optimized datapath to reduce resources
- Supports vectorial adders for Packed-SIMD operations (32/16/8-b scalar **&** 16/8-b packed-SIMD)
- Adder followed by shifter for fixed point normalization
- Clip unit uses one adder as comparator and the main comparator



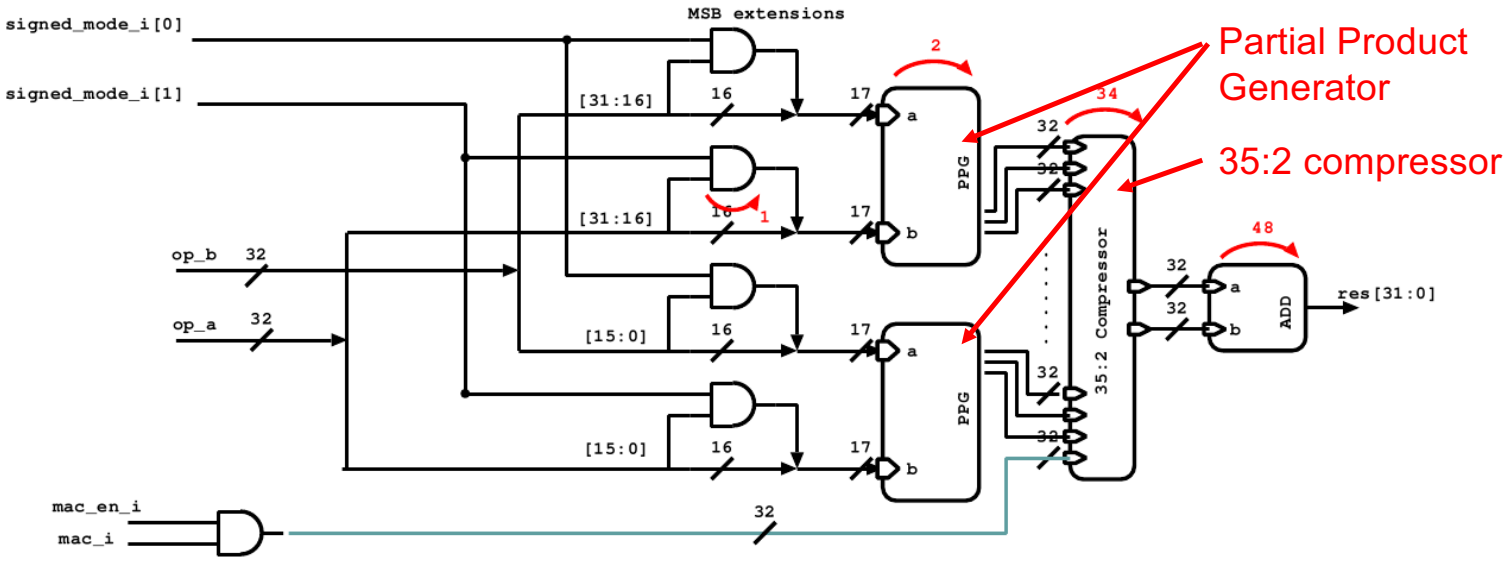
Expanding SIMD Dot Product

- **Packed-SIMD Dot Product: (half word example)**

C[31:0] = A[31:16]*B[31:16] + A[15:0]*B[15:0] + C[31:0]

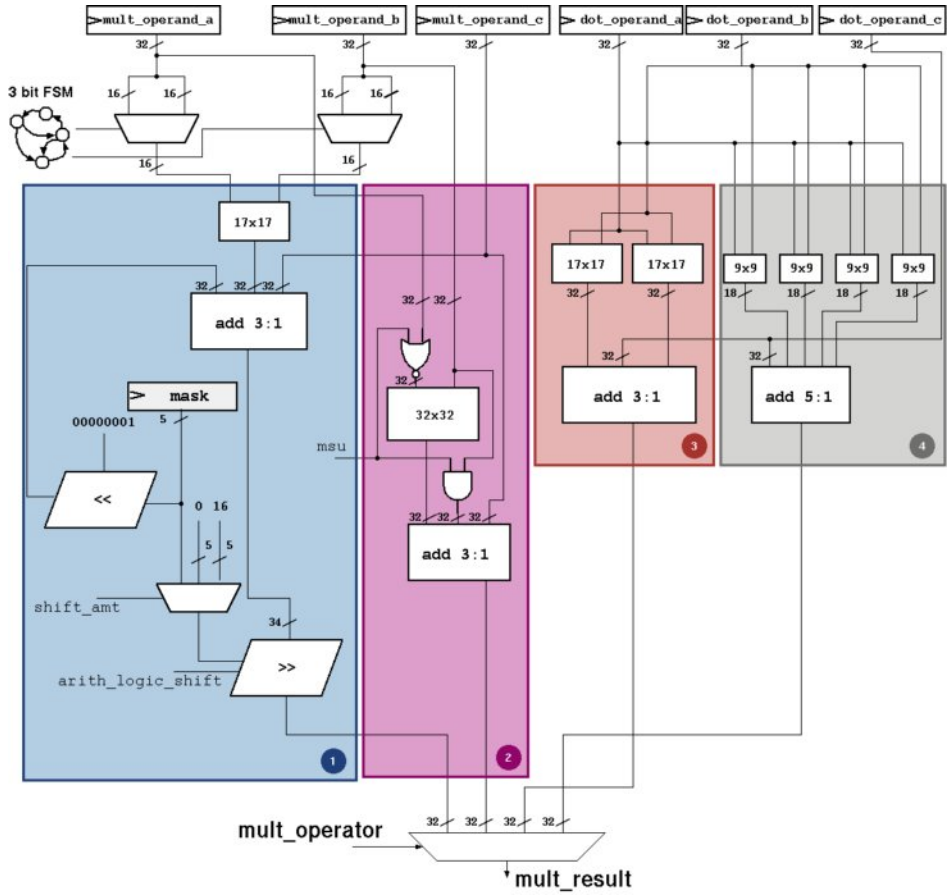
16*16 bit 16*16 bit 32 bit

→ 2 multiplications, 1 addition, 1 accumulation in 1 cycle (2x for bytes)



xPULP Packed-SIMD Multiplier and Dot-Product Unit

- **(blue)** 16x16 (scalar) with sign selection for short multiplications [with round and normalization].
  - 5 cycles FSM for higher 64-bits (mulh* instructions)
- **(purple)** One single cycle mac unit that performs MAC, MSU and MUL
- **(red)** short parallel dot product – 16-b Packed-SIMD
- **(grey)** byte parallel dot product – 8-b Packed-SIMD



Benefits of Packed-SIMD Instructions

- Parallelizes operations (2x to 4x speed gain)
- Minimizes the number of Load/Store instruction for exchanges between memory and register file (up 2 or 4 data transferred at once)
  - Vector Elements likely at contiguous addresses in memory → Fetch 32-b at time (e.g., two 16-b elements of a vector or four 8-b elements of a vector)
- Improve register file use (1 register holds 2 or 4 values)

→ Can be used to dramatically increase the performance of most DSP algorithms running on embedded systems (including filtering, image processing, audio processing, AI)

Compiler Builtins for Extension

No Packed SIMD data type in GCC or LLVM compiler → Hardware SIMD operations to be inferred 'manually' in C code

- We can exploits Intrinsics: special functions that map directly to inlined DSP instructions.

- **Dot-product without accumulation between unsigned/ signed char vectors (v4u/v4s):**

```
S = __builtin_dotu/sp4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3],
A is v4u/v4s, B is v4u/v4s
```

- **Also with mixed signs:**

```
S = __builtin_dotusp4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3] ,
A is v4u, B is v4s
```

- **All of these are also available for 16-b and with accumulation (over accumulator S):**

```
S = __builtin_sdotup4(A, B, S);
S = __builtin_sdotsp4(A, B, S);
S = __builtin_sdotusp4(A, B, S);
S = __builtin_sdotup2(A, B, S);
S = __builtin_sdotsp2(A, B, S);
S = __builtin_sdotusp2(A, B, S);
```

FIR Inner Loop Exec. in RV32IM + MAC + pointer post-incr. + HW Loop + Packed-SIMD (8-bit)

```
for(int j=0; j<coeff_len; j++) {
  sum += arr[i+j] * coeff[j];
}
```

ANSI C

Lower-level view

```
for(int j=0; j< coeff_len/4; j++) {
  int8 arr_int   = arr[i+j];
  int8 coeff_int = coeff[j];
  arr_int = arr_int * coeff_int;
  sum = sum + arr_int;
}
```

```
a7 = coeff_len / 4

lp.setup t3, a7, Lend

Lstart:
p.lw t2, 0x0(t1!)
p.lw a4, 0x0(t3!)
Lend:
pv.sdotp (t0, a4, t2)
```

Loop coeff_len/ 4 times!

Load 1 SIMD vector (32-bit) , of *four* 8-bit elements

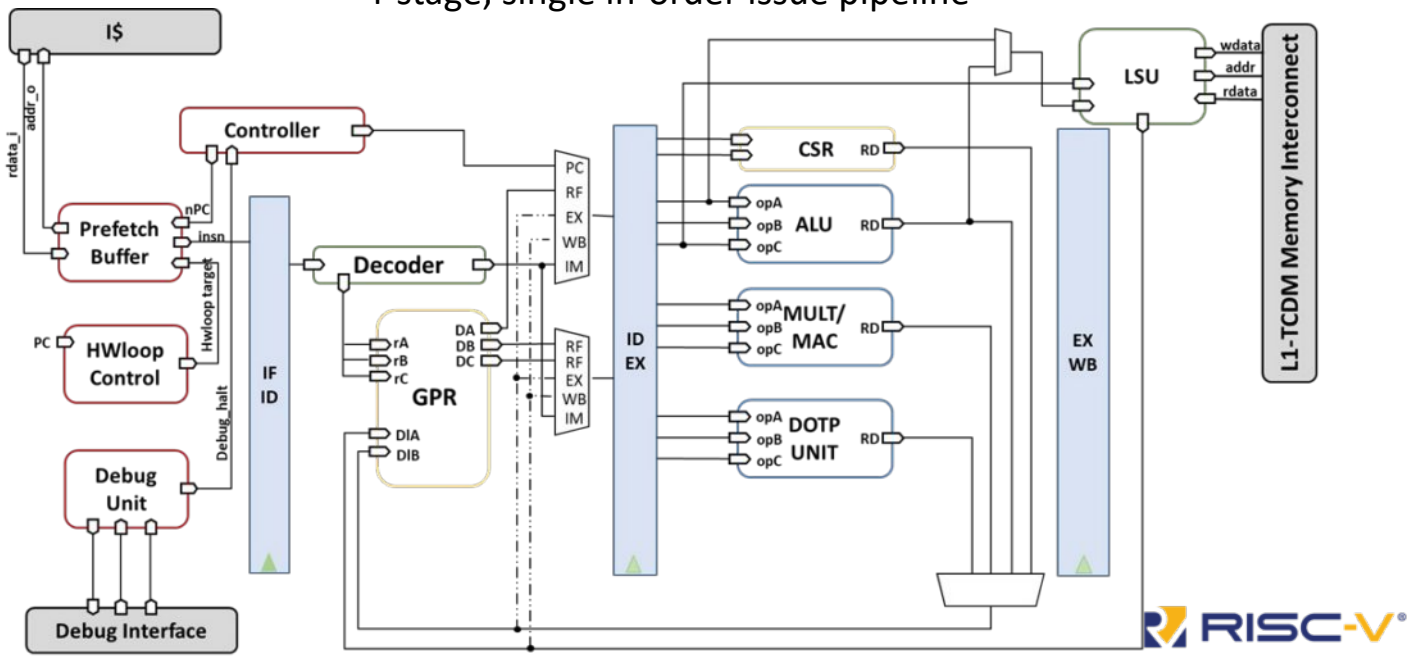
Perform 4 MAC ops on SIMD vectors
Each SIMD vector contains 4 8-bit elements

per iteration:
1 MAC **SIMD** Operation
3 instructions
efficiency = 1/3 = 0.33 ☺

But higher throughput compared to non-SIMD case!!

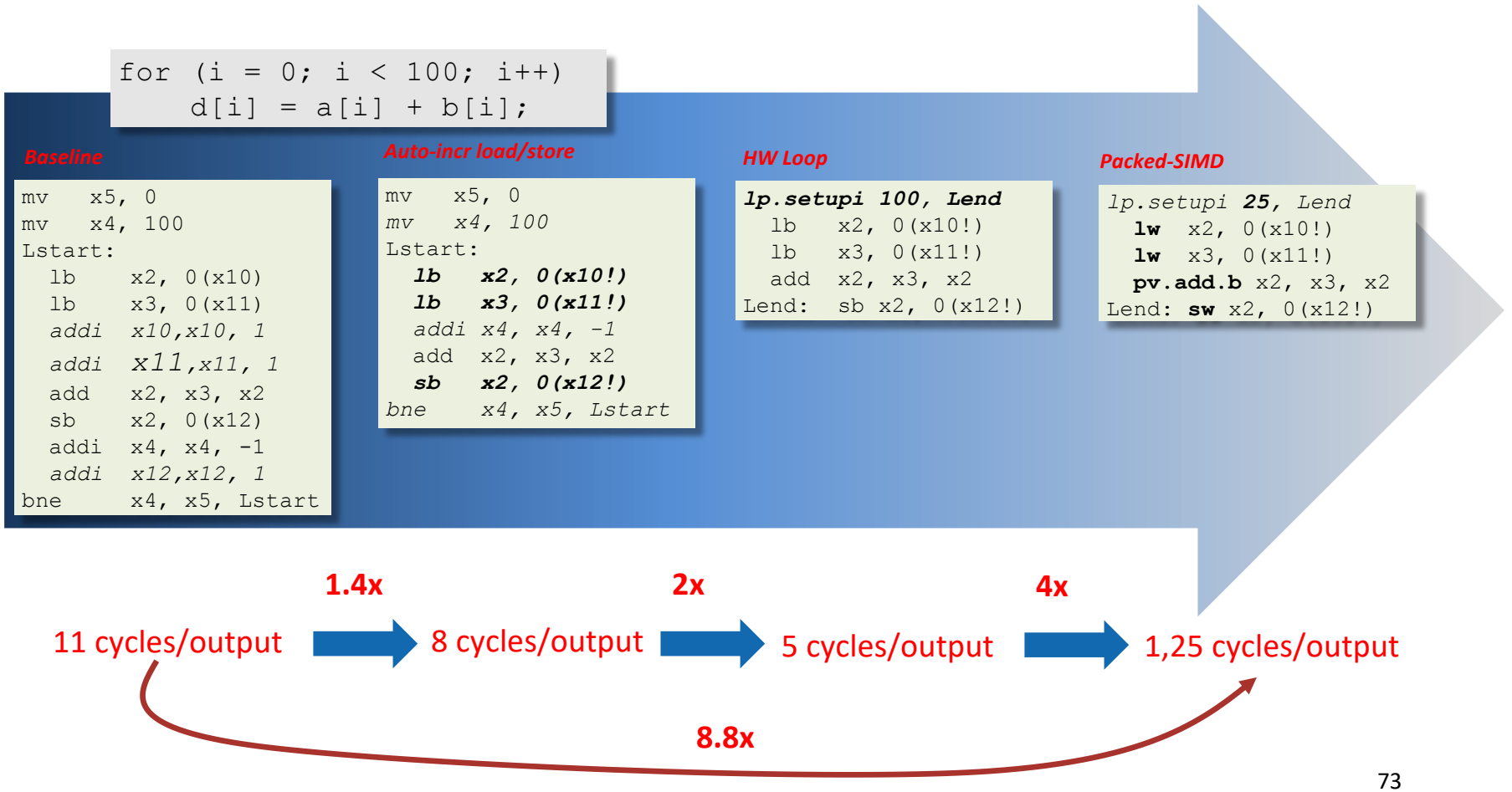
RI5CY Implements the Xpulp ISA

4-stage, single in-order issue pipeline



- **[Baseline] RV32IMC:** not good for machine learning workload (~44kGE)
- [DSP] **Xpulp**: 16/8-bit SIMD, HW loops, post-modified LD/ST & bit-manip. insns (~55kGE)

Extensions at work: RI5CY Implements Xpulp

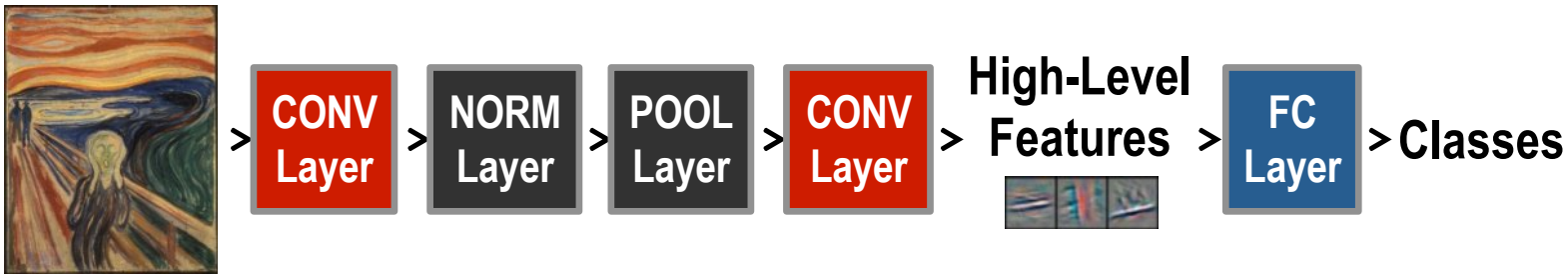


Bringing It All Together – How To Add an ISA Extension

- **Analyze application and identify bottlenecks**
- **Design the instruction (instruction format, encoding, src-dst regs, ..)**
- **Describe the instruction in RISC-V compiler (LLVM or GCC)**
  - Back-end (binutils, disassembler, etc) for debugging & assembly-level or intrinsics usage
  - Front-end (compiler optimization steps) for C to assembly translation and optimization
- **(Optional) Test the new instruction into an instruction set simulator**
- **Describe the micro-architecture (extend the core etc)**
- **Evaluate the Power, Performance (timing), Area costs of the micro-archi**
  - Through physical implementation of the micro-architecture targeting a technology

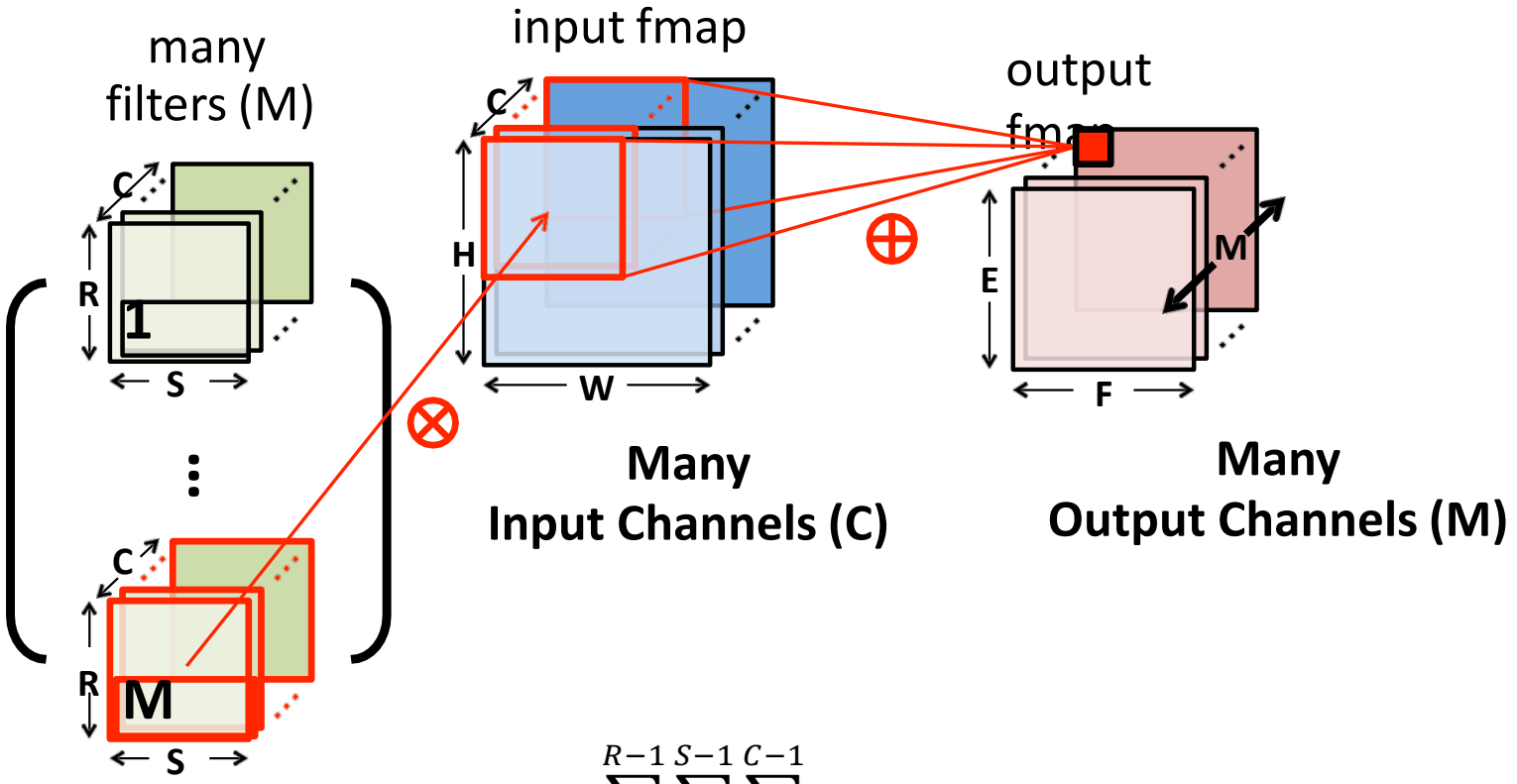
Iterations may be needed

Accelerating AI Through HW/SW Domain-Specialization



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

The Convolution Layer



$$O[m][x][y] = \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} \sum_{k=0}^{C-1} \boldsymbol{I}[n][k][Ux+i][Uy+j] \cdot \boldsymbol{W}[m][k][i][j]$$

Naïve 6-Loop Implementation of Conv Layers

```
for (m=0; m<M; m++) {
    for (x=0; x<F; x++) {
        for (y=0; y<E; y++) {
            O[n][m][x][y] = B[m];

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

for each output fmap value

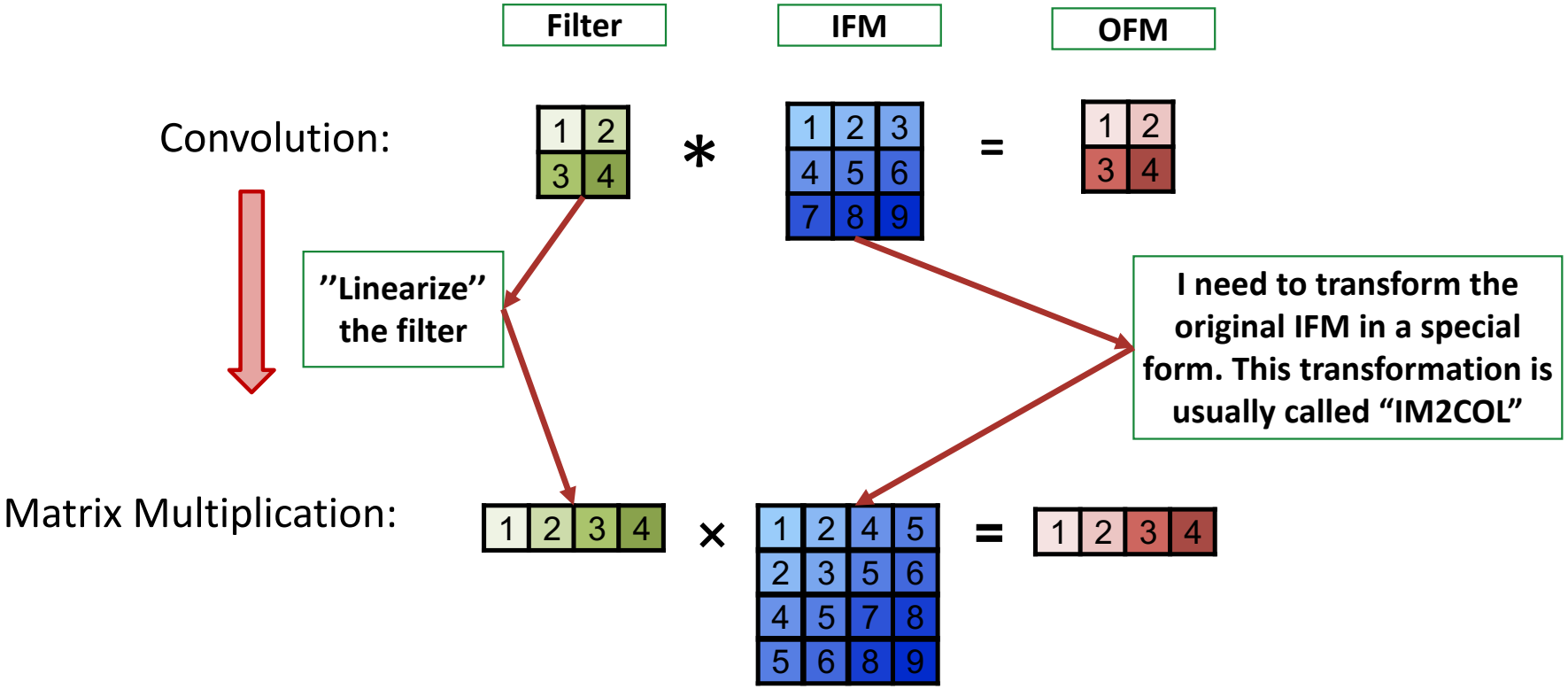
convolve a window and apply activation

How?

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

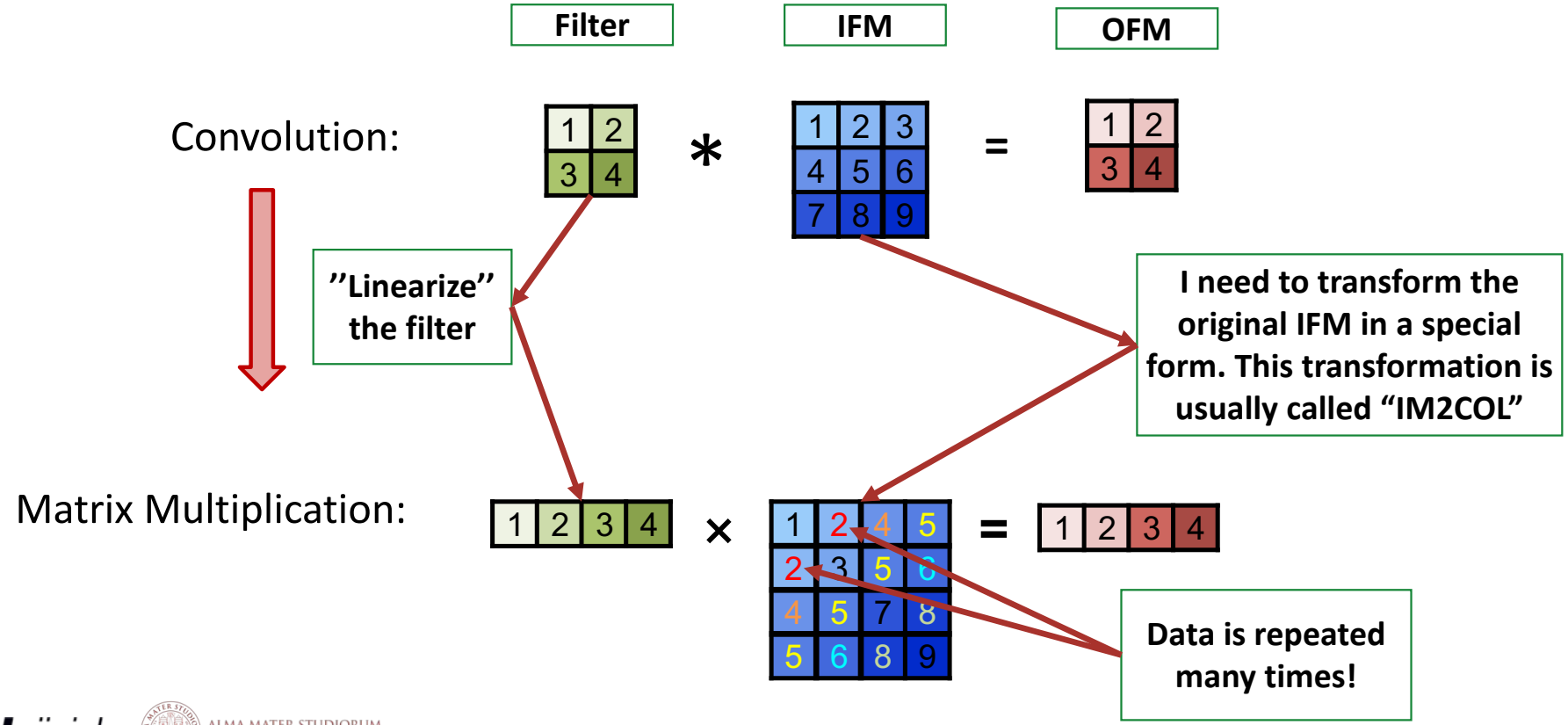
Convolution Layer as a Matrix Multiplication

- **Convert to Matrix Multiplications by using Toeplitz Matrices**

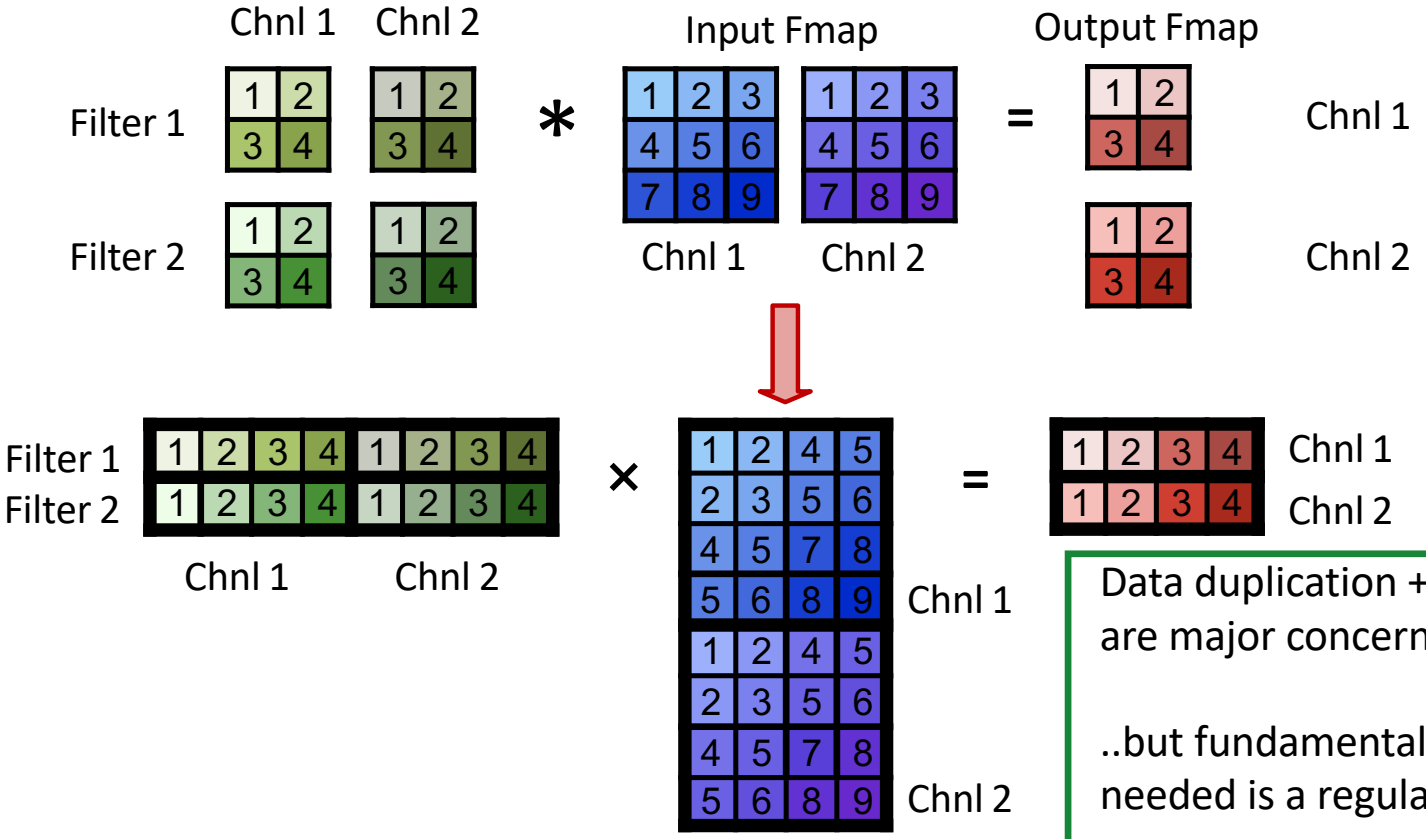


Convolution Layer as a Matrix Multiplication

- **Nothing comes for free → A lot of data duplication! (Memory overhead)**



Conv Layer With Multiple Channels and Filters



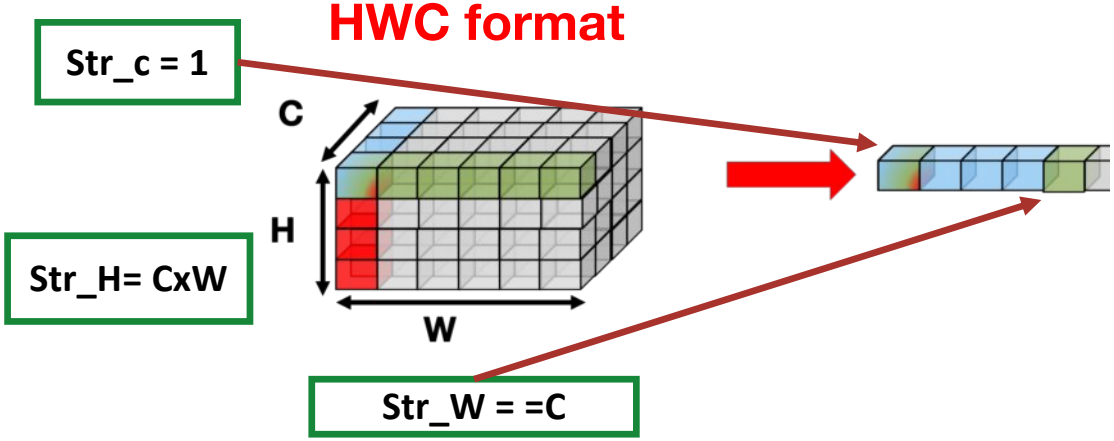
Data duplication + Memory bandwidth are major concerns

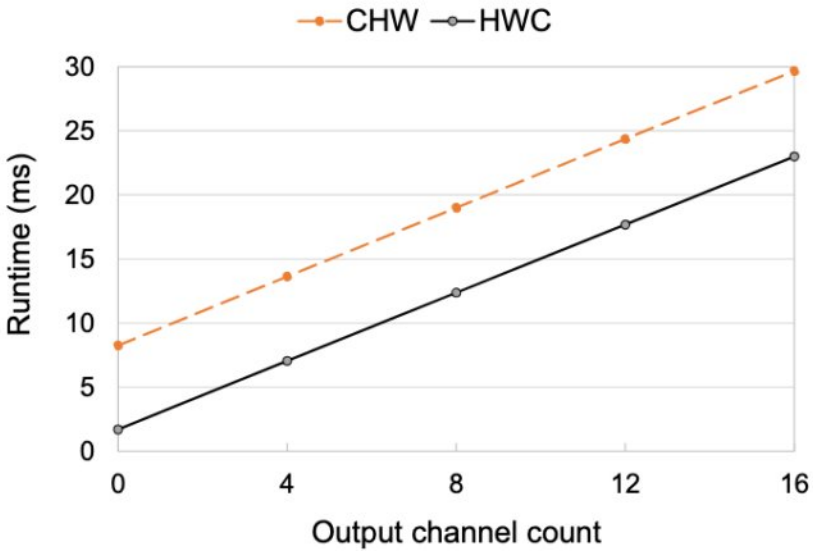
..but fundamentally, the operation needed is a regular **DOT PRODUCT**

$$O = r \cdot c = \sum_{i=1}^{N} c_i w_i$$

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

- **C-H-W → stores the spatial dimensions at contiguous memory addresses**
  - → less efficient for convolutions as more 'memory' operations and data manipulations required for the ***IM2COL*** *phase (especially for low-bitwidth arith and exploiting Packed SIMD)*
- **H - W– C → stores channels at contiguous addresses (preferred layout in edge devices)**
  - → more efficient data movements to build ***IM2COL*** buffers (especially for low-bitwidth precisions exploiting packed-SIMD computation)



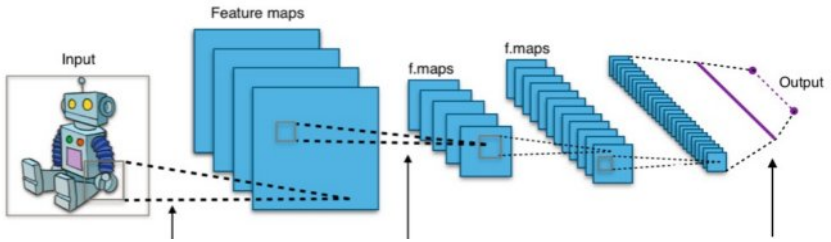


PULP-NN [Garofalo 19] https://arxiv.org/abs/1908.11263

ARM CMSIS-NN [Cmsis2018]

PULP-NN: Optimized AI Kernels for RISC-V Devices

- **Compute RISC-V Backend Library for QNN inference at the Edge**
  - Support for GEMM/GEMV/Convolutions, but also non-linear layers
    - Multiple precisions: 16-b/8-b/4-b/2-b/1-b → symmetric and asymmetric (Mixed precisions)
- 1) Optimized convolution through efficient IM2COL transformations
- 2) Optimized Data Layout (HWC)
- 3) Maximizes data reuse of “weights” and “activations” at RF level
- 4) Exploits the RV32IMCXpulp ISA for efficient SIMD computation
- 5) exploits multi-core cluster execution (more in the next lecture)
- **Open-Source at**
  - https://github.com/pulp-platform/pulp-nn
  - https://github.com/pulp-platform/pulp-nn-mixed



PULP-NN: optimized computational back-end

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

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

Psum stationary in RF

PULP-NN: optimized computational back-end

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

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

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

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

PULP-NN: optimized computational back-end

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

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

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

PULP-NN: optimized computational back-end

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

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

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

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

PULP-NN: optimized computational back-end

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

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

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

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

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

PULP-NN: optimized computational back-end

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

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

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

PULP-NN: optimized computational back-end

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

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

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

PULP-NN: optimized computational back-end

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

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

Optimizatons:
- Reuse same im2col over multiple conv filters
- Build more im2col buffers in //

→ **LOOP Unrolling + Data Reuse in RF**

→ **Until you saturate RF regs**

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

PULP-NN: 2x2 kernel, Xpulp Exploitation

RV32IMC

N

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

N/4

RV32IMCXpulp

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

8x less instructions than RV32IMC

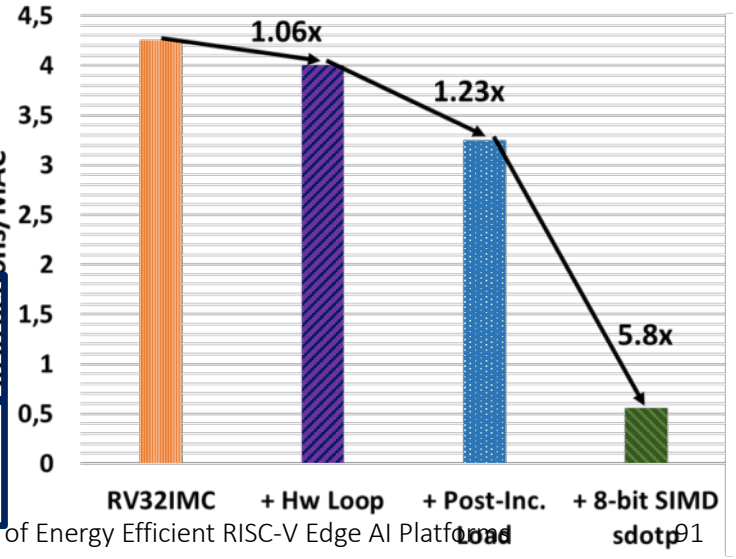
Compute eff: **4ops/8insn = 0.5** ☺

8-bit Convolution – 2x2 MatMul kernel

HW Loop

LD/ST with post increment

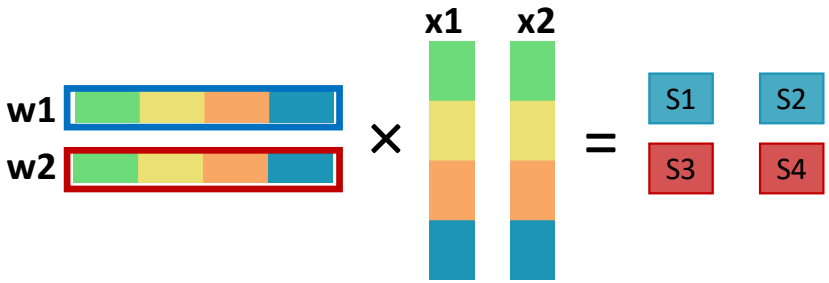
8-bit SIMD sdotp



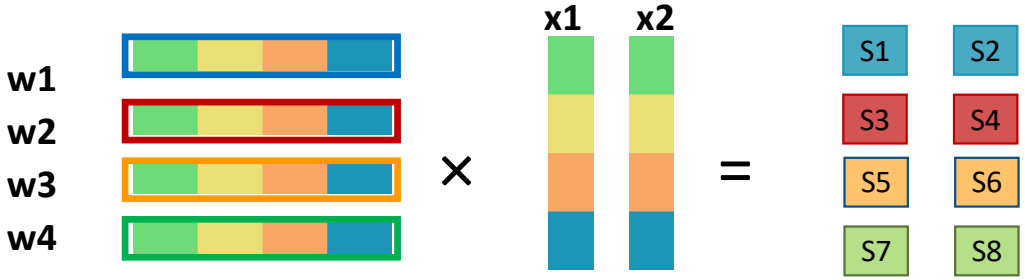
Can we Exploit More Data Reuse at RF Level?

8-bit Convolution

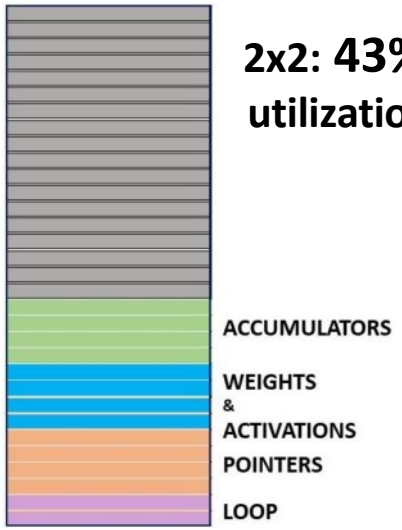
Initial Matrix Multiplication Layout (from CMSIS-NN) : 2x2



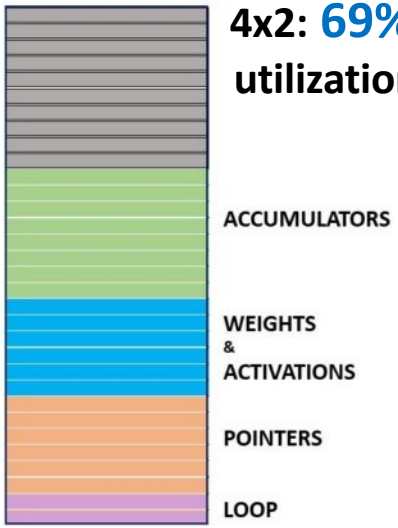
PULP-NN Matrix Multiplication Layout: 4x2



RegisterFile of the RI5CY core: 32 general purpose registers



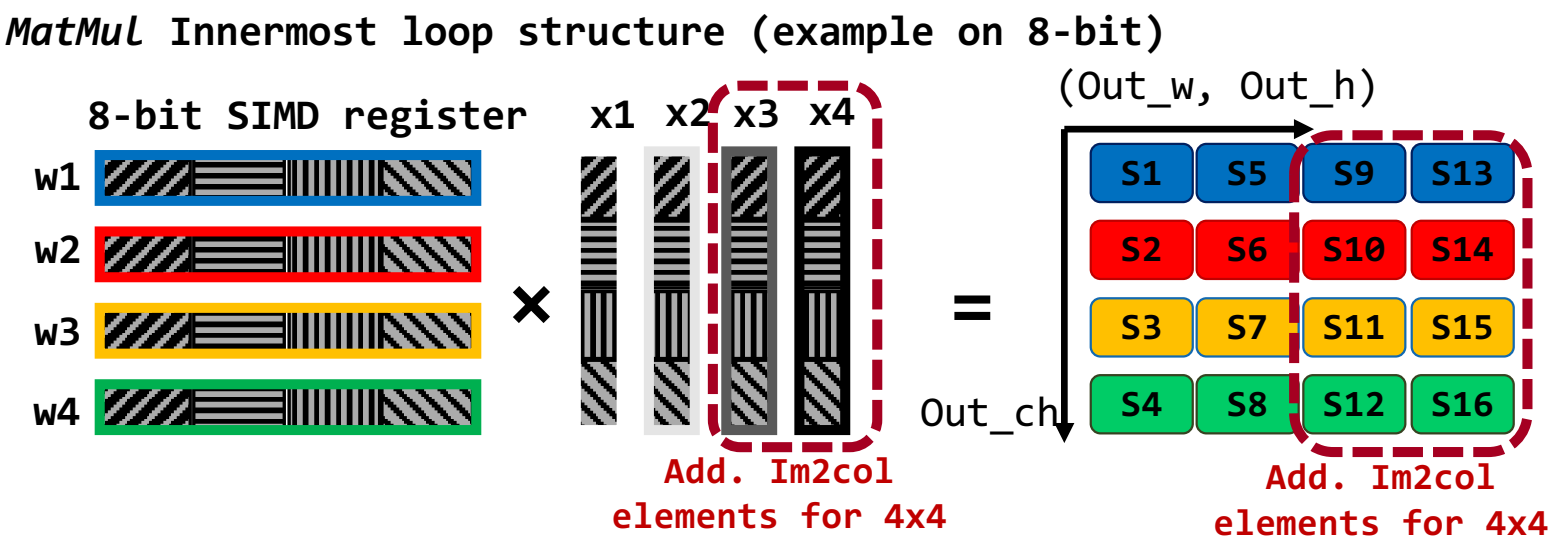
2x2: 43% utilization



4x2: 69% utilization

More Data Reuse
&
Higher utilization of the RF

Can We Improve Further?



Register File Resources required	4x2 kernel	4x4 kernel
	SIMD	SIMD
Loop Setup	2	2
Addresses for MEM access	6	8
MAC operands	6	8
MAC accumulators	8	16
Total	22 regs	34 regs

PULP-NN: Optimized Back-End

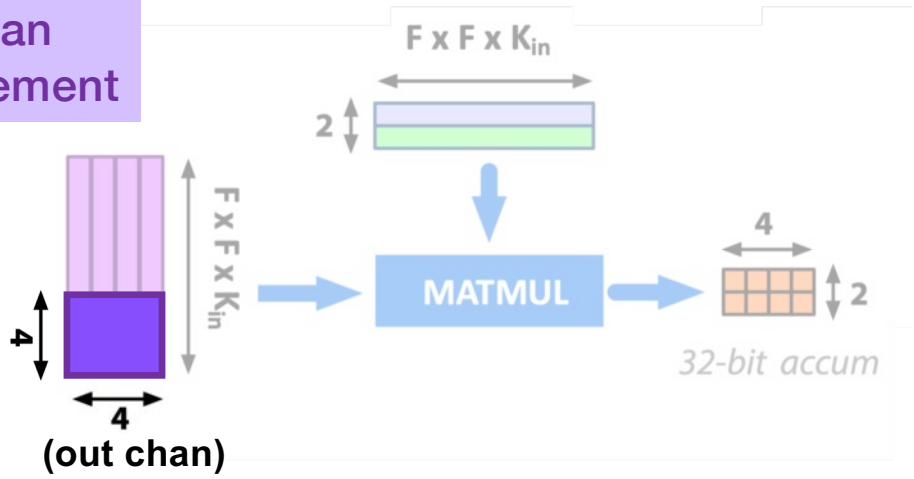
Target **int8** execution of CONV, FC, ... primitives

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

HWC format

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

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



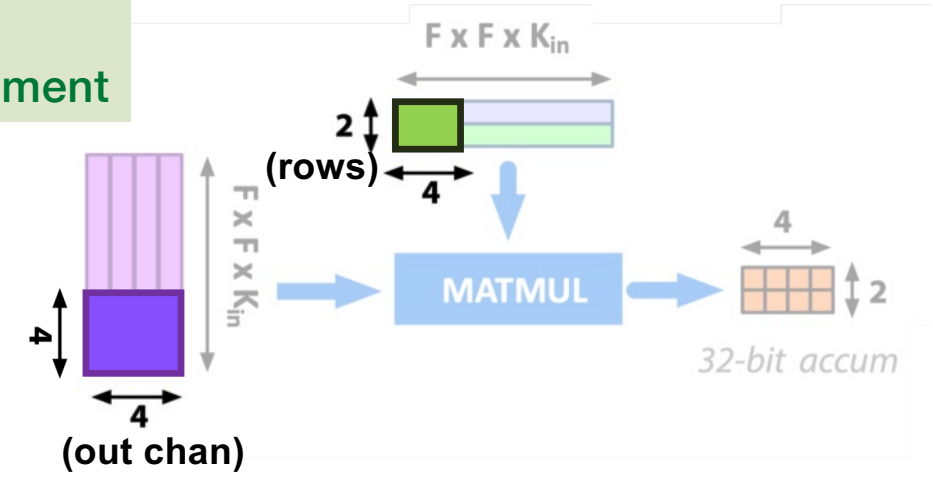
PULP-NN [Garofalo 19] https://arxiv.org/abs/1908.11263

PULP-NN: Optimized Back-End

Target **int8** execution of CONV, FC, ... primitives
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

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

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



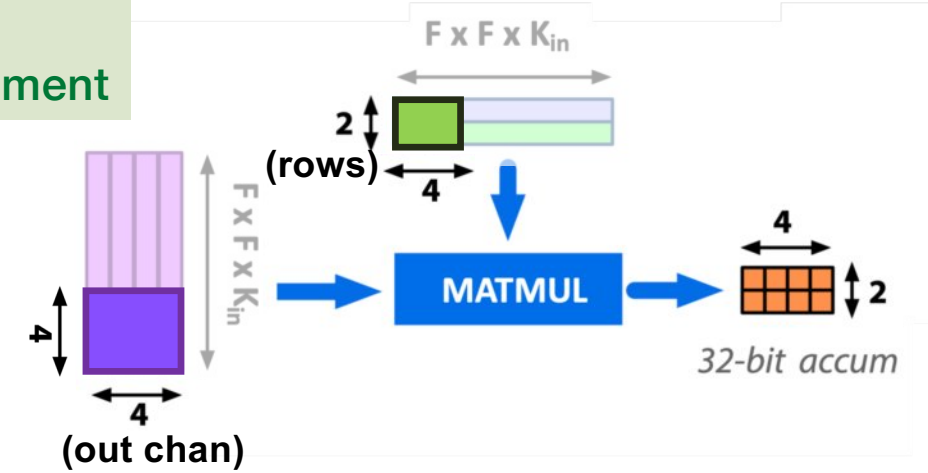
PULP-NN [Garofalo 19] https://arxiv.org/abs/1908.11263

PULP-NN: Optimized Back-End

Target **int8** execution of CONV, FC, ... primitives
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

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

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



Compute 32 MAC over 8 accumulators
dot-product instructions

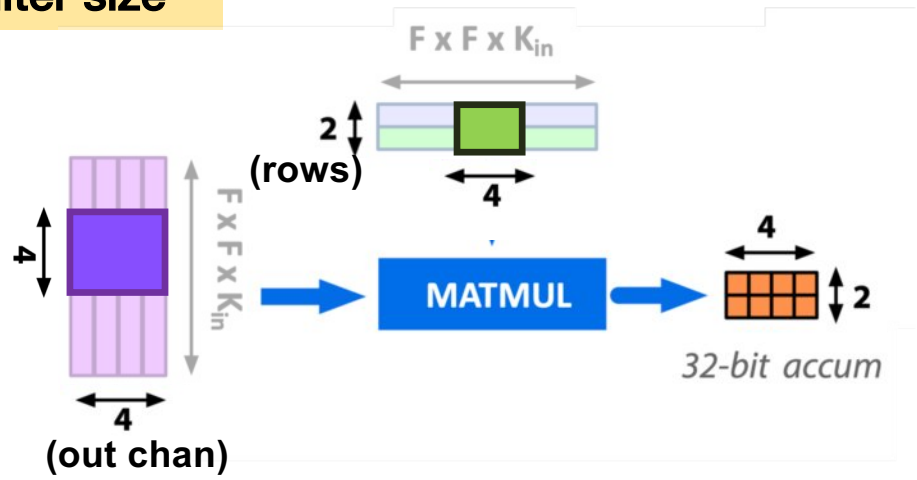
PULP-NN [Garofalo 19] https://arxiv.org/abs/1908.11263

PULP-NN: Optimized Back-End

Target **int8** execution of CONV, FC, ... primitives
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

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

Loop over in chan, filter size



Comp eff: 8ops/14insn = 0.57

PULP-NN [Garofalo 19] https://arxiv.org/abs/1908.11263

Fused Mac-Load SIMD Instruction (Xpulpnn)

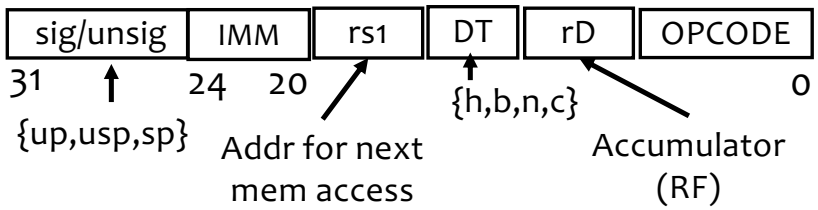
- **Goal:** Increase Operation Efficiency, i.e. reducing the cost of LOADs
- **Observation 1:** L1 Memory Access Pattern Extremely Regular during *MatMuls*
  - At *i-th* iteration of the innermost loop we already now which are next operands to be fetched from the memory to be consumed during *(i+1)-th* iteration
- **Observation 2:** create dedicated registers (NN-RF) to alleviate the pressure on the GP-RF and increase data reuse
- **Intuition:**
  - Fuse SIMD (*s)dotp* and LOAD ops in one single-cycle latency insns → reduce cost of LOADs
    - It's not a dual-issue micro-archi optimization..
  - Inputs for (s)dotp/output for LOAD from/to NN-RF → alleviate pressure on GP-RF of the processor
- **SW:** optimized routines to use mac-load, and intense data reuse strategy at RF/NN-RF level

Fused Mac-Load SIMD Instruction

- The Mac-Load specifies the **operations on a dedicated NN-RF** (instead of relying on the GP-RF)
- After DSE, the NN-RF is paritioned with 2 slots (regs) for “activations” and 4 slots for “weights”

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

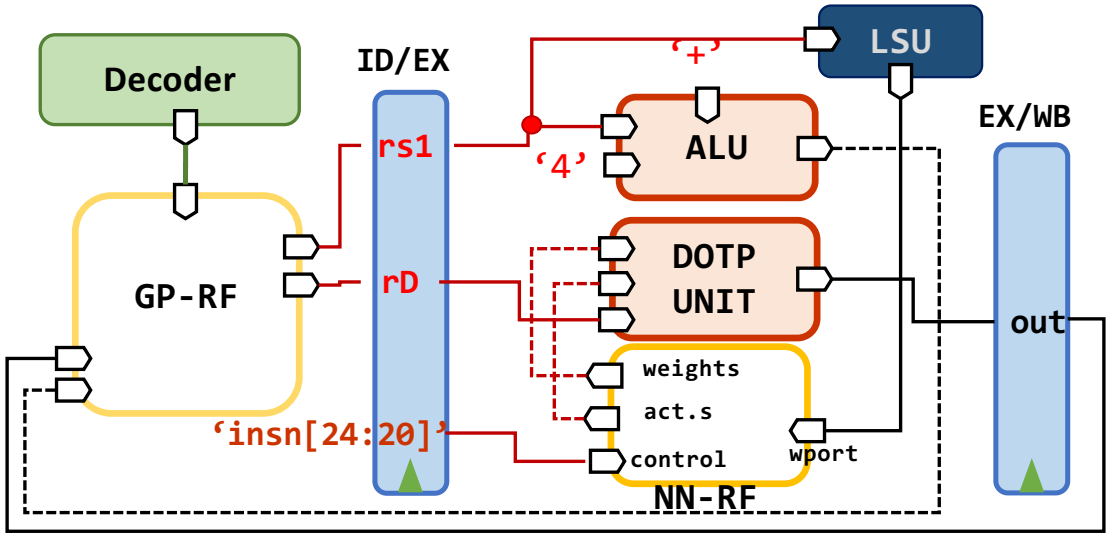
pv.mlsdot{up,usp,sp}.{h,b,n,c} rD, rs1, IMM



5-b IMMEDIATE description

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

Mac-load micro-archi



Performance Analysis on Assembly Code

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

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

HW LOOP

LD/ST WITH POST INCREMENT

8-B SIMD MAC

8 SIMD MACs with 6 explicit LOADs

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

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

8 SIMD MACs with 1 explicit LOAD

W_i X_i : weight/activation elements

AW_i AX_i : addresses for the MEM access

S_i : accumulators

L_i : loop setup

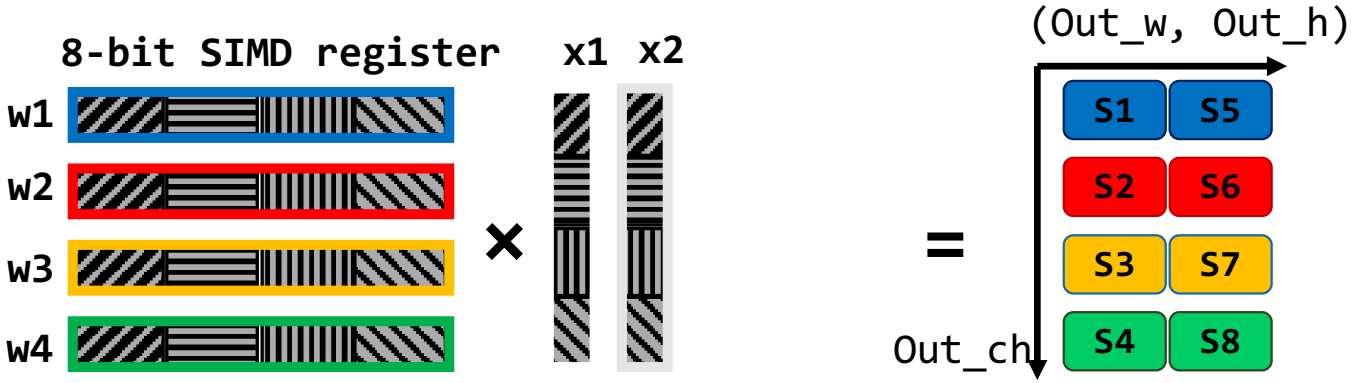
$$OpEff = 0.89$$

Can we still do better?

Goal: Increase Operation Efficiency (XpulpNN)

Can We Do Better? Analysis of Processor Resources

***MatMul* Innermost loop structure (example on 8-bit)**

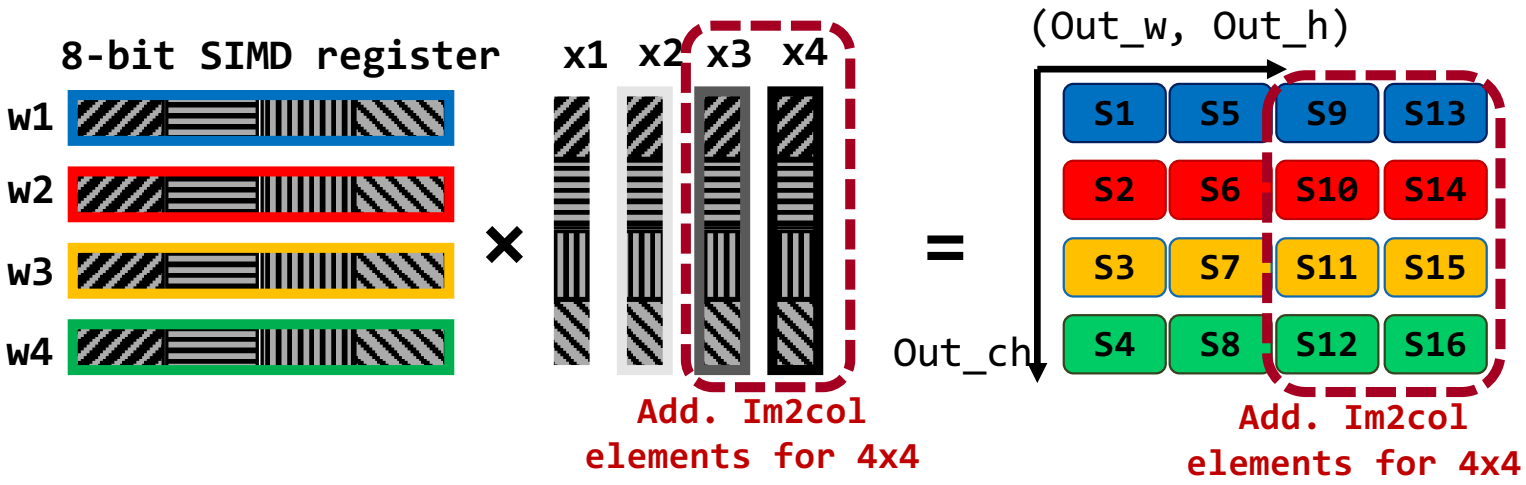


Register File Resources required	4x2 kernel
	SIMD
Loop Setup	2
Addresses for MEM access	6
MAC operands	6
MAC accumulators	8
Total	22 regs

Goal: Increase Operation Efficiency (XpulpNN)

Can We Do Better? Analysis of Processor Resources

MatMul **Innermost loop structure (example on 8-bit)**



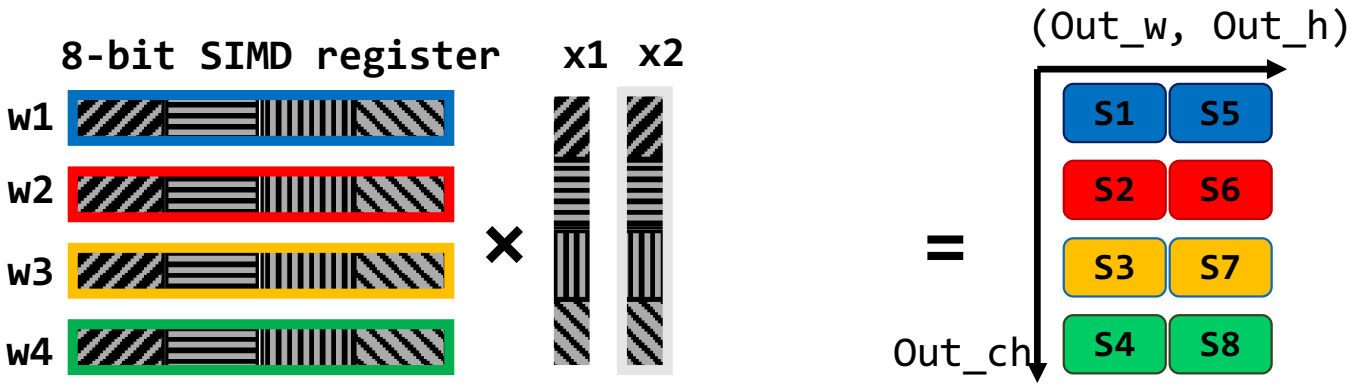
Register File Resources required	4x2 kernel
	SIMD
Loop Setup	2
Addresses for MEM access	6
MAC operands	6
MAC accumulators	8
Total	22 regs

4x4 kernel
SIMD
2
8
8
16
34 regs

Goal: Increase Operation Efficiency (XpulpNN + M&L)

Can We Do Better? Analysis of Processor Resources

***MatMul* Innermost loop structure (example on 8-bit)**

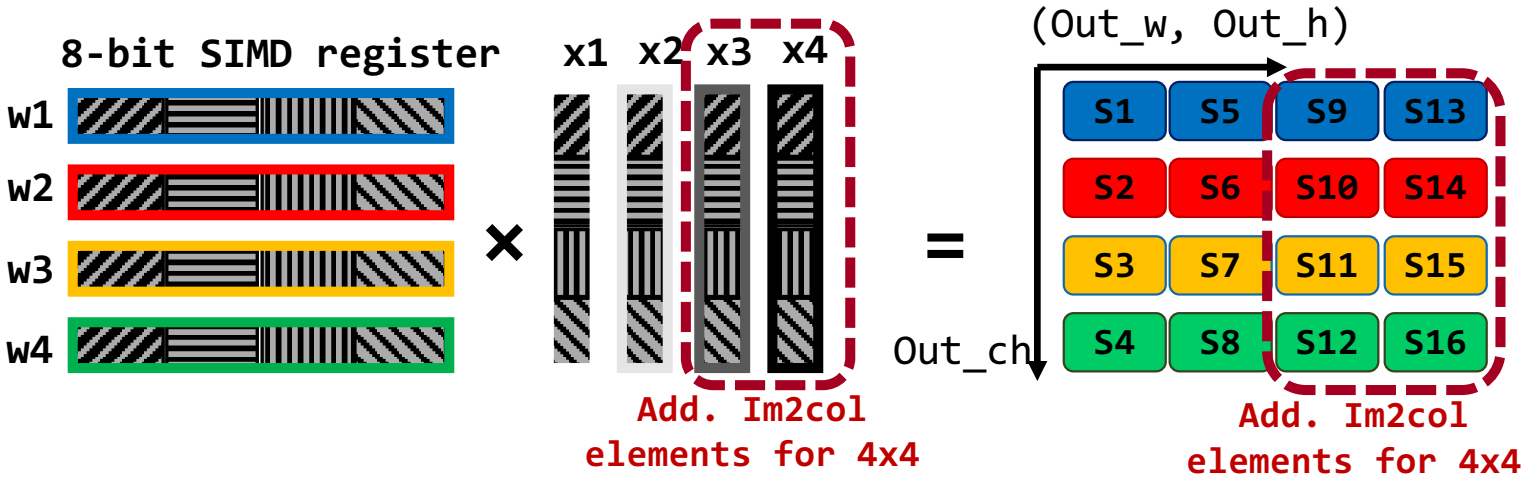


Register File Resources required	4x2 kernel		4x4 kernel
	SIMD	SIMD+mlsdotp	SIMD
Loop Setup	2	2	2
Addresses for MEM access	6	6	8
MAC operands	6	0 (op. from NN-RF)	8
MAC accumulators	8	8	16
Total	22 regs	16 regs	**34 regs**

Goal: Increase Operation Efficiency (XpulpNN + M&L)

Can We Do Better? Analysis of Processor Resources

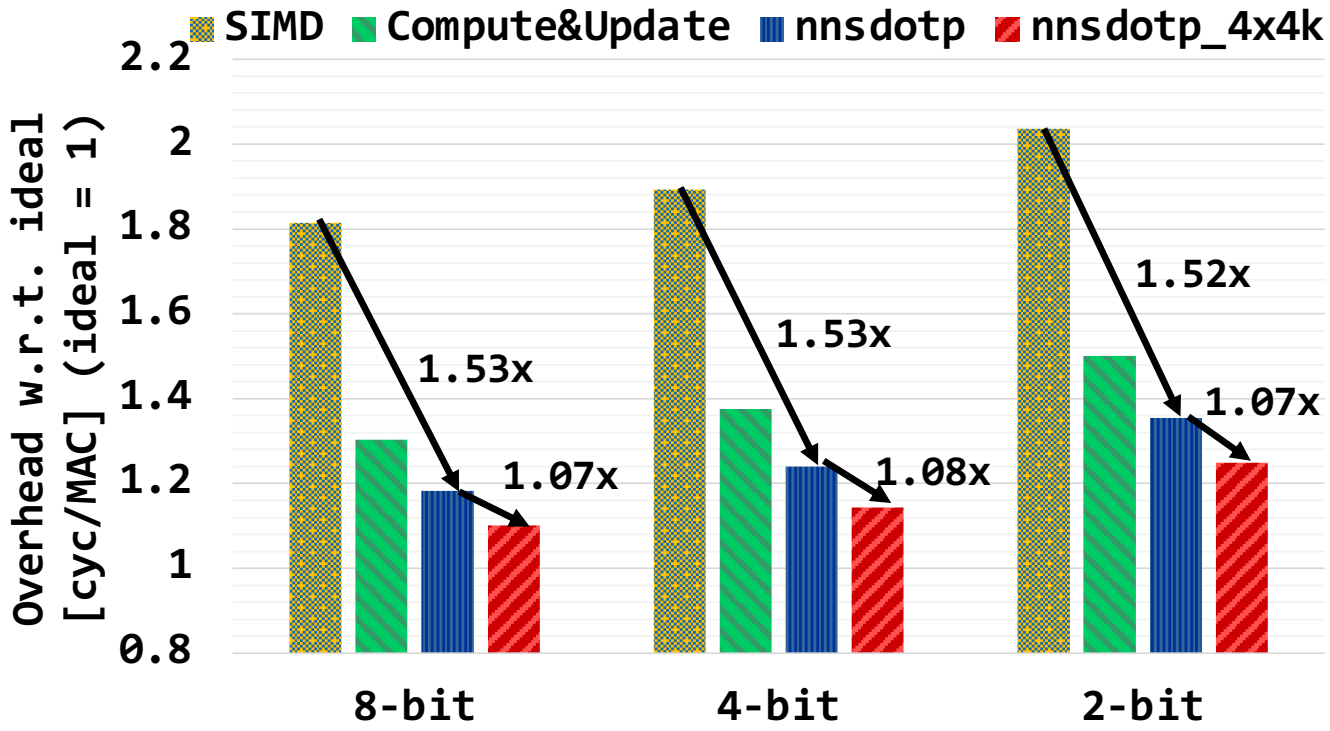
***MatMul* Innermost loop structure (example on 8-bit)**



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

M&L Operation Efficiency

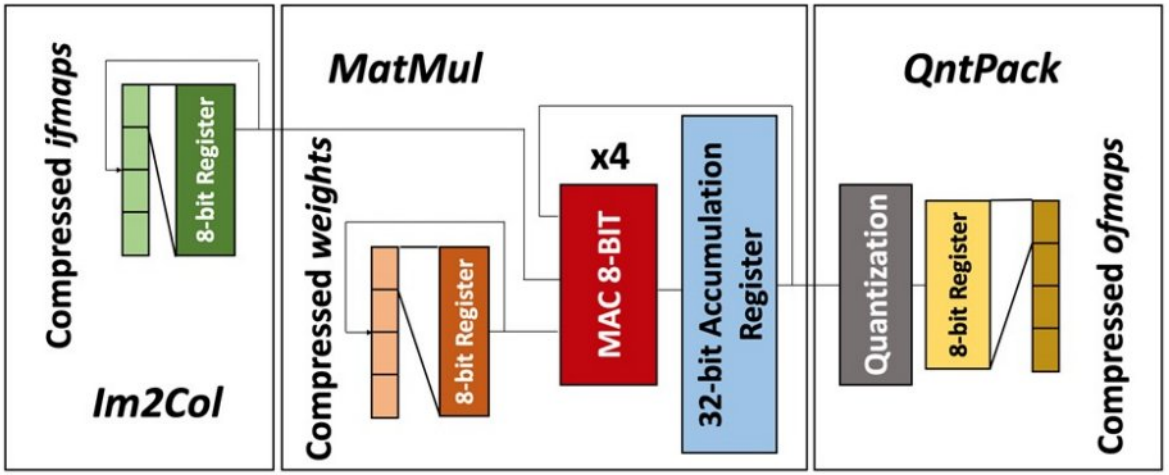
Cycles to compute one MAC operation within the *MatMul* kernel. Single Core Execution.



- Utilization of the HW resources (Multi-precision Dotp Unit) **>90% (~94% best case)** during kernel execution;
- Btw, using M&L seems quite complicated…a user never wants to deal with that
  - → Optimized SW exposed to user as API

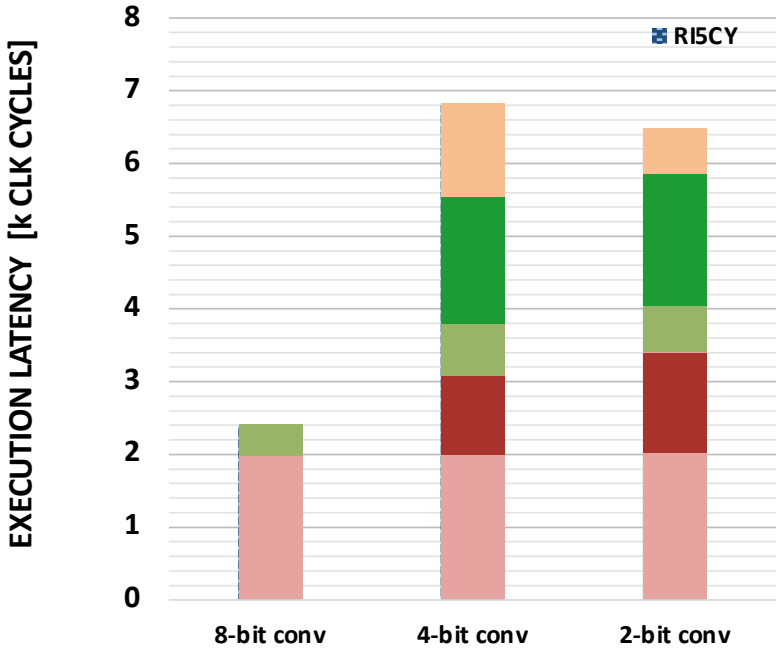
What About Low-Byte and Mixed-Precisions?

- **4-b, 2-b, 1-b (or mixed-precision) common for some heavily QNNs**
  - Reduced footprint of the model → good for memory and BW transfers
  - Computation on less bits → Higher energy-efficiency if well supported by the processor
- **Xpulp does not natively support 4-b, 2-b, 1-b operations in the ISA**
  - Unpacking necessary to feed the 8-b SIMD units → Huge Overhead!!



Analyzing Overheads to Eliminate Them

Sub-byte Convolution Kernel on Xpulp ISA



MatMul kernel

Overhead to extract compressed weights

Im2col function

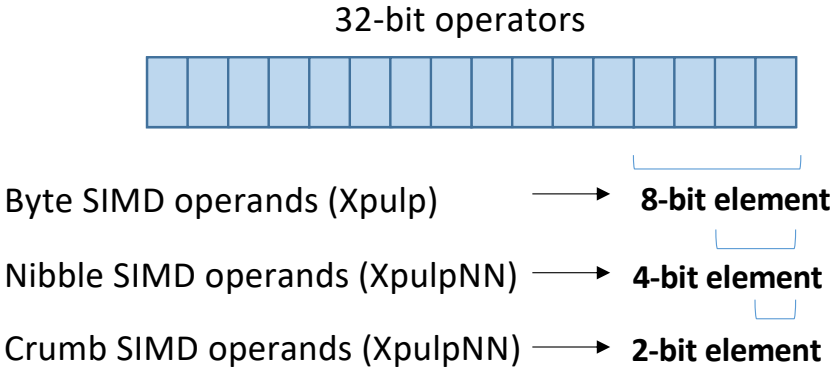
Overhead to extract compressed im2col

Re-Quantization

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

xPULPNN: RISC-V ISA Extensions

Arithmetic SIMD instructions



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

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

Mixed-Precision

8x4 SIMD sdotp operation

RI5CY Assembly

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

- High Performance overhead!
  - Many extra instructions for bit manipulation

Mixed-Precision: Dynamic Bit-Scalable Execution

8x4 SIMD sdotp operation

RI5CY Assembly

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

ISA extensions

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

- One instruction per format and type

Standard Instructions

pv.dotsp.h	pv.dotup.h	pv.dotusp.h	pv.sdotsp.h	pv.sdotup.h	pv.sdotusp.h
pv.dotsp.b	pv.dotup.b	pv.dotusp.b	pv.sdotsp.b	pv.sdotup.b	pv.sdotusp.b
pv.dotsp.n	pv.dotup.n	pv.dotusp.n	pv.sdotsp.n	pv.sdotup.n	pv.sdotusp.n
pv.dotsp.c	pv.dotup.c	pv.dotusp.c	pv.sdotsp.c	pv.sdotup.c	pv.sdotusp.c
pv.dotsp.m4x2	pv.dotup.m4x2	pv.dotusp.m4x2	pv.sdotsp.m4x2	pv.sdotup.m4x2	pv.sdotusp.m4x2
pv.dotsp.m8x2	pv.dotup.m8x2	pv.dotusp.m8x2	pv.sdotsp.m8x2	pv.sdotup.m8x2	pv.sdotusp.m8x2
pv.dotsp.m8x4	pv.dotup.m8x4	pv.dotusp.m8x4	pv.sdotsp.m8x4	pv.sdotup.m8x4	pv.sdotusp.m8x4
pv.dotsp.m16x8	pv.dotup.m16x8	pv.dotusp.m16x8	pv.sdotsp.m16x8	pv.sdotup.m16x8	pv.sdotusp.m16x8
pv.dotsp.m16x4	pv.dotup.m16x4	pv.dotusp.m16x4	pv.sdotsp.m16x4	pv.sdotup.m16x4	pv.sdotusp.m16x4
pv.dotsp.m16x2	pv.dotup.m16x2	pv.dotusp.m16x2	pv.sdotsp.m16x2	pv.sdotup.m16x2	pv.sdotusp.m16x2
pv.dotsp.sc.h	pv.dotup.sc.h	pv.dotusp.sc.h	pv.sdotsp.sc.h	pv.sdotup.sc.h	pv.sdotusp.sc.h
pv.dotsp.sc.b	pv.dotup.sc.b	pv.dotusp.sc.b	pv.sdotsp.sc.b	pv.sdotup.sc.b	pv.sdotusp.sc.b
pv.dotsp.sc.c	pv.dotup.sc.c	pv.dotusp.sc.c	pv.sdotsp.sc.c	pv.sdotup.sc.c	pv.sdotusp.sc.c
pv.dotsp.sc.n	pv.dotup.sc.n	pv.dotusp.sc.n	pv.sdotsp.sc.n	pv.sdotup.sc.n	pv.sdotusp.sc.n
pv.dotsp.sc.m4x2	pv.dotup.sc.m4x2	pv.dotusp.sc.m4x2	pv.sdotsp.sc.m4x2	pv.sdotup.sc.m4x2	pv.sdotusp.sc.m4x2
pv.dotsp.sc.m8x2	pv.dotup.sc.m8x2	pv.dotusp.sc.m8x2	pv.sdotsp.sc.m8x2	pv.sdotup.sc.m8x2	pv.sdotusp.sc.m8x2
pv.dotsp.sc.m8x4	pv.dotup.sc.m8x4	pv.dotusp.sc.m8x4	pv.sdotsp.sc.m8x4	pv.sdotup.sc.m8x4	pv.sdotusp.sc.m8x4
pv.dotsp.sc.m16x8	pv.dotup.sc.m16x8	pv.dotusp.sc.m16x8	pv.sdotsp.sc.m16x8	pv.sdotup.sc.m16x8	pv.sdotusp.sc.m16x8
pv.dotsp.sc.m16x4	pv.dotup.sc.m16x4	pv.dotusp.sc.m16x4	pv.sdotsp.sc.m16x4	pv.sdotup.sc.m16x4	pv.sdotusp.sc.m16x4
pv.dotsp.sc.m16x2	pv.dotup.sc.m16x2	pv.dotusp.sc.m16x2	pv.sdotsp.sc.m16x2	pv.sdotup.sc.m16x2	pv.sdotusp.sc.m16x2
pv.dotsp.sci.h	pv.dotup.sci.h	pv.dotusp.sci.h	pv.sdotsp.sci.h	pv.sdotup.sci.h	pv.sdotusp.sci.h
pv.dotsp.sci.b	pv.dotup.sci.b	pv.dotusp.sci.b	pv.sdotsp.sci.b	pv.sdotup.sci.b	pv.sdotusp.sci.b
pv.dotsp.sci.c	pv.dotup.sci.c	pv.dotusp.sci.c	pv.sdotsp.sci.c	pv.sdotup.sci.c	pv.sdotusp.sci.c
pv.dotsp.sci.n	pv.dotup.sci.n	pv.dotusp.sci.n	pv.sdotsp.sci.n	pv.sdotup.sci.n	pv.sdotusp.sci.n
pv.dotsp.sci.m4x2	pv.dotup.sci.m4x2	pv.dotusp.sci.m4x2	pv.sdotsp.sci.m4x2	pv.sdotup.sci.m4x2	pv.sdotusp.sci.m4x2
pv.dotsp.sci.m8x2	pv.dotup.sci.m8x2	pv.dotusp.sci.m8x2	pv.sdotsp.sci.m8x2	pv.sdotup.sci.m8x2	pv.sdotusp.sci.m8x2
pv.dotsp.sci.m8x4	pv.dotup.sci.m8x4	pv.dotusp.sci.m8x4	pv.sdotsp.sci.m8x4	pv.sdotup.sci.m8x4	pv.sdotusp.sci.m8x4
pv.dotsp.sci.m16x8	pv.dotup.sci.m16x8	pv.dotusp.sci.m16x8	pv.sdotsp.sci.m16x8	pv.sdotup.sci.m16x8	pv.sdotusp.sci.m16x8
pv.dotsp.sci.m16x4	pv.dotup.sci.m16x4	pv.dotusp.sci.m16x4	pv.sdotsp.sci.m16x4	pv.sdotup.sci.m16x4	pv.sdotusp.sci.m16x4
pv.dotsp.sci.m16x2	pv.dotup.sci.m16x2	pv.dotusp.sci.m16x2	pv.sdotsp.sci.m16x2	pv.sdotup.sci.m16x2	pv.sdotusp.sci.m16x2

- More than 100 *dotp* insns needed to support all various precision permutations
  - Huge encoding space
  - High Decoding Stage complexity of the micro-architecture

How to reduce such complexity at zero performance penalty?

Mixed-Precision: Dynamic Bit-Scalable Execution

8x4 SIMD sdotp operation

RI5CY Assembly

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

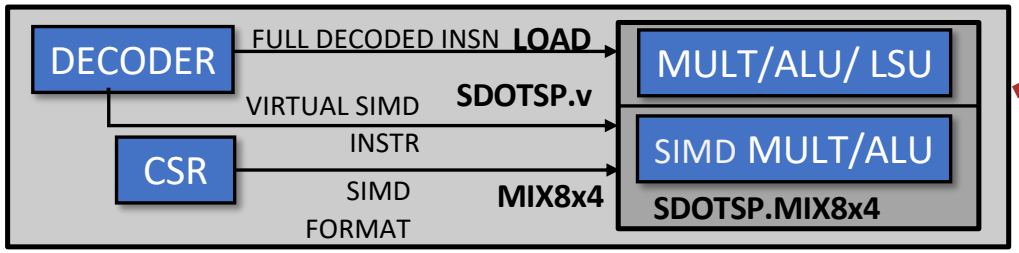
ISA extensions

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

- One instruction per format and type

Dynamic Bit-Scalable Execution

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



Virtual Instructions

pv.dotsp.v	pv.sdotsp .v
pv.dotsp.sc .v	pv.sdotsp.sc .v
pv.dotsp.sci .v	pv.sdotsp.sci .v
pv.dotup .v	pv.sdotup .v
pv.dotup.sc .v	pv.sdotup.sc .v
pv.dotup.sci .v	pv.sdotup.sci .v
pv.dotusp .v	pv.sdotusp .v
pv.dotusp.sc .v	pv.sdotusp.sc .v
pv.dotusp.sci .v	pv.sdotusp.sci .v

Mixed-Precision: Dynamic Bit-Scalable Execution

8x4 SIMD sdotp operation

RI5CY Assembly

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

ISA extensions

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

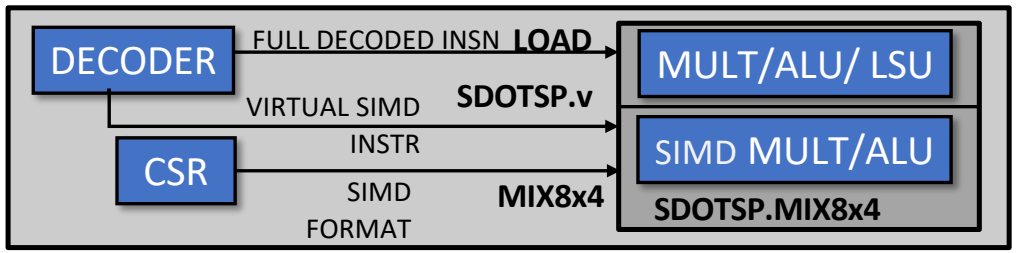
- One instruction per format and type

Dynamic Bit-Scalable Execution

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

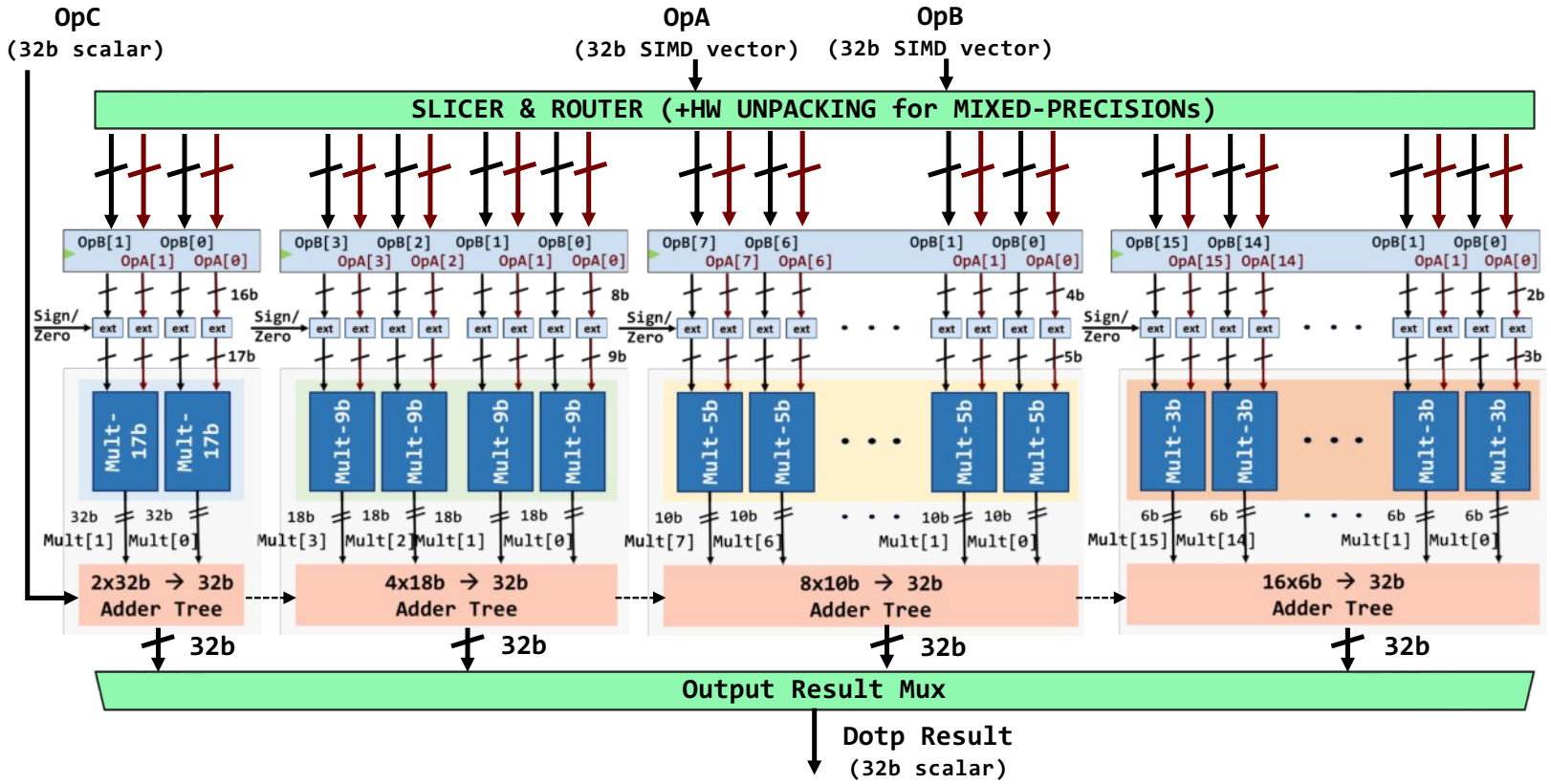
Reuse of Code

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



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

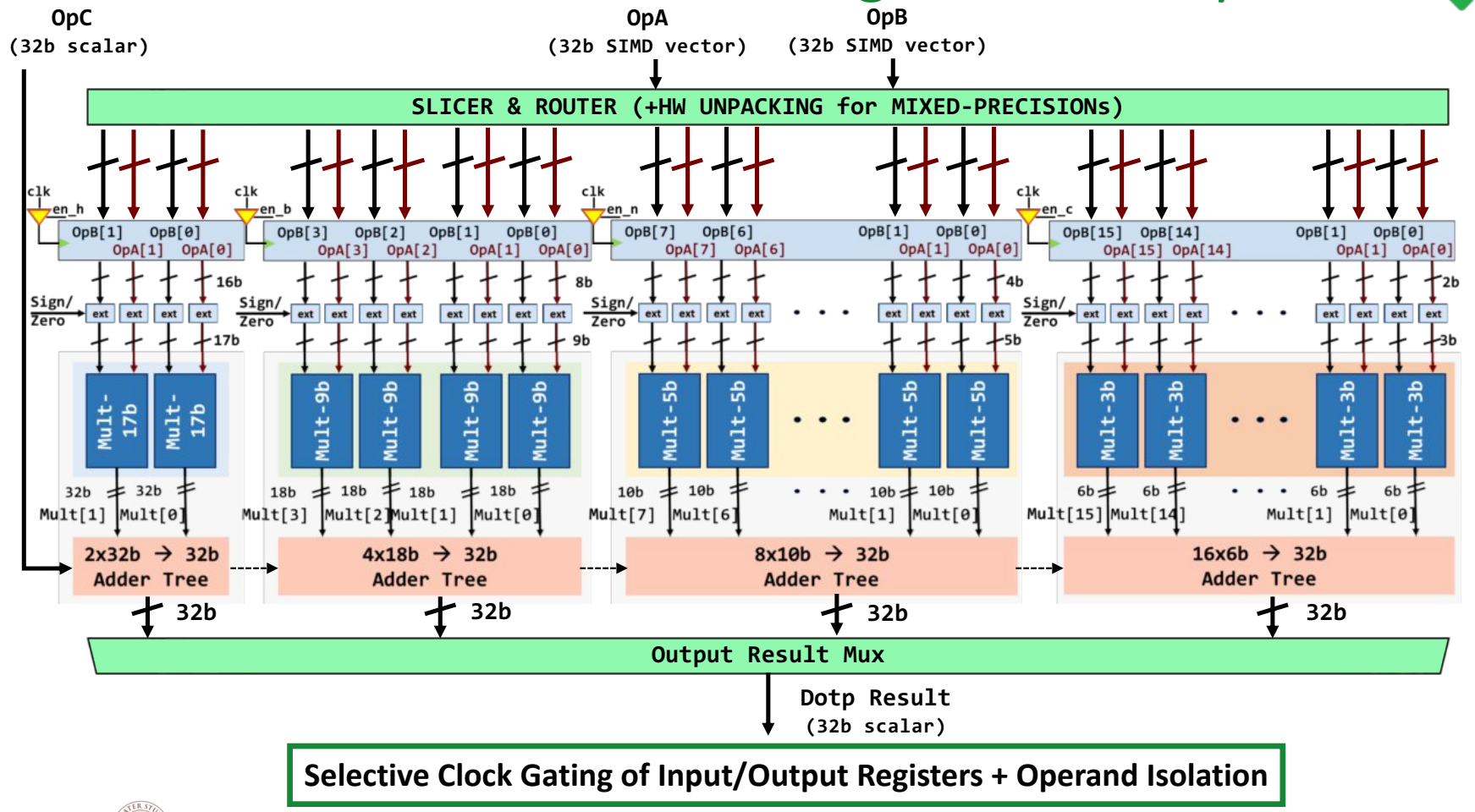
Micro-archi: The Multi-Precition Integer SIMD *sdotp* Unit



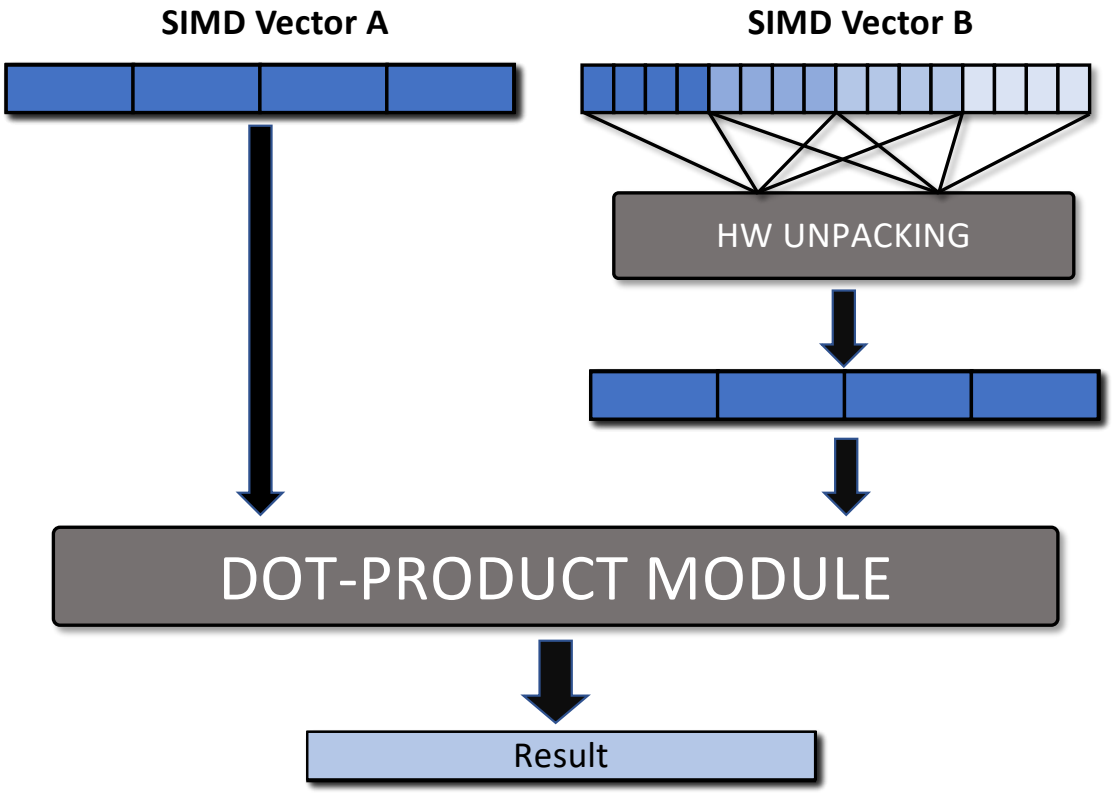
- RI5CY already features 16-b/8-b SIMD datapath
- We add different multiplier regions dedicated to each ``precision'' (4-b/2-b)
- **Pros:** do not increase the critical path of the core
- **Cons:** higher power consumption due to additional multipliers

Can we reduce switching activities of unused "regions"?

Micro-archi: The Multi-Precition Integer SIMD *sdotp* Unit



Micro-archi: Mixed-Precision Multipliers

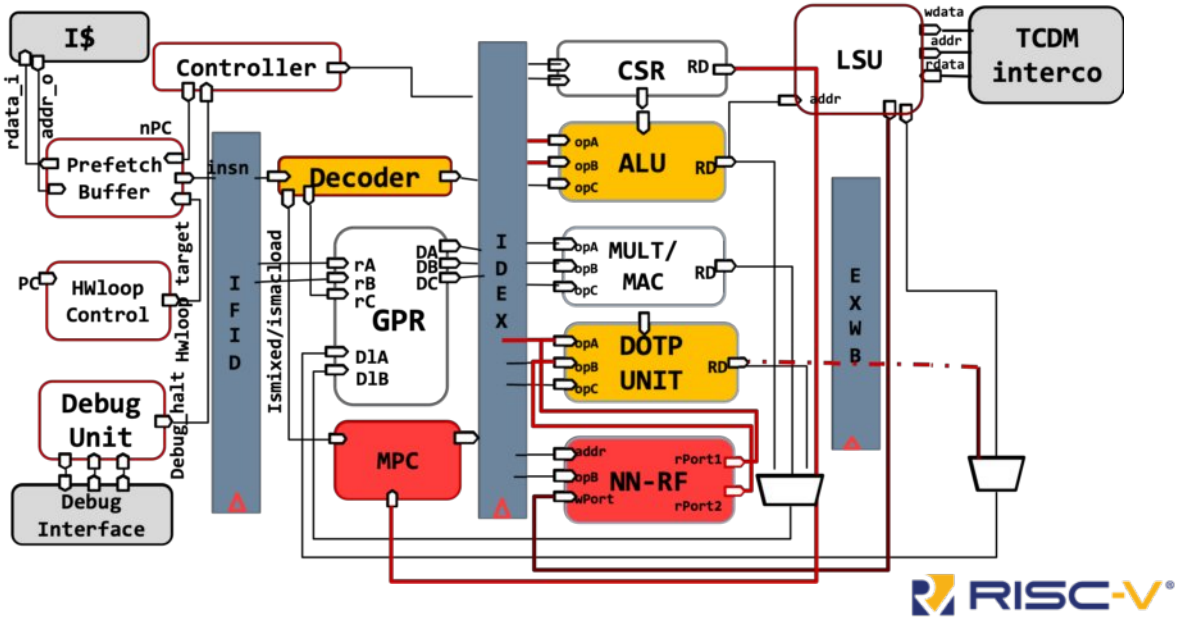


- Mixed-Precision operations require a controller: selection of the correct sub-word of the lowest precision operand (Vector B) to be used in current SIMD op.

- The controller is programmed by control status registers (CSRs).

Extended RI5CY uArchi to support XpulpNN

4-stage, single in-order issue pipeline



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

Sum-up: DNN Acceleration Through ISA Extensions

8-bit Convolution Kernel

RV32IMC

N

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

RV32IMCXpulp [DSP]

N/4

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

HW Loop

LD/ST with post increment

8-bit SIMD sdotp

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

RV32IMCXpulpNN [AI]

N/4

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

8-bit SIMD Fused MAC-Load

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

- **Power ⇧ (slightly); Latency ⇩⇩⇩(>10x) on QNN kernels ⇛ Energy ⇩⇩**
- Parallelization of the workload (multi-core) for high throughput

How Much Does it Cost?

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

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

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

Summary

- **SW & ISA:**
  - Assembly analysis gives a clear understanding of acceleration opportunities throug ISA ext.
  - SW needs to exploit the new/custom/extended ISA to be efficient at application level
    - ..plus algorithmic-level optimizations
- **uArchitecture:**
  - Careful micro-architectural design to not introduce op. frequency regressions
  - Fine-grained clock gating reduces drastically dynamic power consumption
  - New ISA support do not jeopardize power figures of the processor on GP applications
    - Yet, it drastically improves the energy-efficiency on AI kernels
- **Next:**
  - All nice, but a core alone cannot do much..
  - You need an energy-efficient system as well!
    - Careful design of the memory hierarchy, interconnects, and more..

ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

HW/SW Co-design of Energy Efficient RISC-V Edge AI Platforms

ACACES 2026, Fiuggi (Italy), 12-17 July 2026

Part 3: Parallel Ultra Low Power Cluster and Vector Lockstep Execution Modes

Angelo Garofalo angelo.garofalo@unibo.it
agarofalo@iis.ee.ethz.ch

pulp-platform.org
@pulp_platform
company/pulp-platform
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

Outline

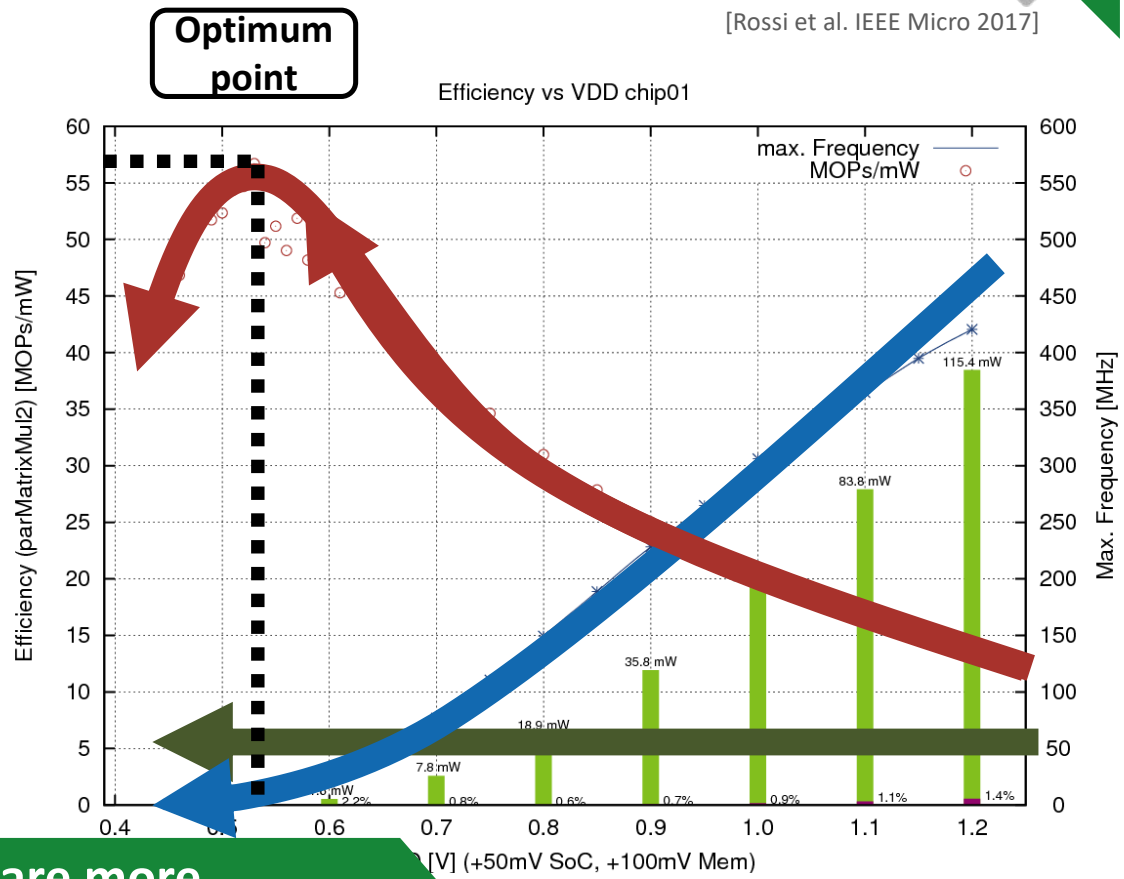
- Why multi-core clusters of tiny/simple RISC-V cores?
- PULP: Parallel Ultra Low Power Cluster.
  - Multiple Instruction Multiple Data (MIMD) approach
  - Multi-bank scratchpad memories and Low-latency interconnect
  - Low-overhead Hardware Synchronization
  - DMA for software-managed memory and double-buffering
  - MCU host domain
- AI Parallel Execution, Power analysis, and optimization solutions
  - Vector Lockstep Execution Modes (VLEM) as alternative to MIMD
- Scalability issues of the MIMD approach and alternatives: a deep study
- Summary

Near-Threshold Parallelism for Efficient TinyML Systems

[Rossi et al. IEEE Micro 2017]

- **As VDD decreases, operating speed decreases as well.**
- **However efficiency increases→ more work done per Joule**
  - Until leakage effects start to dominate
- **More units in parallel**
  - Get performance up (if you can keep them busy)
  - Energy efficiency stays high!

AI workload
High Parallelism → Multi-Core
Resilient to Low-Precision → SIMD arithmetic

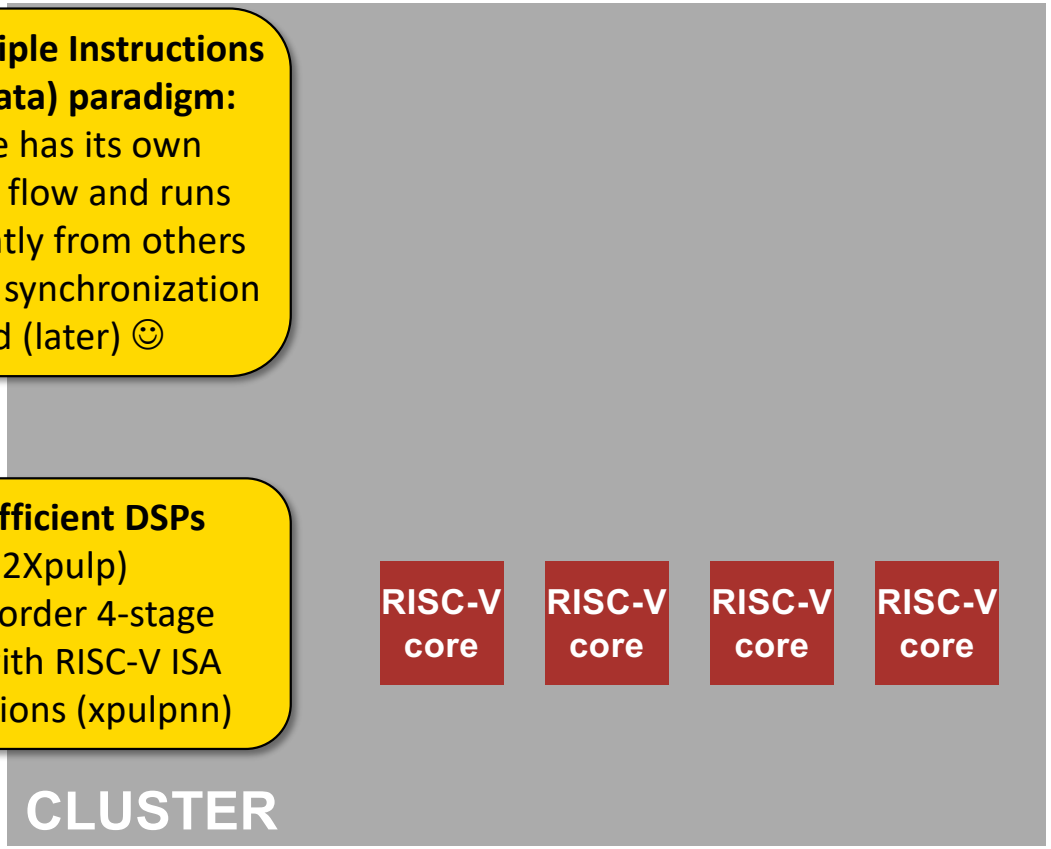


N cores running at moderate f, low Vdd are more energy efficient than a single core at N×f, high Vdd

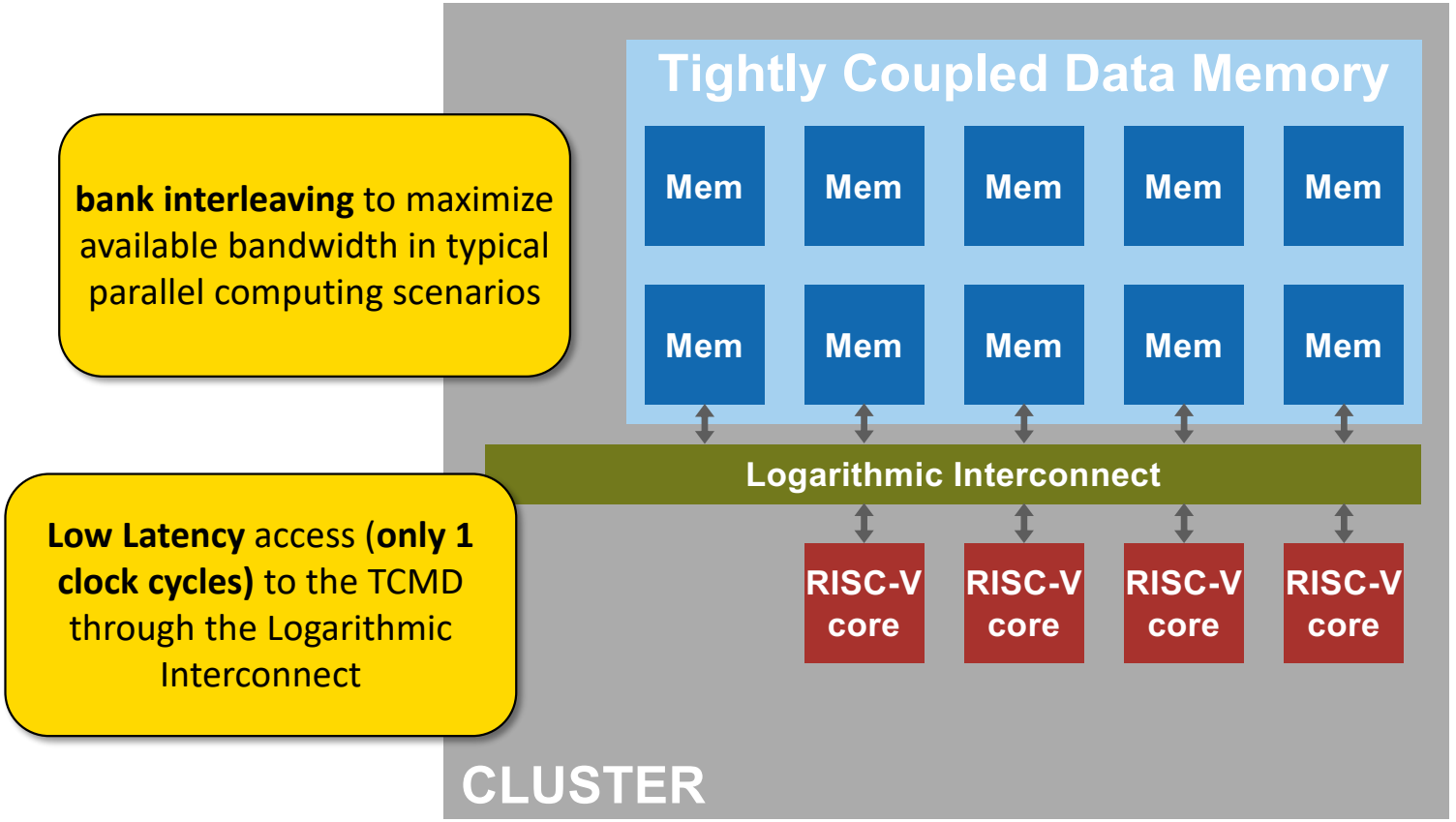
PULP Cluster: N Specialized RISC-V Cores

MIMD (Multiple Instructions Multiple Data) paradigm: each core has its own instruction flow and runs independently from others → Ofc some synchronization needed (later) ☺

1-to-16 efficient DSPs (RV32Xpulp) simple in-order 4-stage pipeline with RISC-V ISA + AI extensions (xpulpnn)



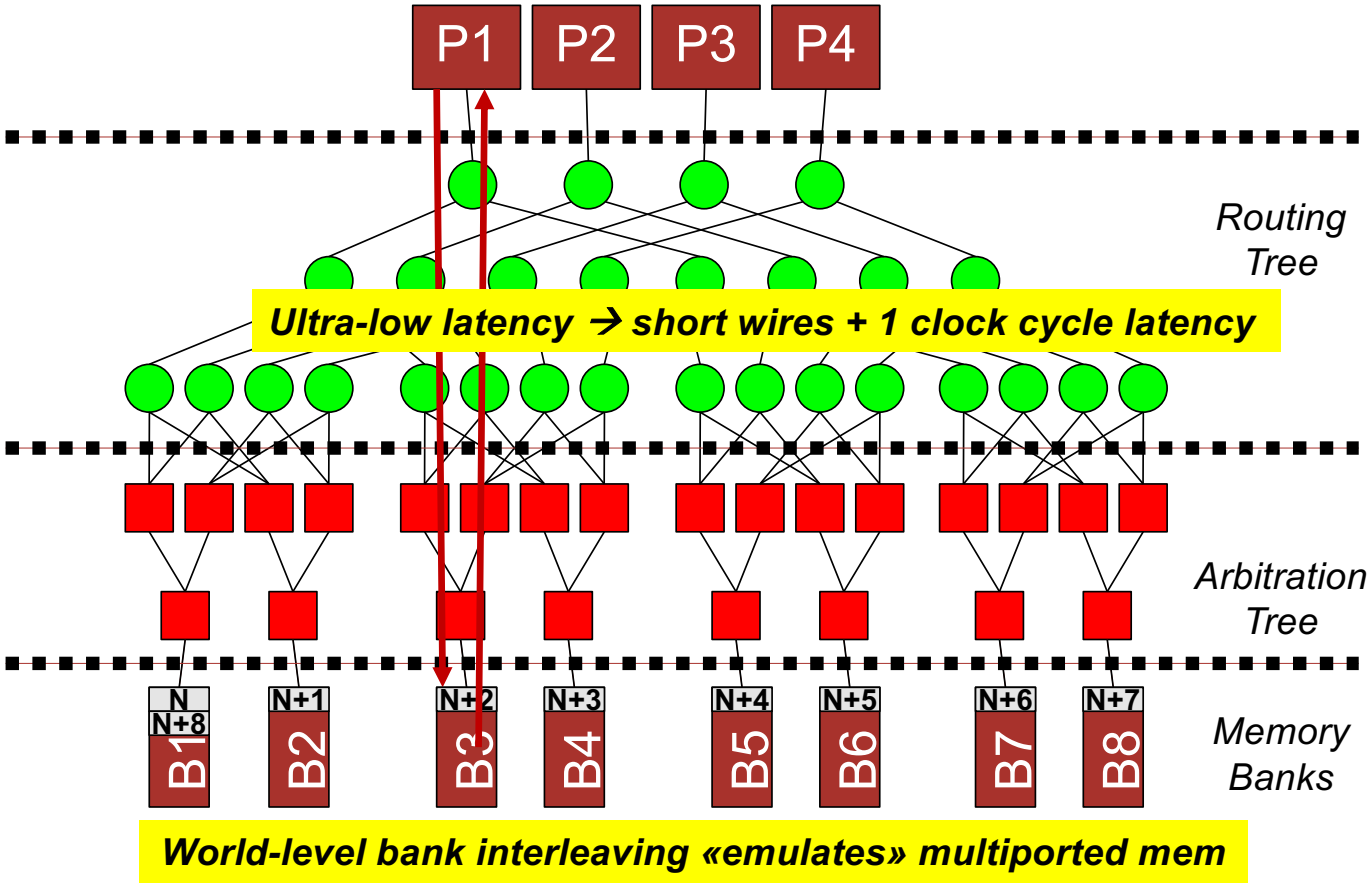
PULP Cluster: Tightly-Coupled Data Memory (TCDM)



bank interleaving to maximize available bandwidth in typical parallel computing scenarios

Low Latency access (**only 1 clock cycles)** to the TCMD through the Logarithmic Interconnect

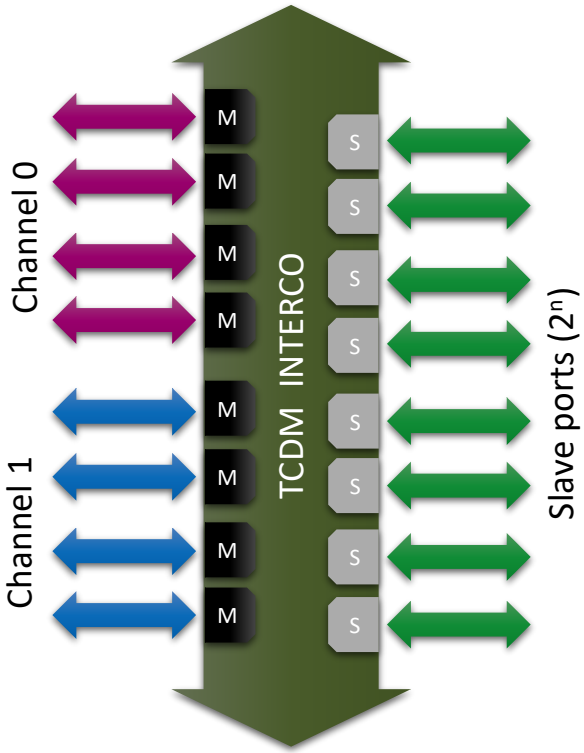
PULP Cluster: The Logarithmic Interconnect



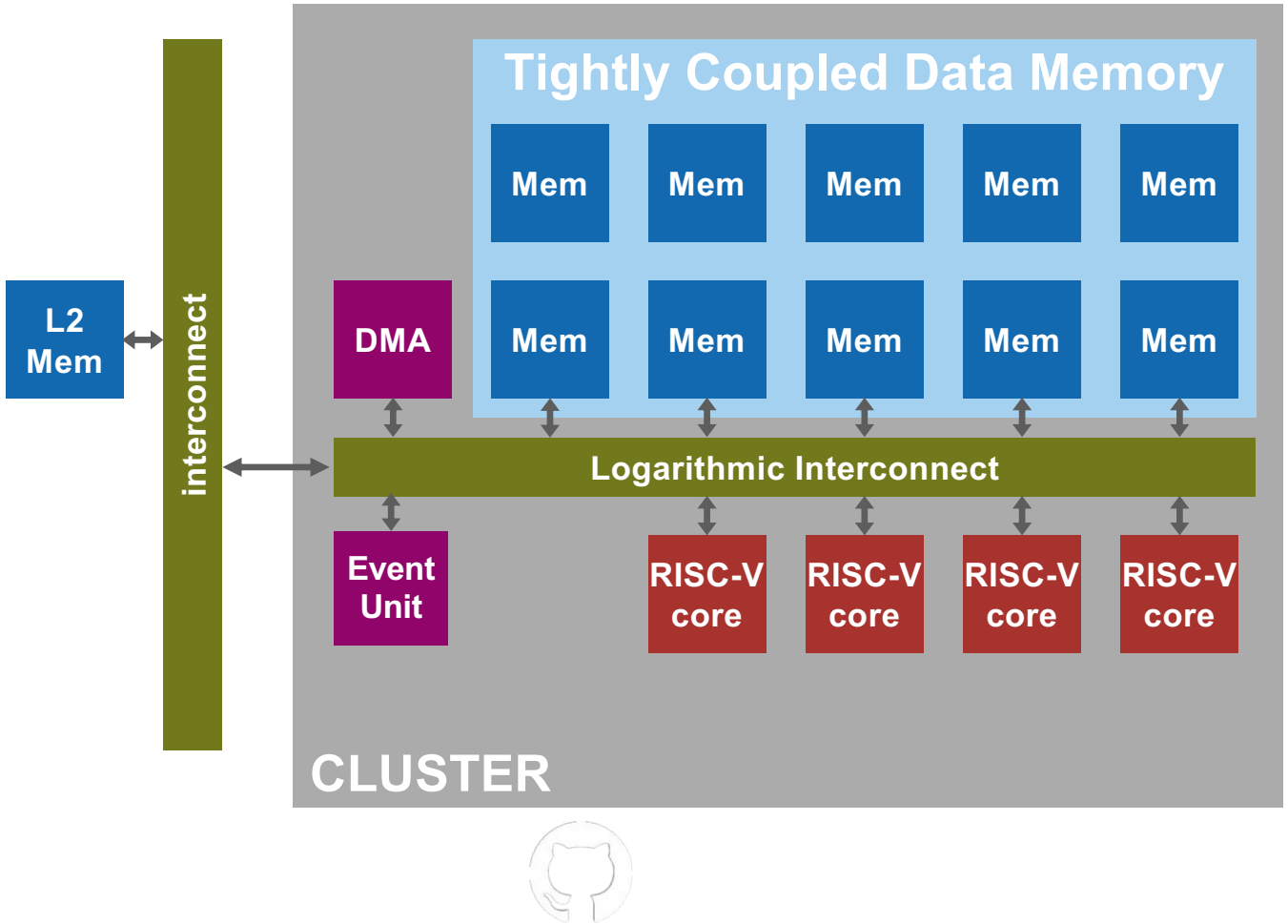
A. Rahimi, I. Loi, M. R. Kakoee and L. Benini, "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters," *2011 Design, Automation & Test in Europe*, 2011, pp. 1-6.

The Logarithmic Interconnect

- **Synchronous low-latency interconnect based on binary trees**
- **Distributed arbitration (round robin)**
  - Rather than centralized..
- **Combinational handshake (same clk cyc)**
  - A mem request from the core granted in the same clk cycle if no contention
- **Deterministic access latency to get data (1 clk cycle)**
  - Data available in the processor 1 cyc after gnt was given
- **Word-level interleaving supported**
  - Emulates multi-ported memories
    - High bandwidth, reduced conflicts
  - Bank conflicts can be alleviated → banking factor (BF=2, 4,..)

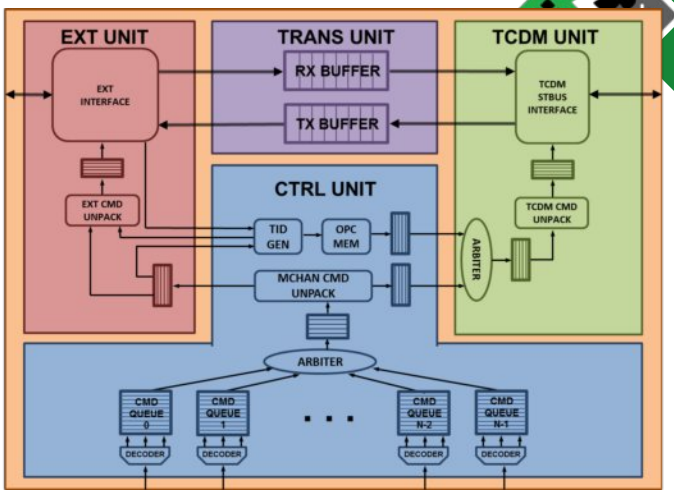


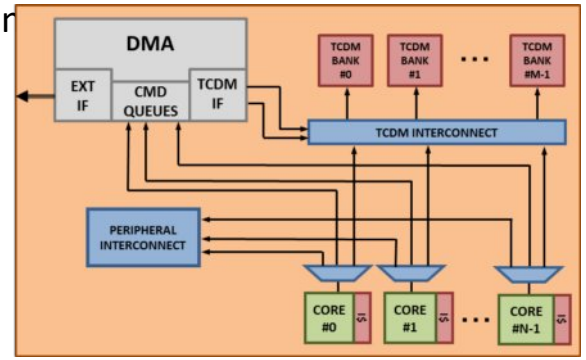
DMA for data transfers from/to L2



PULP MCHAN DMA Engine

- **A DMA engine optimized for integration in tightly coupled processor clusters**
  - Dedicated, per-core non blocking programming channels
  - Ultra **low latency programming** (~10 cycles)
  - **Small footprint** (30Kgates) : avoid usage of large local FIFOs by forwarding data directly to TCDM (no store and forward)
  - Support for **multiple outstanding transactions**
  - **Parallel RX/TX** channels allow achieving full bandwidth for concurrent load and store operations
- **Configurable parameters:**
  - # of core channels
  - Size of command queues
  - Size of RX/TX buffer
  - # of outstanding transactions





Integration in a PULP cluster

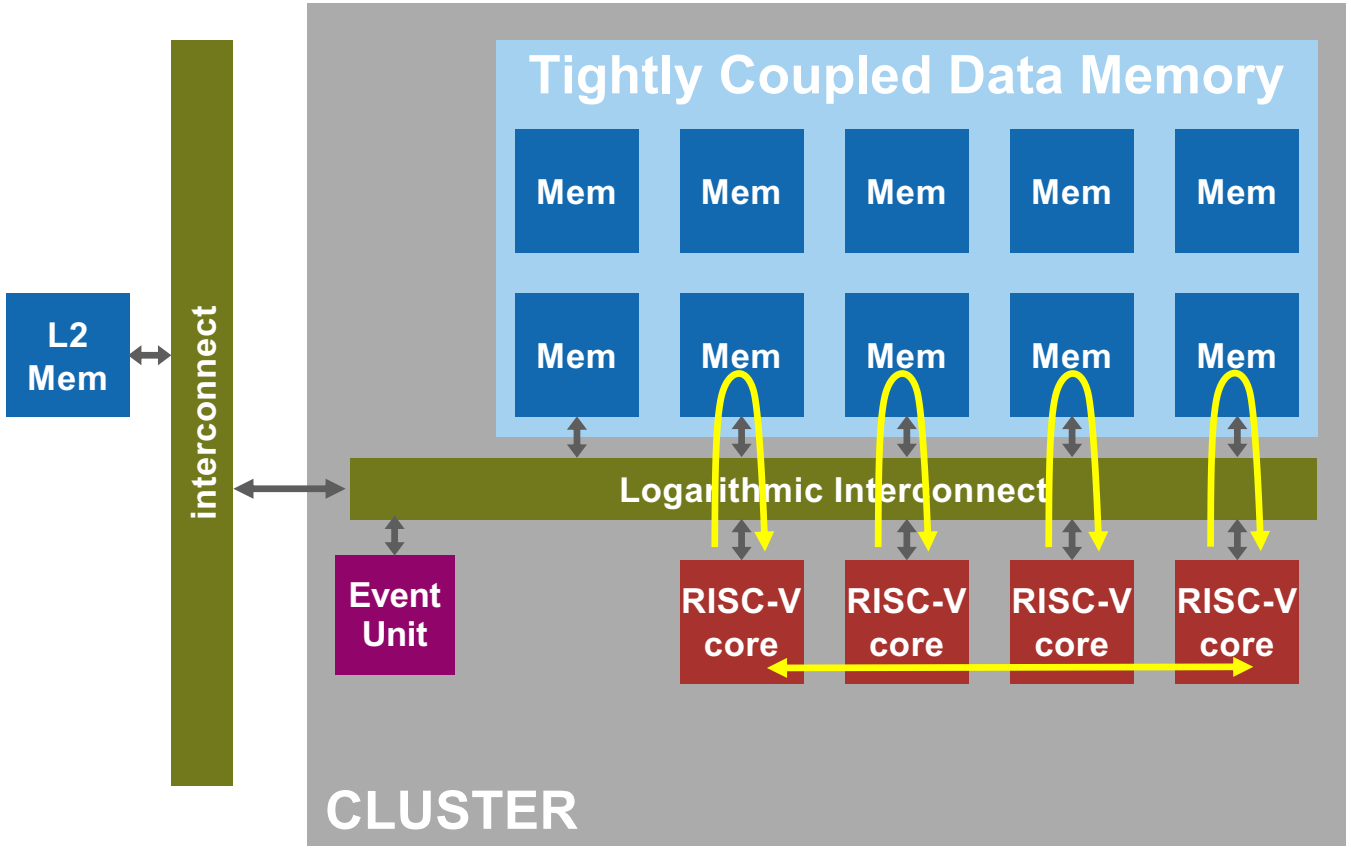
D. Rossi, I. Loi, G. Haugou, and L. Benini. 2014. Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters. In Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14).

ETH zürich

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

PULP Cluster: Fast Synchronization in HW

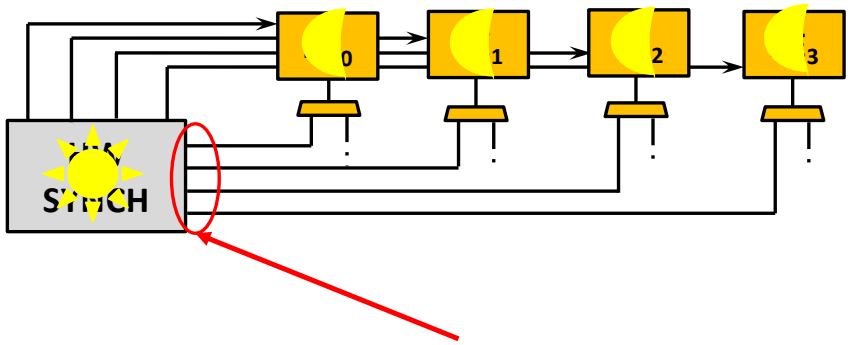
- Fast and low-energy thread dispatching and synchronization
- **Automatic clock gating** of PEs waiting on a barrier
- **Event-based computation** (between DMA and RISC-V cores)



F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang and L. Benini, "Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters," in *IEEE* TPDS, vol. 32, no. 3, pp. 633-648, 1 March 2021.

PULP Cluster: Synchronization and Events

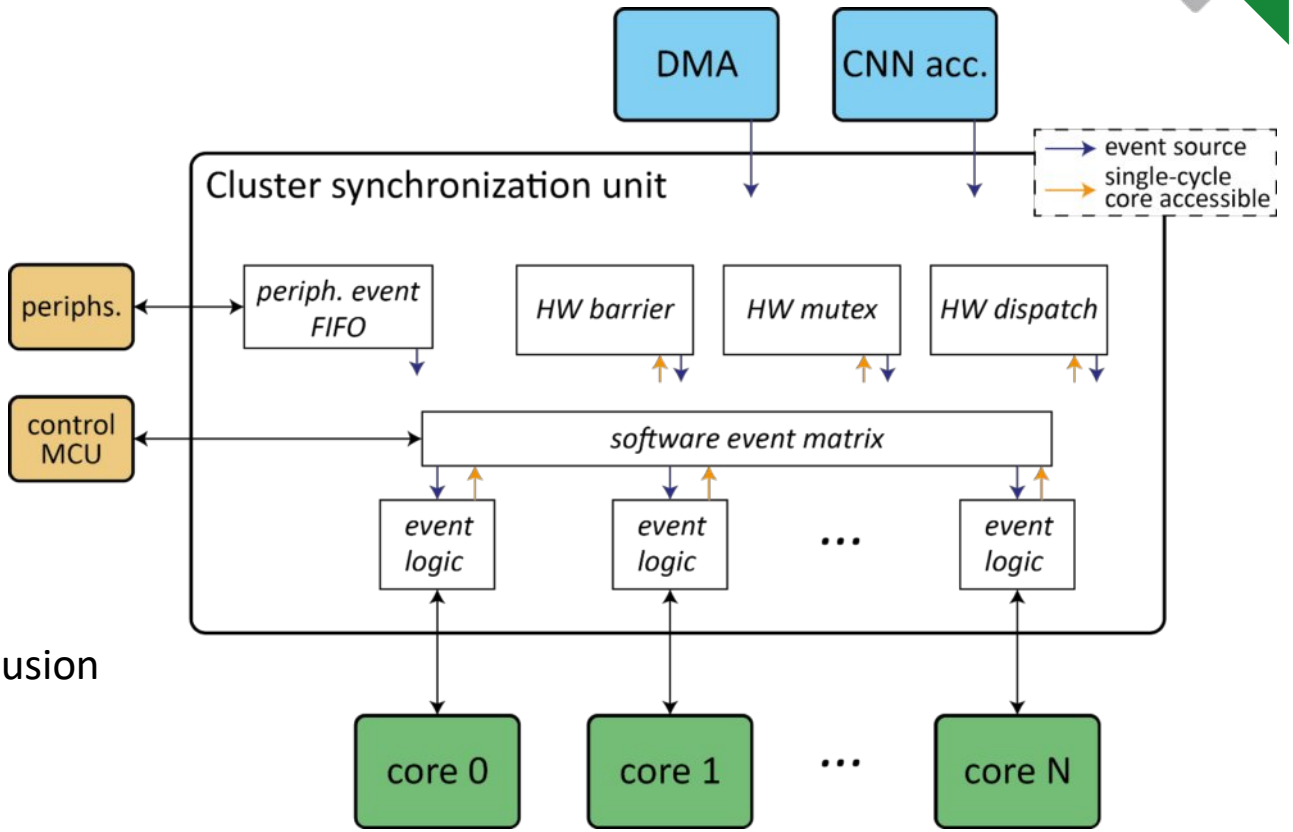
- **Baseline Synch:**
  - Work is spread among cores (fork)
  - Cores execute task in parallel (independently)
  - Test-and-set mechanisms on “join" barrier (**“polling-like approach”**)
  - A lot of busy waiting → inefficient
- **Event-based synch:**
  - Cores signals end of task to the HW Synch through dedicated ports
  - → automatically clock gated → No busy waiting!
  - On the ”join” barrier, HW synch wakes up the core and move on



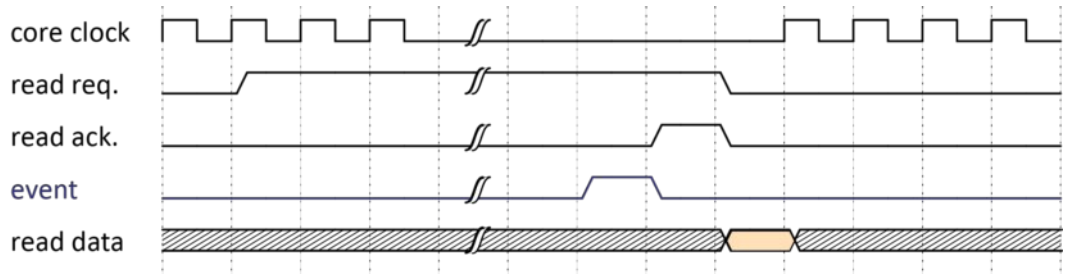
Avoid busy waiting!
Minimize sw synchro. overhead
Efficient fine-grain parallelization

PULP Cluster: The Event Unit

- **Event collector from many actors**
  - DMA
  - Cores
  - MCU
  - External peripherals
  - Accelerators (more tomorrow)
- **It handles**
  - Synchronization primitives
    - HW barriers in fork-join paradigms
  - …but also HW Mutex (mutual exclusion accesses to shared resources)
    - Useful in multithreading



PULP Cluster: Energy Efficient Event Handling



instruction turns off core clock
→ NO pipeline flush!

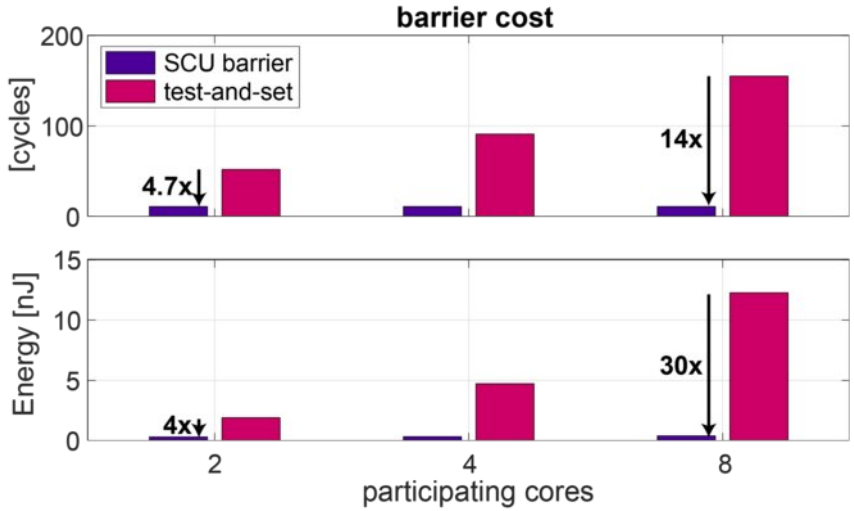
```
evt_data = evt_read(addr);
```

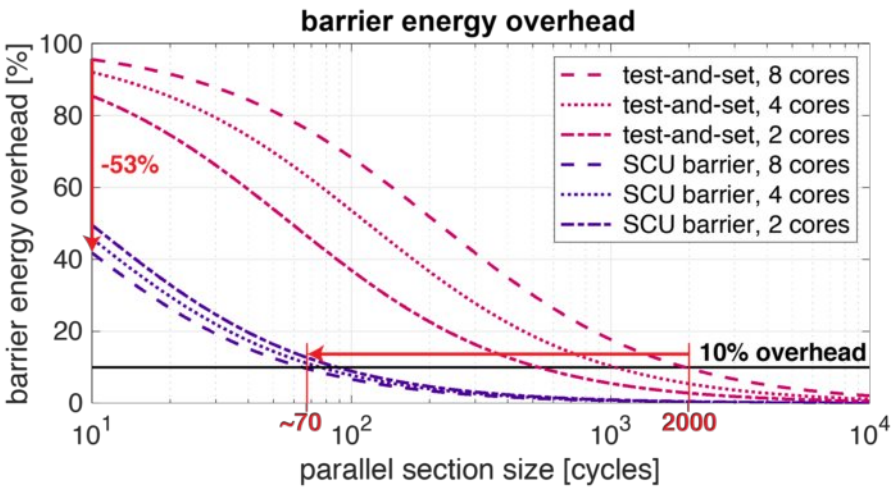
- event buffer content,
- program entry point
- mutex message
- triggering core id
- …

- trigger sw event,
- trigger barrier,
- try mutex lock,
- read entry point,
- auto buffer clear?
- …

- Single instruction rules it all:
  - address determines action
  - Data which is read provides corresponding information

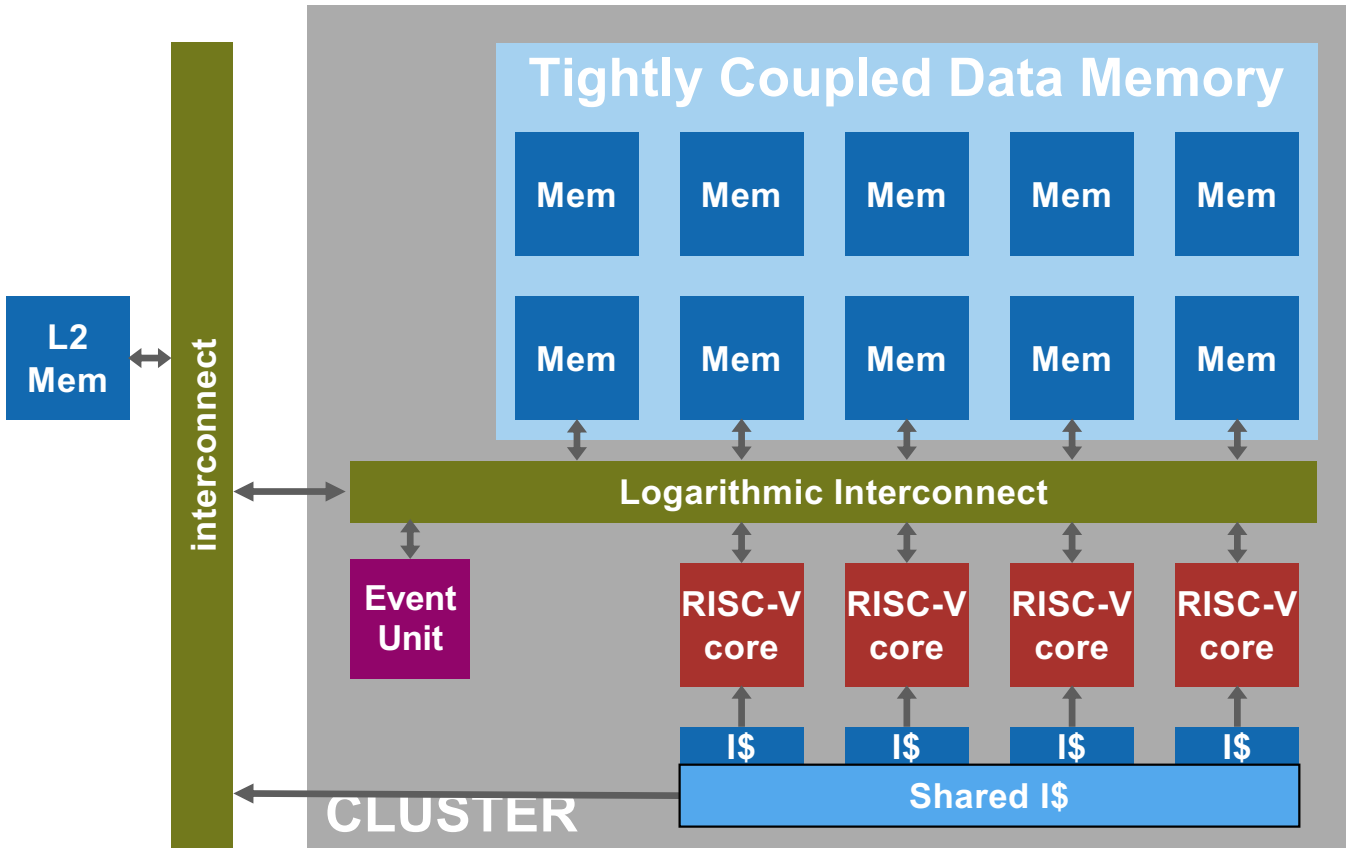
PULP Cluster: Performance of HW Barriers



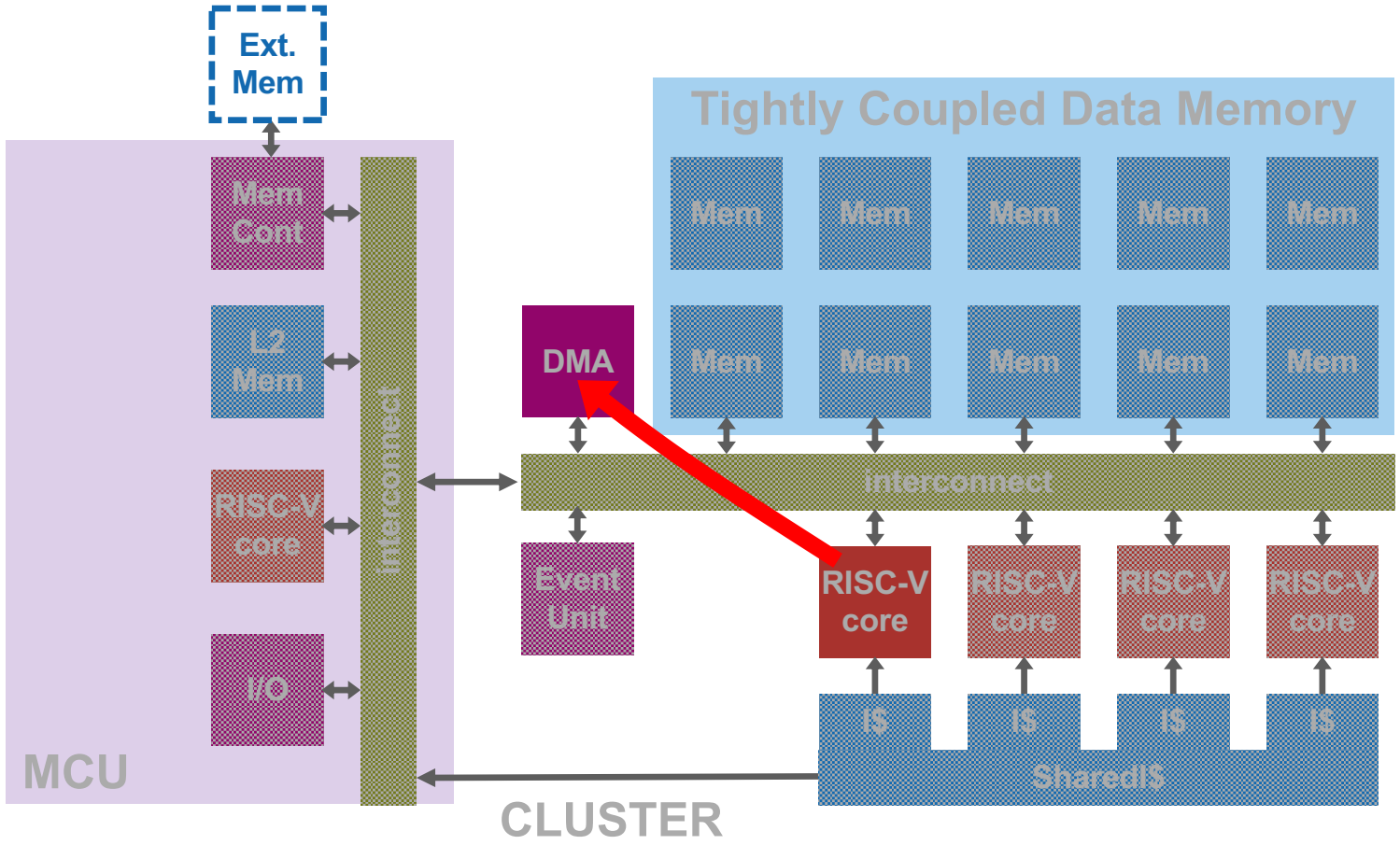


- Fully parallel access to SCU: Barrier cost constant
- Primitive energy cost: Down by up to 30x
- Minimum parallel section for 10% overhead in terms of …
  - … cycles: ~100 instead of > 1000 cycles
  - … energy: ~70 instead of > 2000 cycles

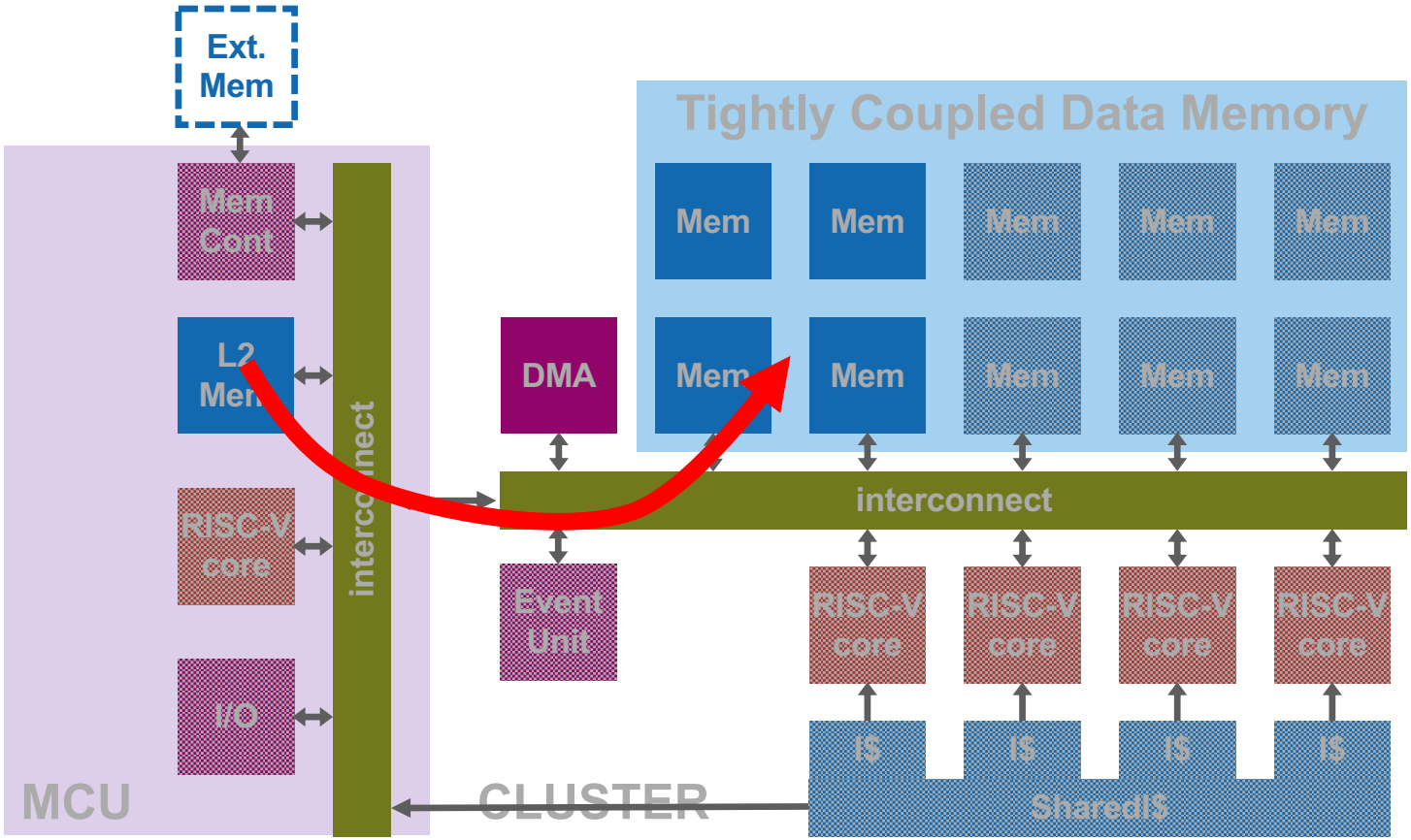
Shared instruction cache with private “loop buffer”



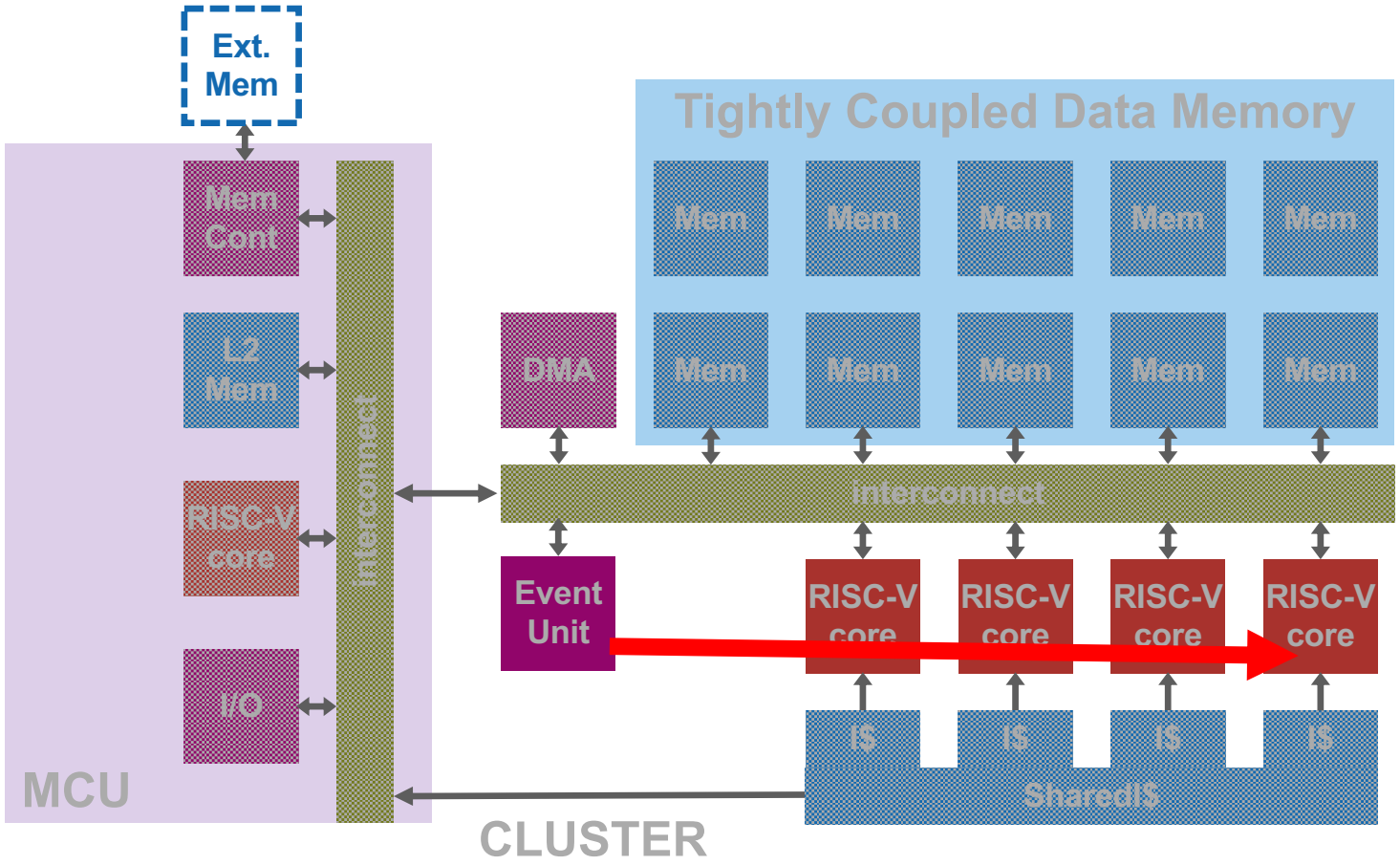
How do we work: Initiate a DMA transfer



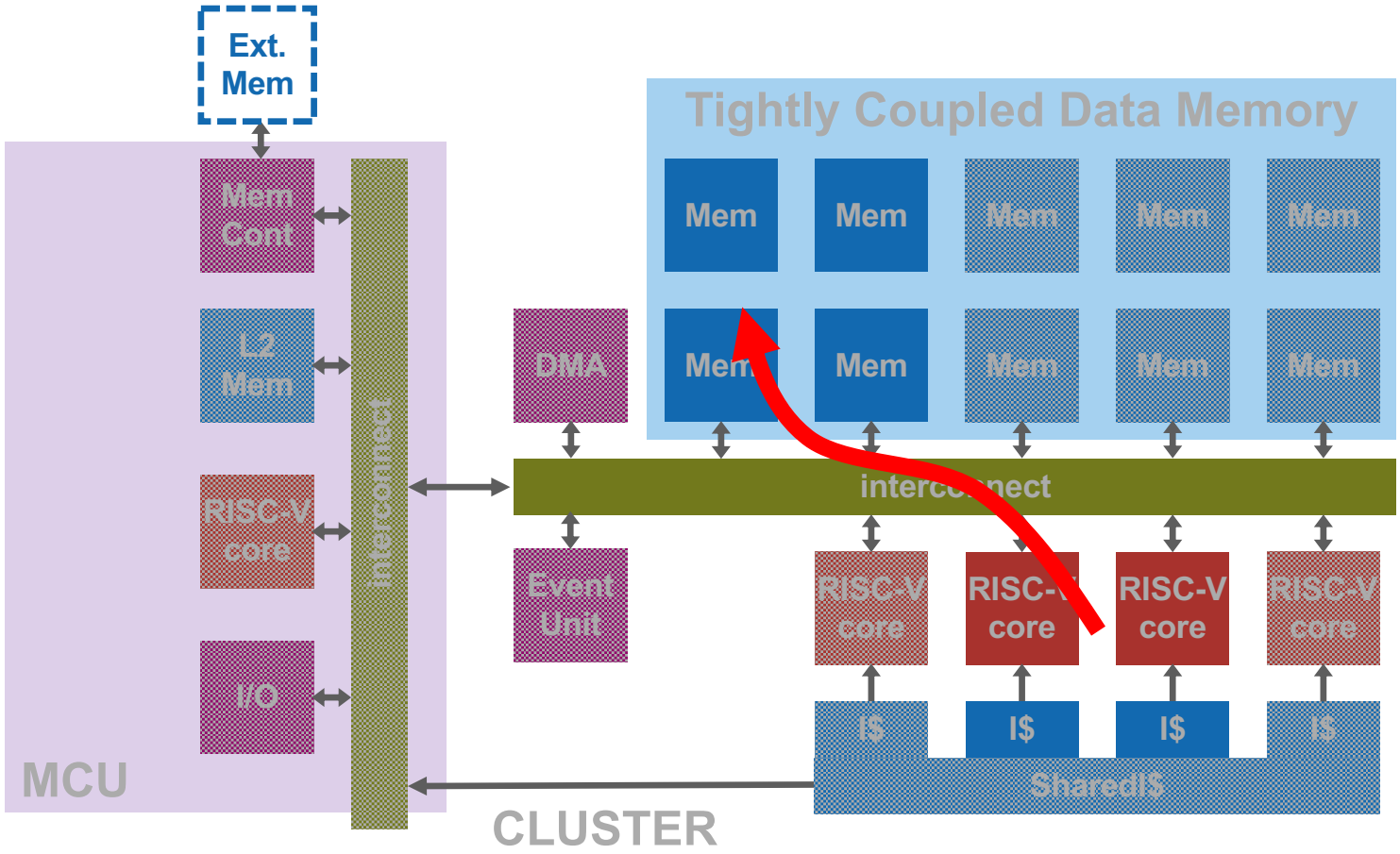
Data copied from L2 into TCDM



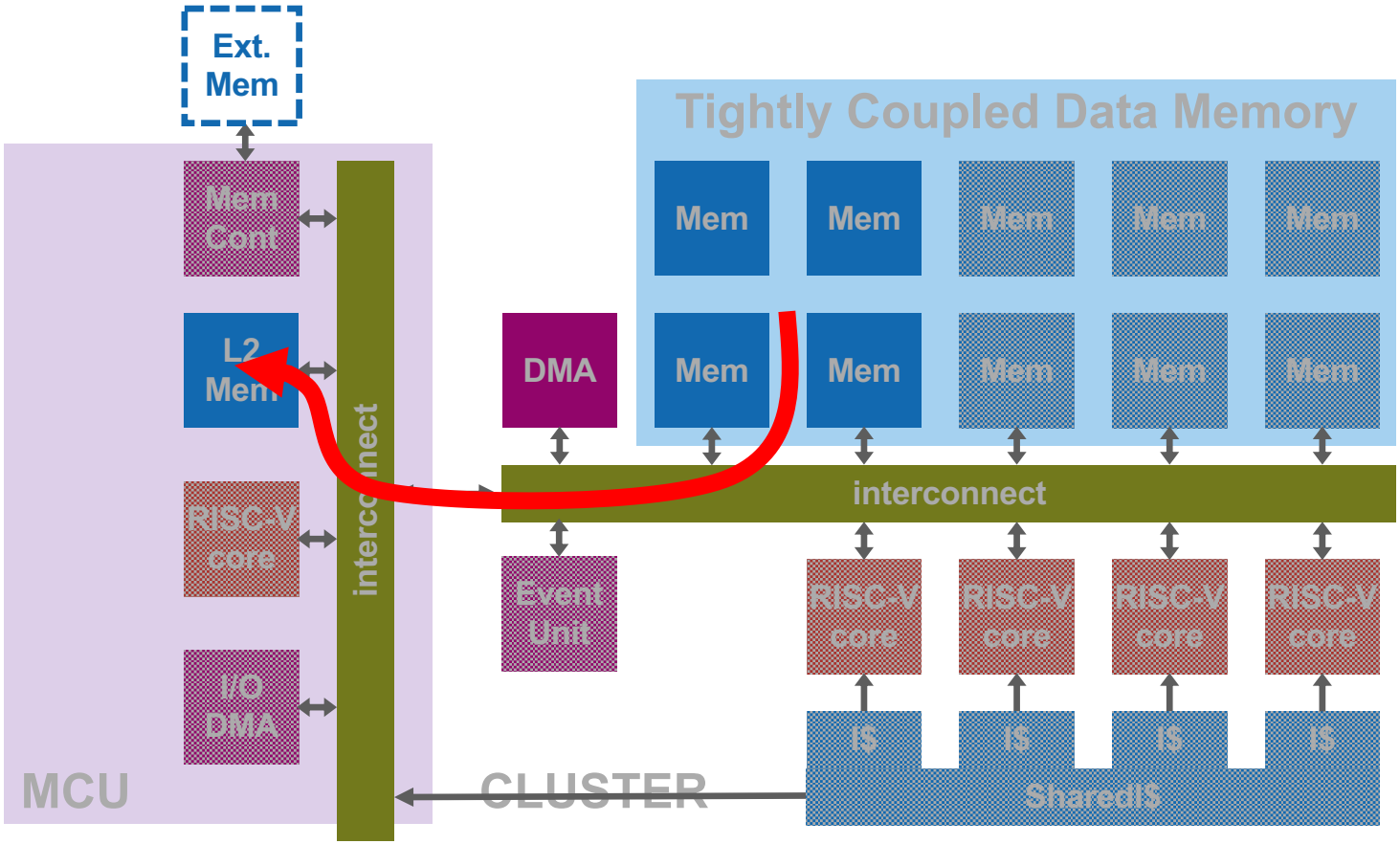
Once data is transferred, event unit notifies cores



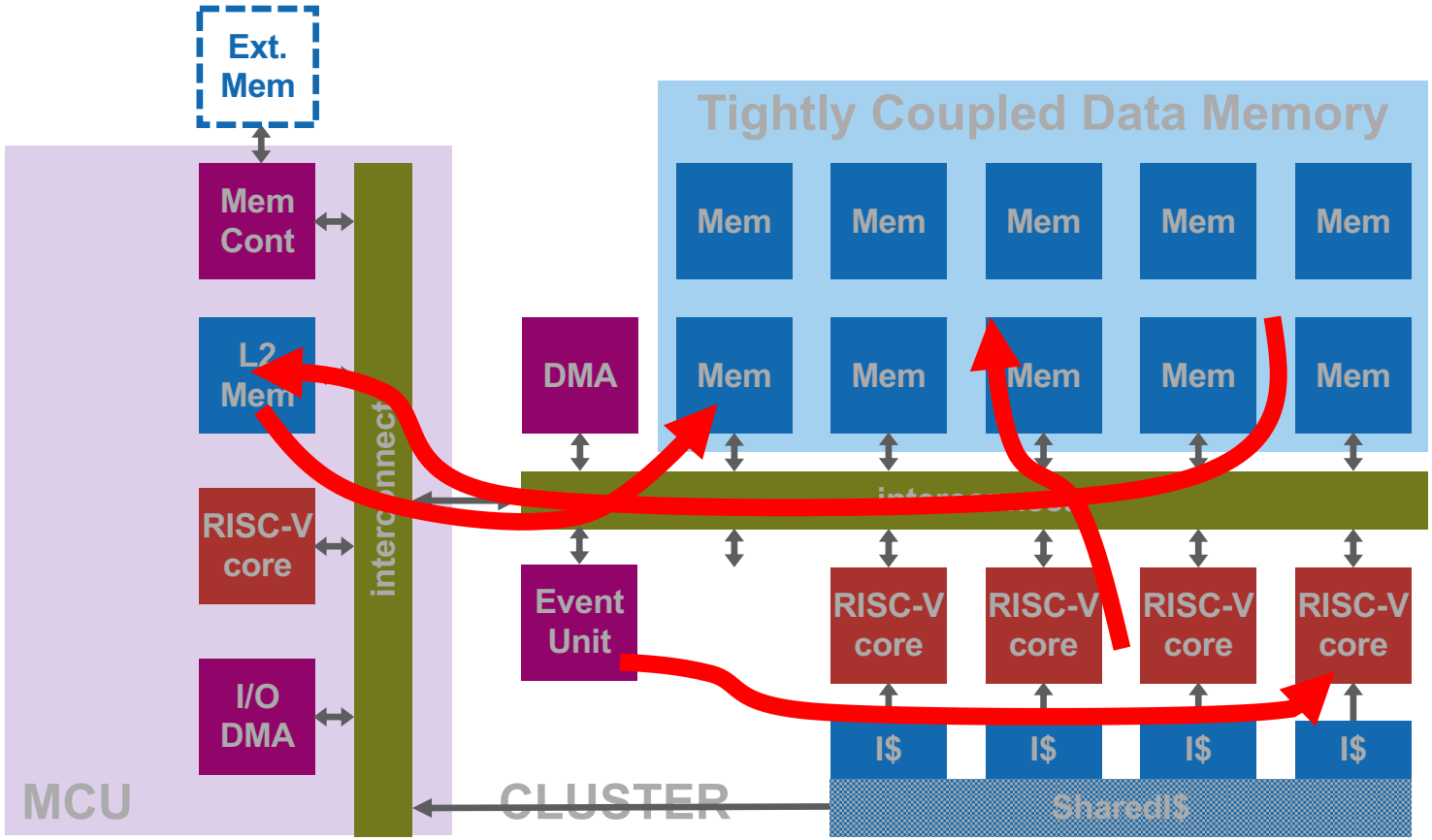
Cores can work on the data transferred



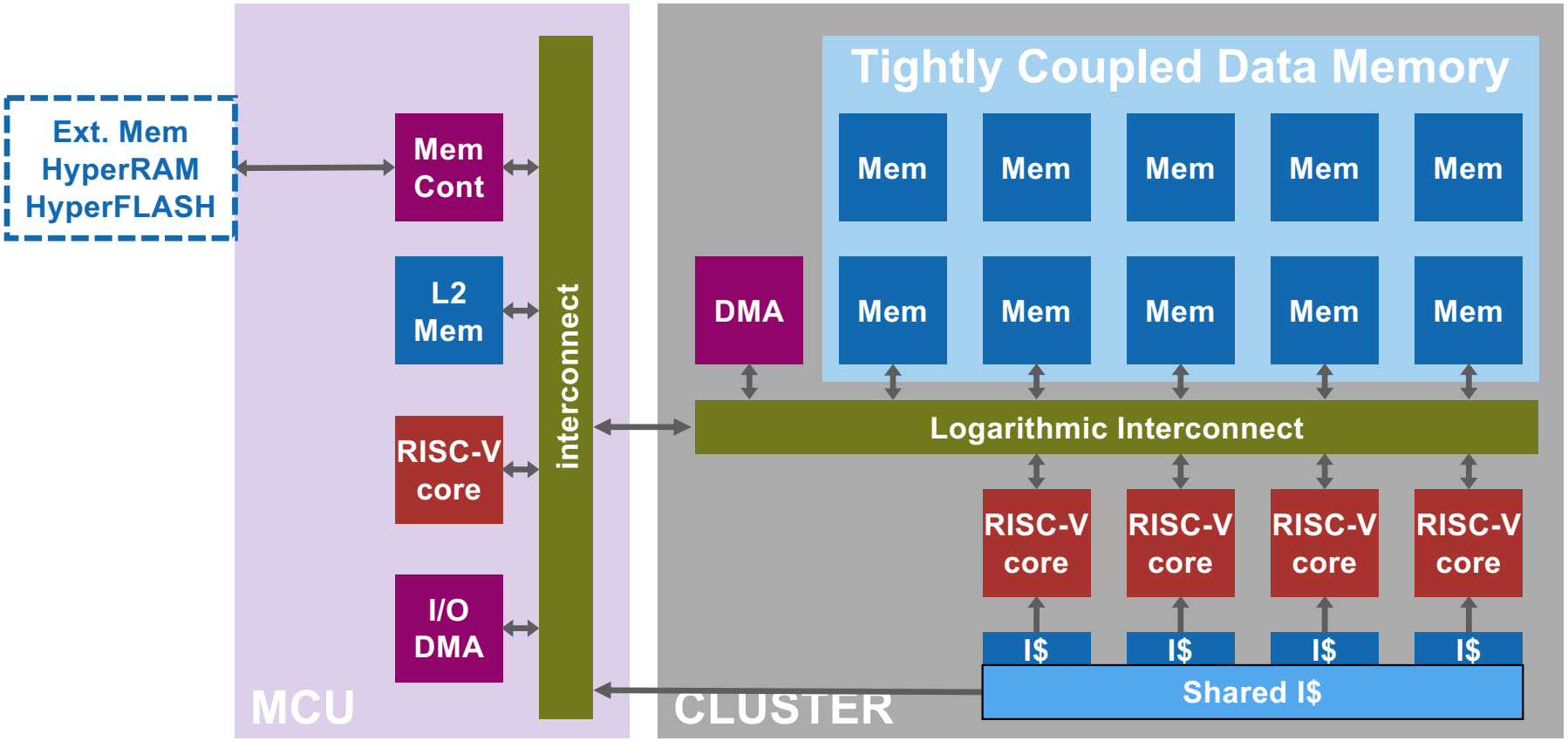
Once our work is done, DMA copies data back



During normal operation all of these occur concurrently



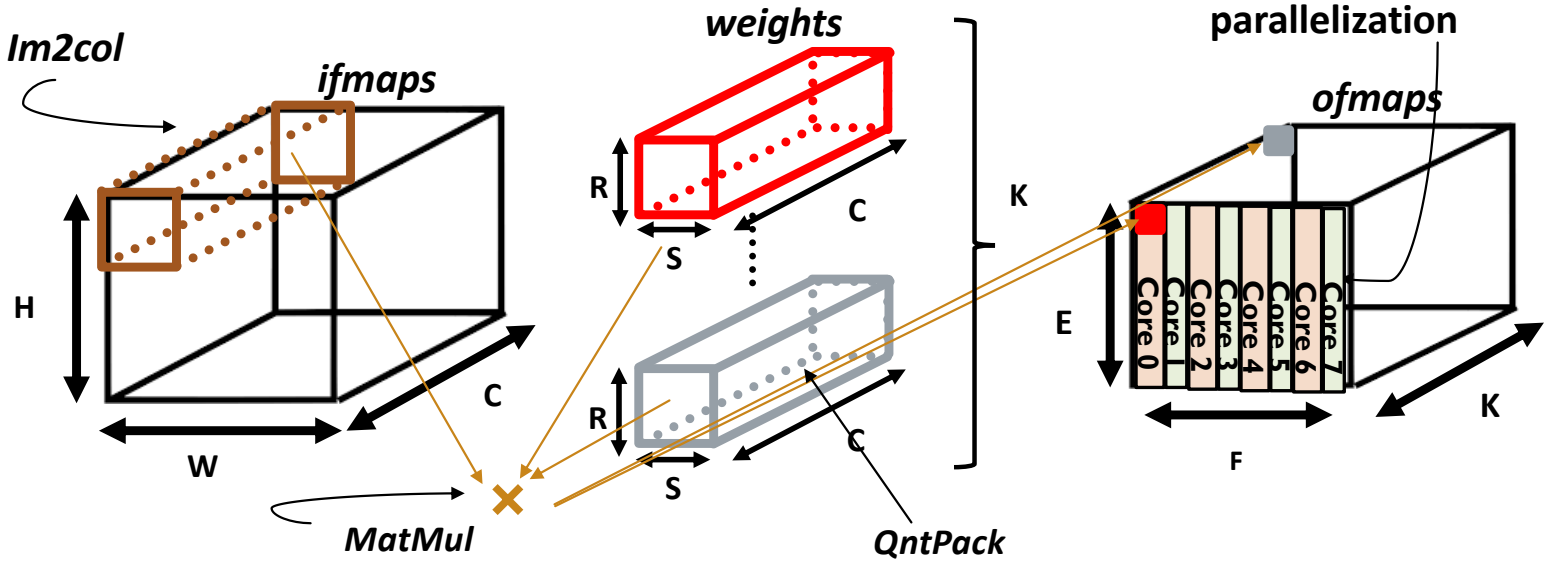
An additional I/O controller is used for Ext. L3 Memory



github.com/pulp-platform/pulp

Parallel Execution of a Conv Kernel with PULP-NN

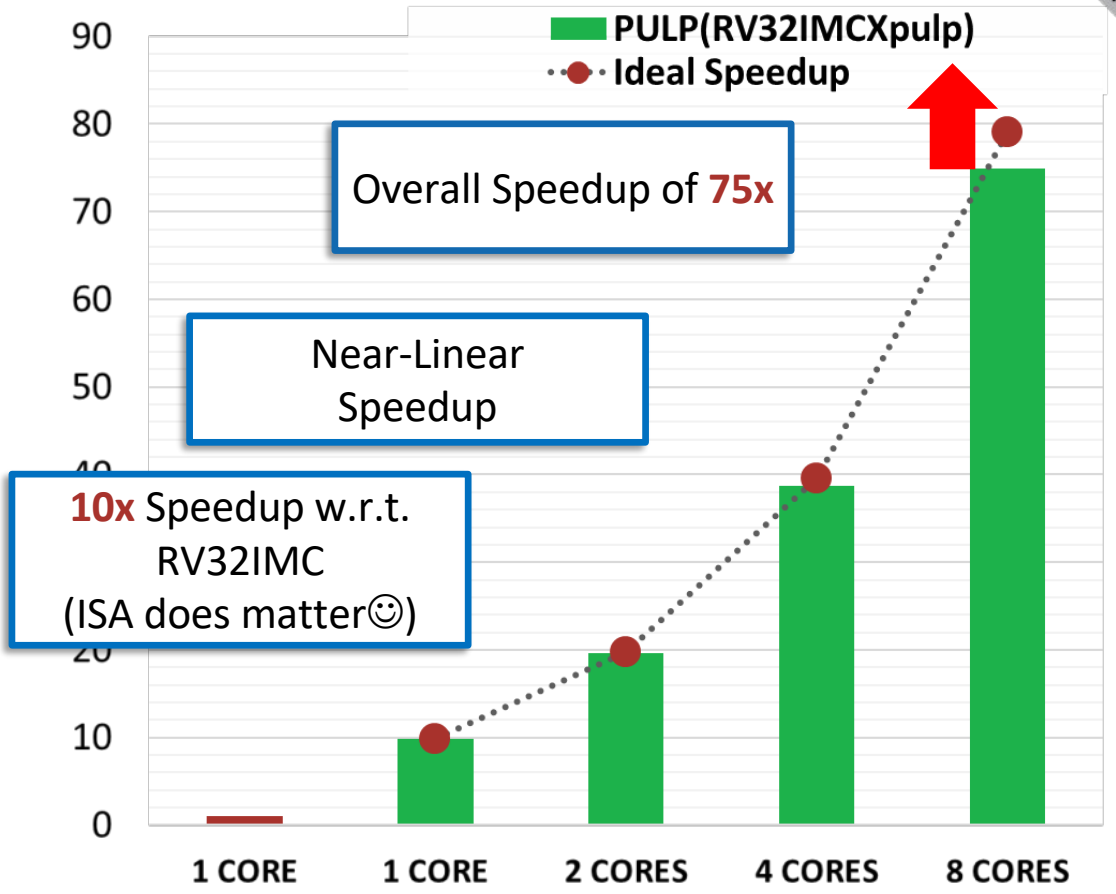
Convolution kernel



- Each core takes care of a chunk of data to be processed (one “column” of the OFM)
- Despite being a MIMD cluster, we use it as a single program multiple data (SPMD) to efficiently run GEMM/GEMV/convolutions etc → all cores execute same instructions (still independently from each other) but on different data

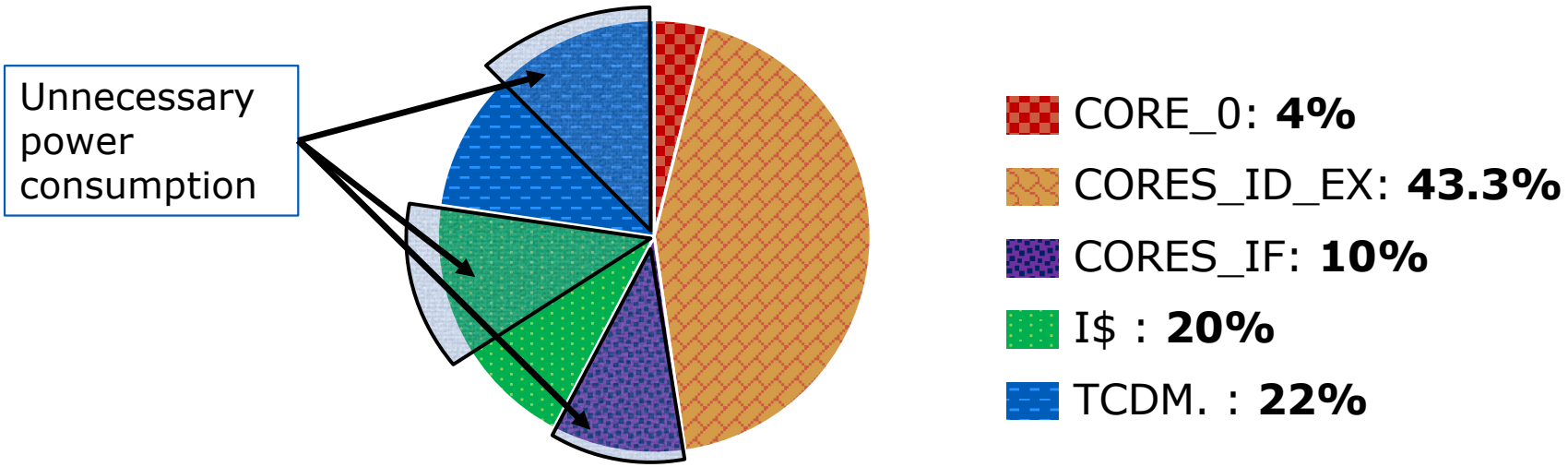
PULP on Steroids

- **8-bit convolution**
  - Optimized SW library
- **10x through xPULP**
  - Extensions bring real speedup
- **Near-linear parallel speedup**
  - Scales well for regular workloads
- **75x overall gain**
  - 2 orders of magniture boost if we use also mac-load (**122x**)
  - Even more if we execute nibble/crumb/mixed precision kernels on xpulpnn …



Power Analysis of a Parallel Convolution

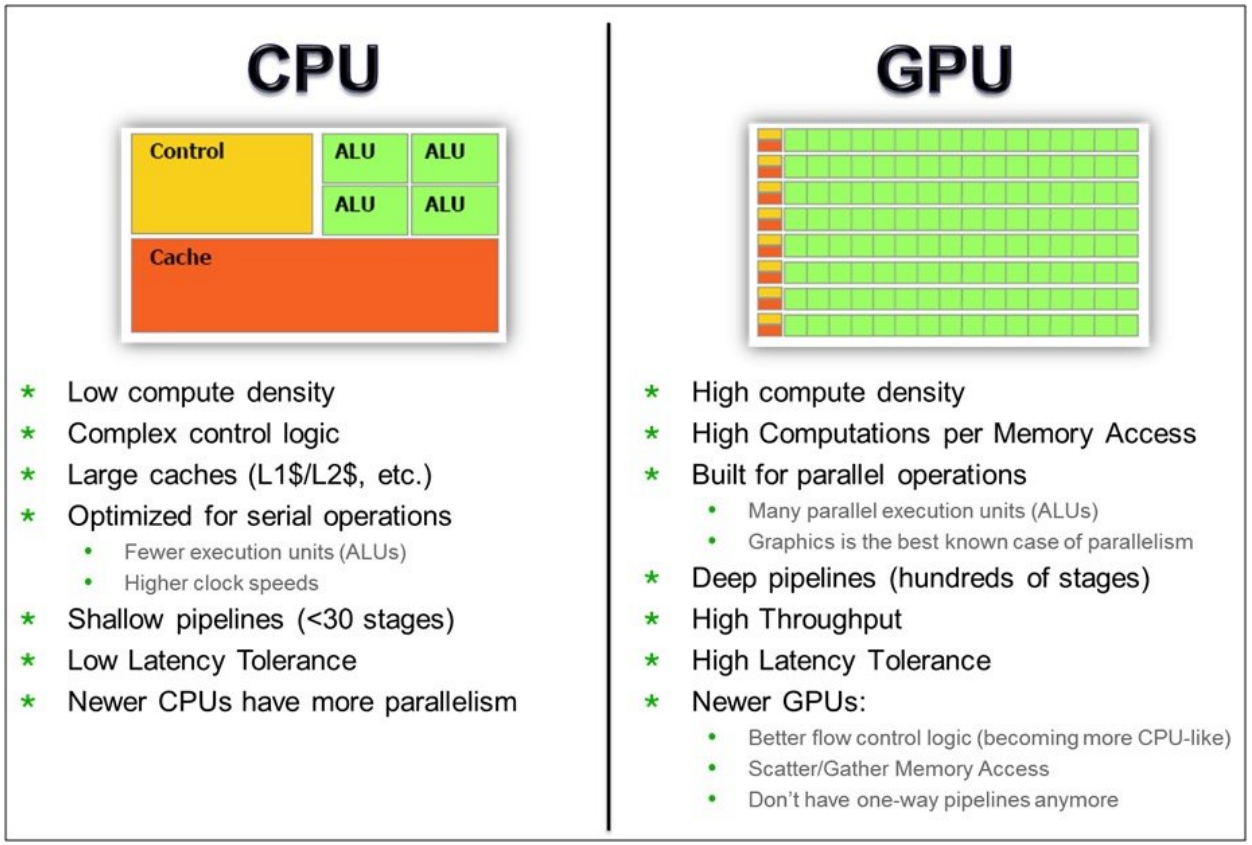
Power analysis of a parallel 8-bit x 4-bit convolution



- Reduce unnecessary power consumption (not spent in computation)
- Exploit convolution's instruction and memory data access pattern regularity

Can We Improve?

- **GP-GPU compute models might work. But…**

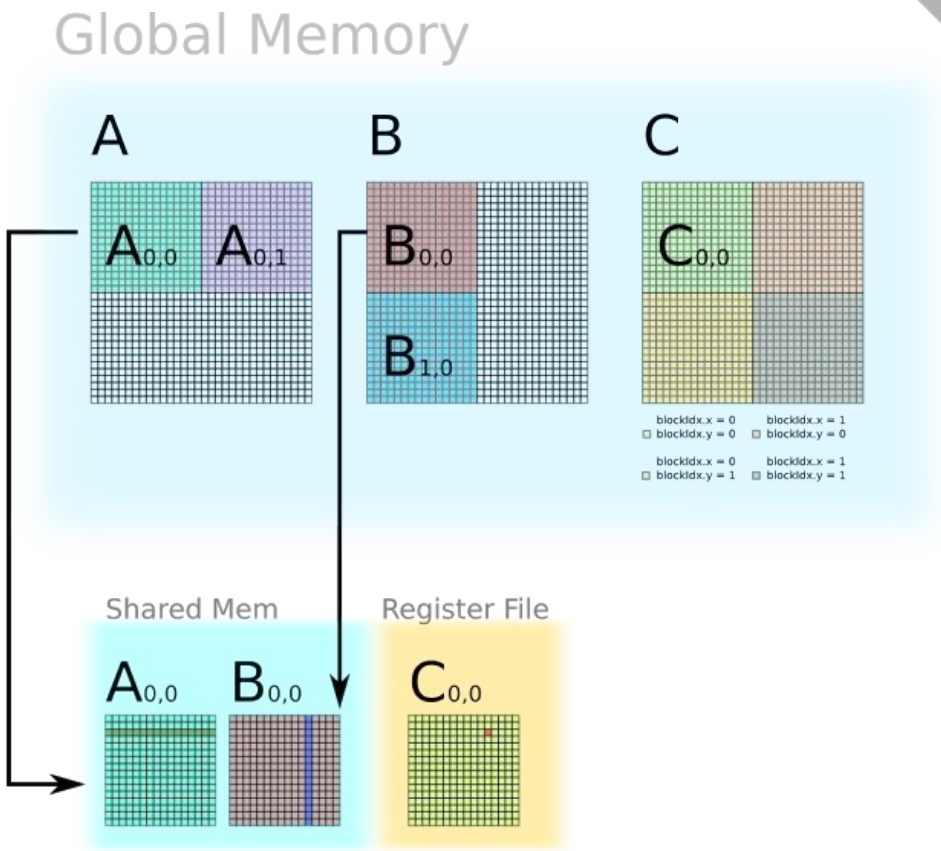


* Low compute density
* Complex control logic
* Large caches (L1$/L2$, etc.)
* Optimized for serial operations
  - Fewer execution units (ALUs)
  - Higher clock speeds
* Shallow pipelines (<30 stages)
* Low Latency Tolerance
* Newer CPUs have more parallelism

* High compute density
* High Computations per Memory Access
* Built for parallel operations
  - Many parallel execution units (ALUs)
  - Graphics is the best known case of parallelism
* Deep pipelines (hundreds of stages)
* High Throughput
* High Latency Tolerance
* Newer GPUs:
  - Better flow control logic (becoming more CPU-like)
  - Scatter/Gather Memory Access
  - Don't have one-way pipelines anymore

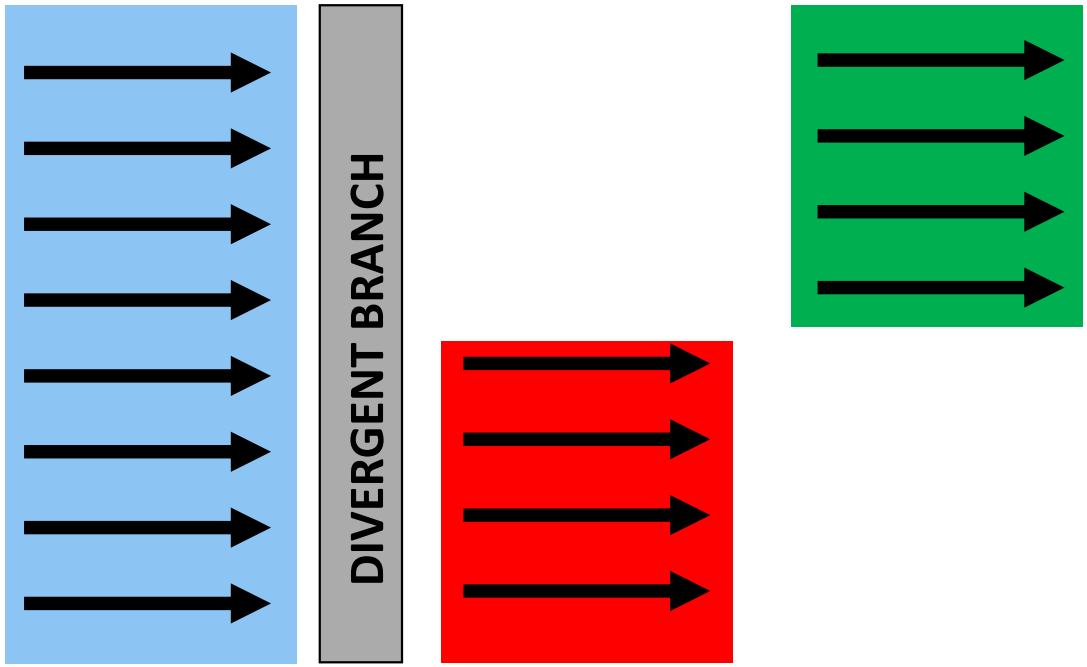
GP-GPU Model

- **Clean Hierarchical Programming and Memory Model (e.g. CUDA)**
- **Parallelism is expressed as Blocks of Threads**
  - Blocks share data through global memory
  - Threads share data through shared memory
  - **Each Thread has some private registers**
- **Exploit Single Instruction Multiple Thread (SIMT) Computational Model**



Limitations of a GP-GPU Model on Low-Power Edge Devices

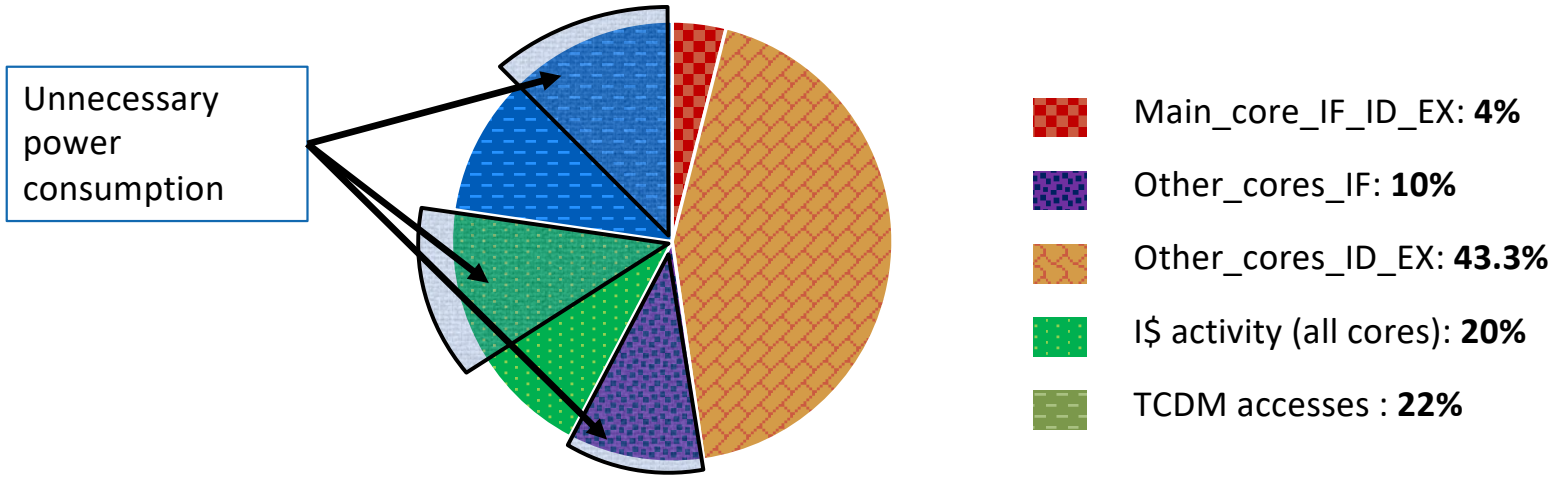
```
LOOP A

IF (ID < 4)
    EXEC X
ELSE
    EXEC Y
```



Divergent Threads Execute Sequentially!!!

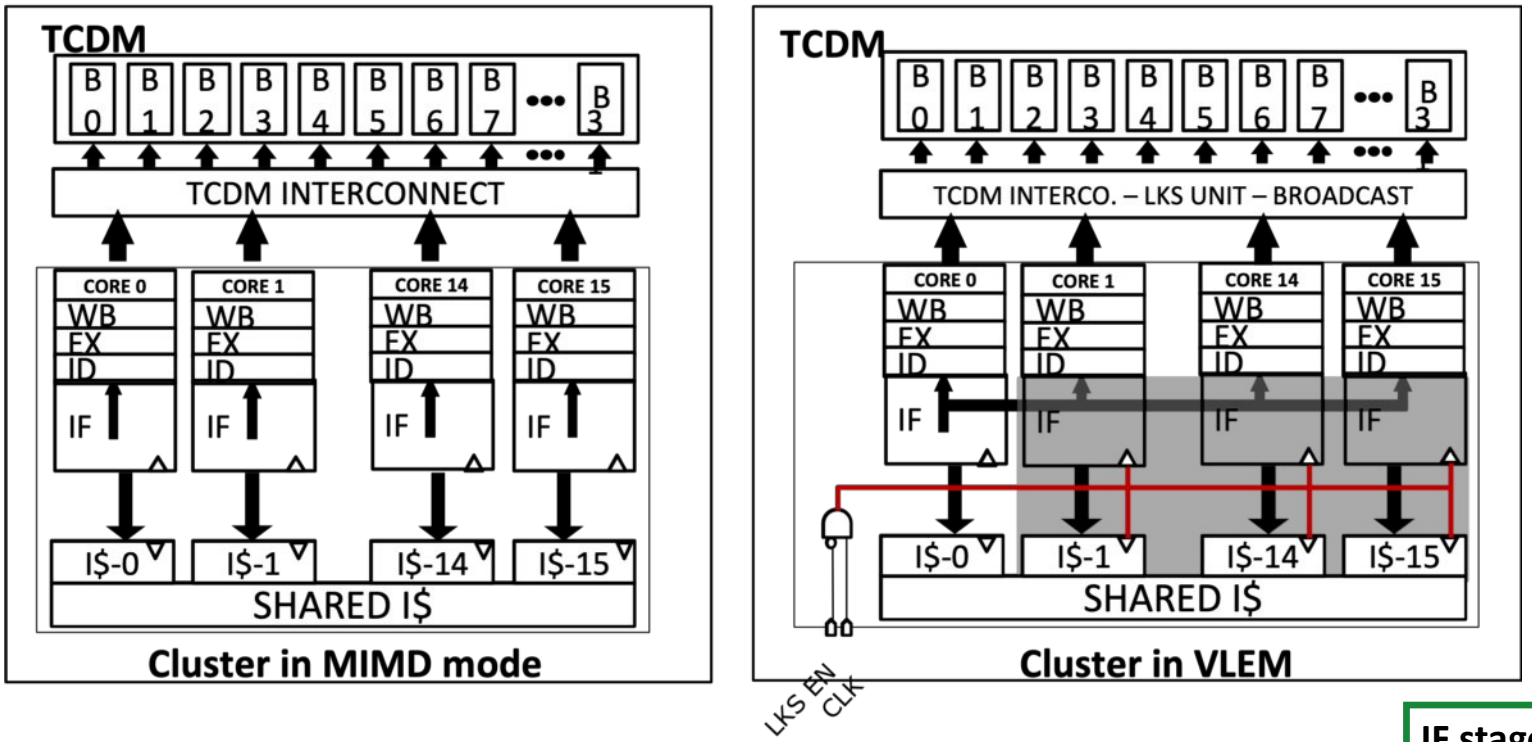
Low-Cost Solution for Edge Devices

Power analysis of a parallel 8-bit x 4-bit convolution



- Reduce unnecessary power consumption (not spent in computation)
- Exploit convolution's instruction and memory data access pattern regularity
- **How do we increase energy efficiency at low extra-area cost on resource-constrained devices?**
  - **→ reconfigurable MIMD/SIMD architecture**

Reconfigurable Vector Lockstep Execution Mode in PULP



- Cores enter in SIMD (VLEM) mode when executing regular kernels (in two clock cycles)
- In SIMD, instruction flow orchestrated only by the MAIN core → **Less energy**
- Cores resume in MIMD mode on divergent branches (..or control tasks) → **Flexibility**

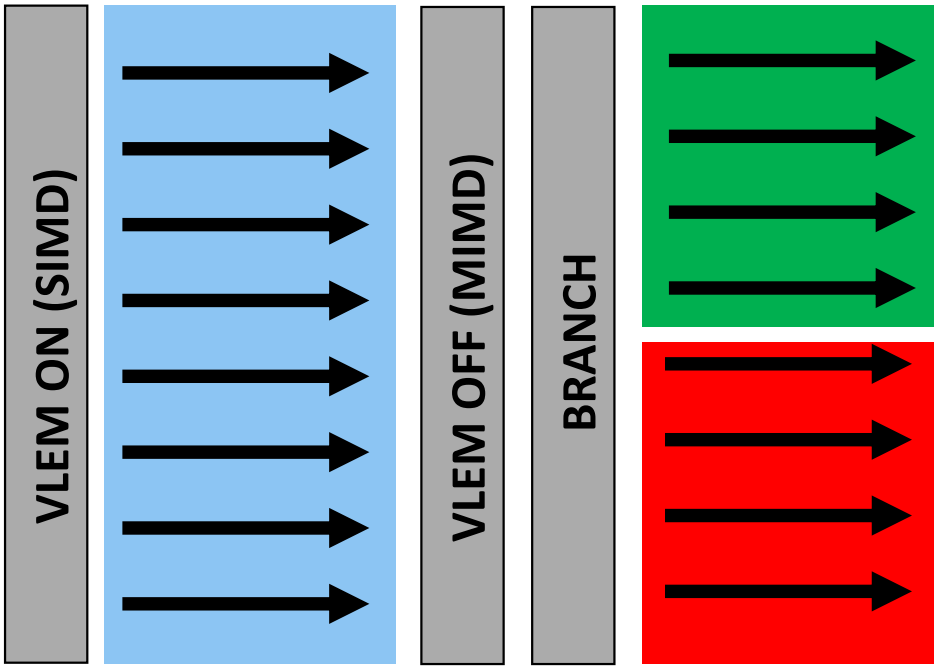
IF stages and private I$ of *follower* cores are clock gated in VLEM to save power

[Garofalo et al. ESSCIRC 2021]

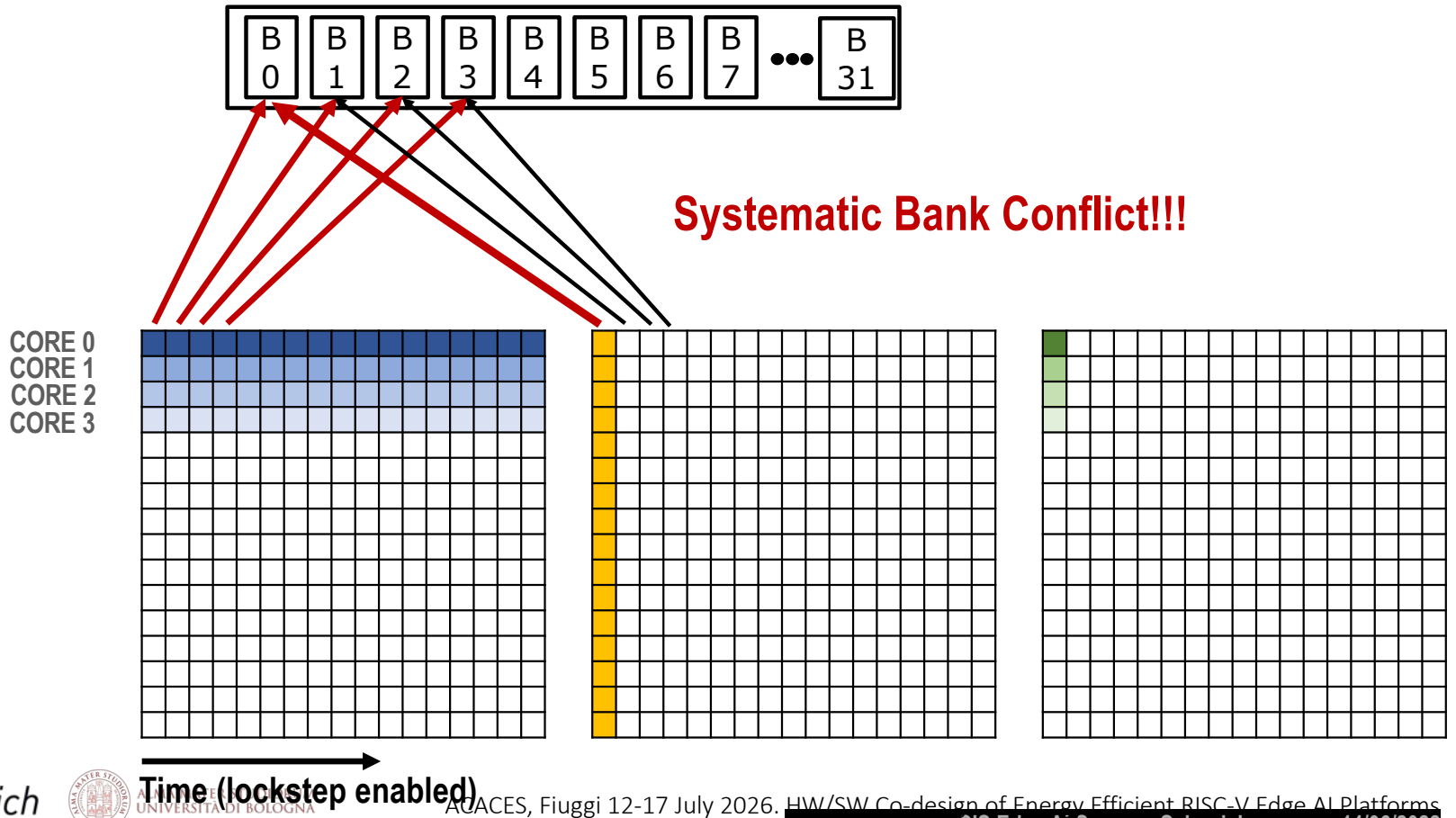
Reconfigurable Vector Lockstep Execution Mode in PULP

- SW RECONFIGURABLE SIMD/MIMD EXECUTION

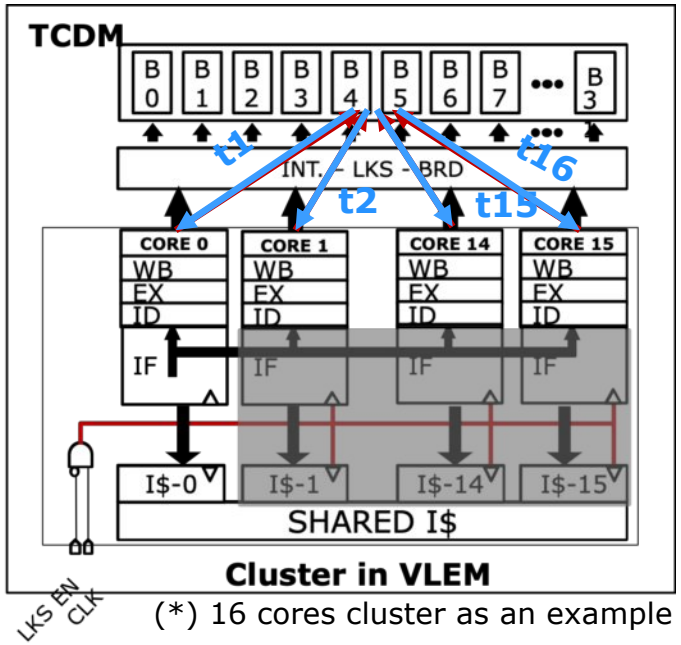
```
LOOP A

IF (ID < 4)
    EXEC X
ELSE
    EXEC Y
```



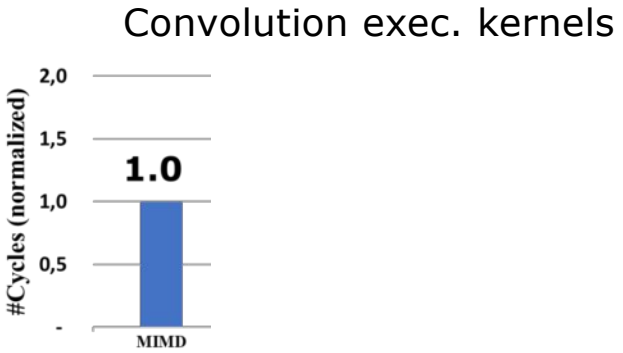
Systematic Bank Conflicts During MatMuls in VLEM



VLEM Performance Degradation due to Mem Conflicts

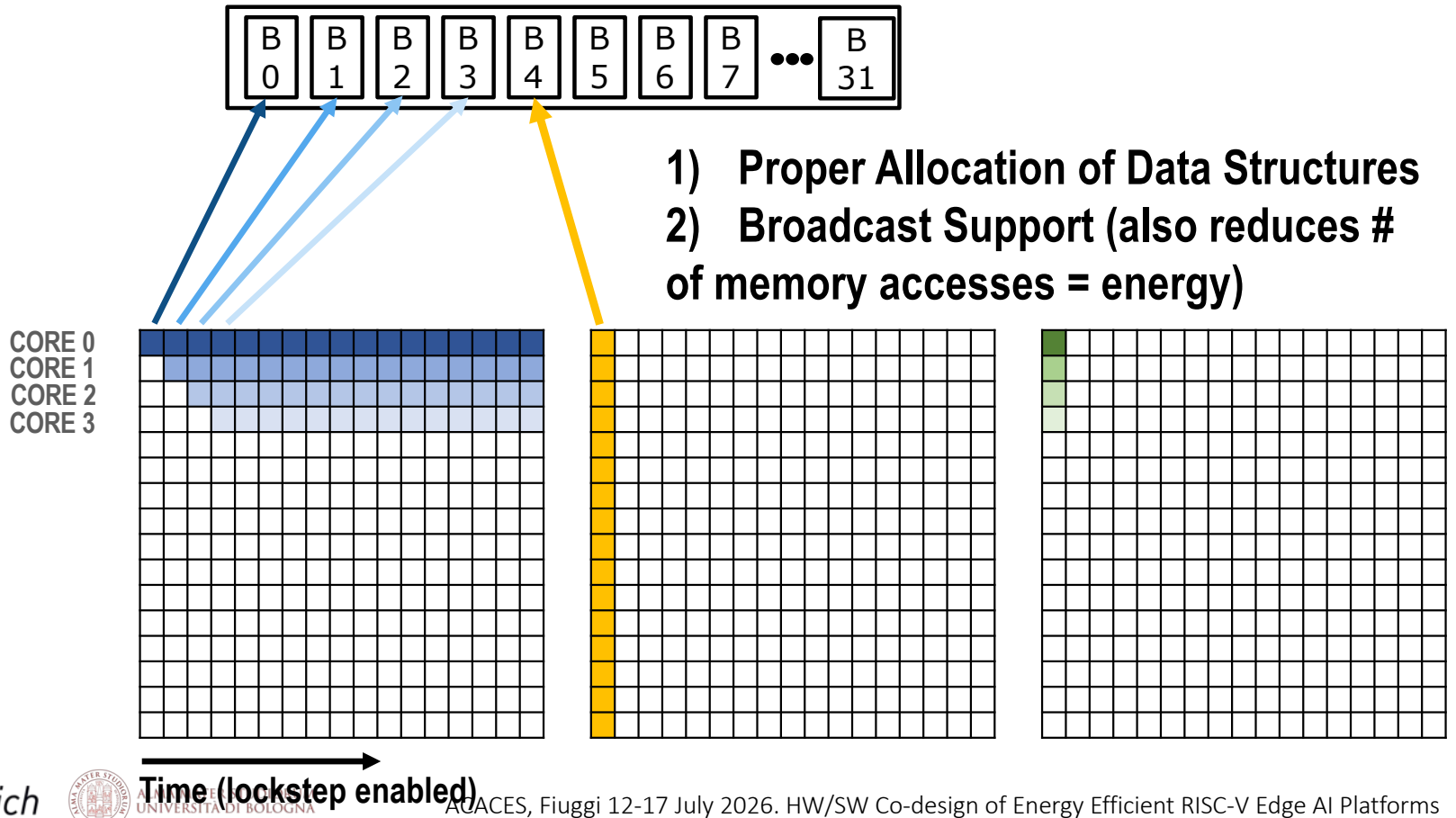


(*) 16 cores cluster as an example

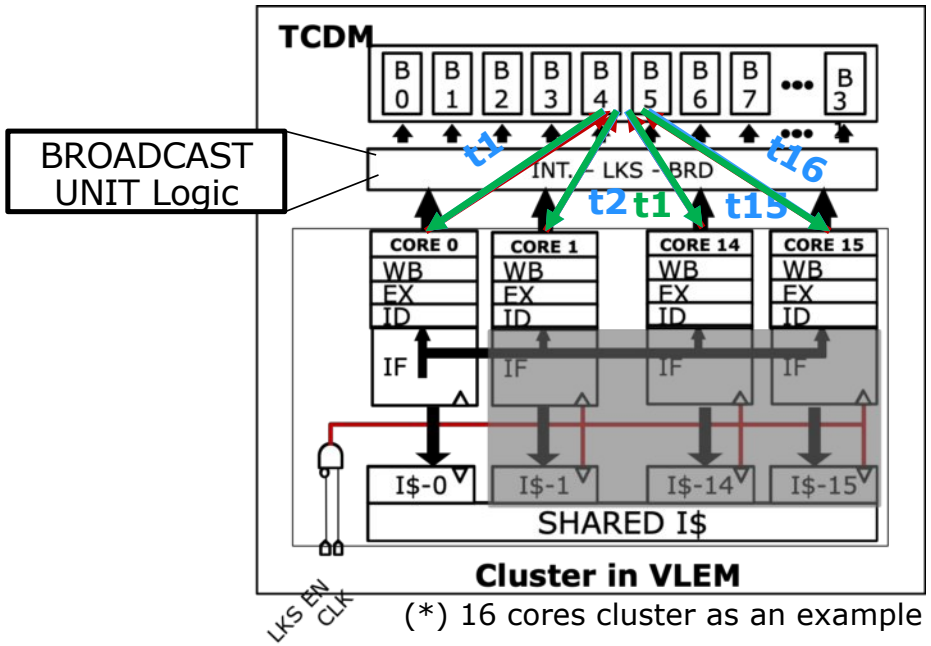


- **Overhead**: many clk cycles to unlock execution in case of concurrent accesses

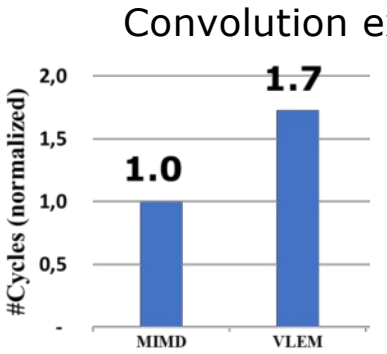
Reduce Conflicts through Broadcast + Misaligned Data



VLEM + Broadcast Unit + Static Misalignment of Data



(*) 16 cores cluster as an example

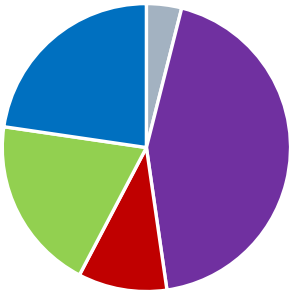


Low-cost solution! → Broadcast logic costs <10% area overhead w.r.t. MIMD only cluster

- **Overhead**: many clk cycles to unlock execution in case of concurrent accesses
  - Eliminate overhead to access at same address → **BROADCAST UNIT**
  - Misalign static data and stacks to avoid accesses to the same mem bank

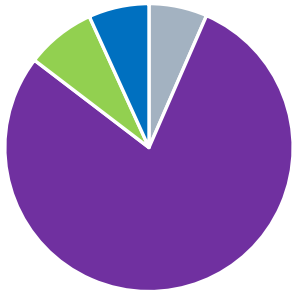
VLEM Energy Efficiency on MatMul Kernels

8x4 MatMul Power in MIMD: **29.2 mW**

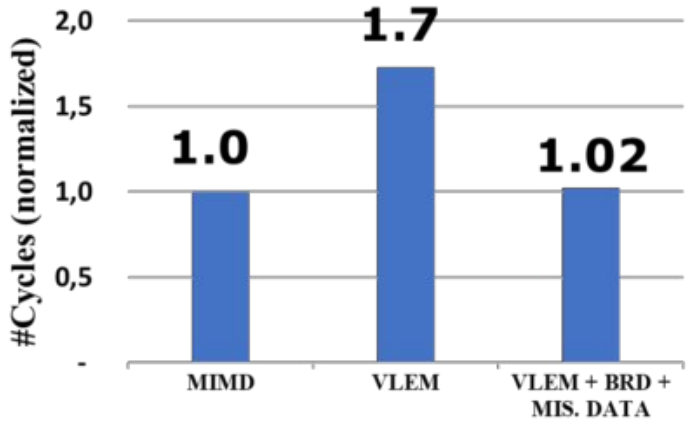


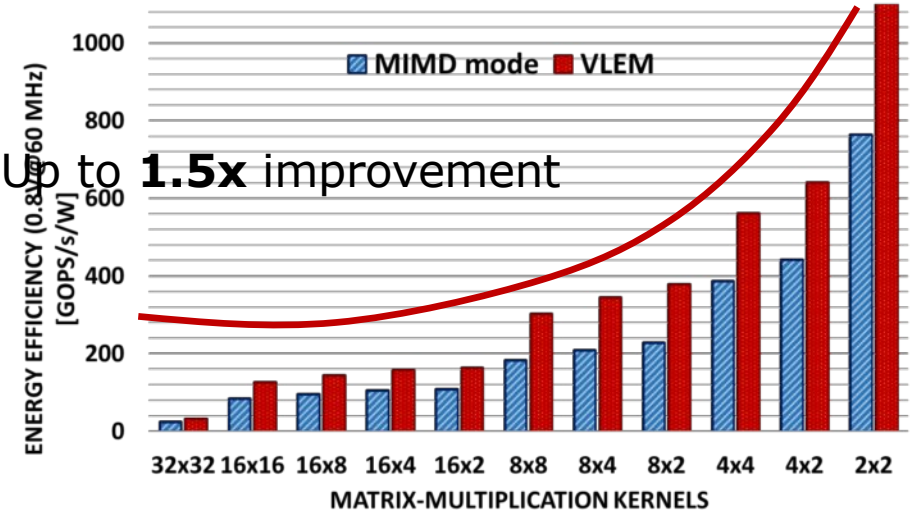
- CORE_0: **4%**
- CORES_1_15_ID_EX: **43.3%**
- CORES_1_15_IF: **10%**
- I$: **20%**
- TCDM. : **22%**

Power in VLEM: **16.1 mW** (~45% reduction wrt MIMD)



- CORE_0: **6%**
- CORES_1_15_ID_EX: **74%**
- ~~CORES_1_15_IF: **0%**~~
- I$: **7.3%**
- TCDM. : **6.3%**

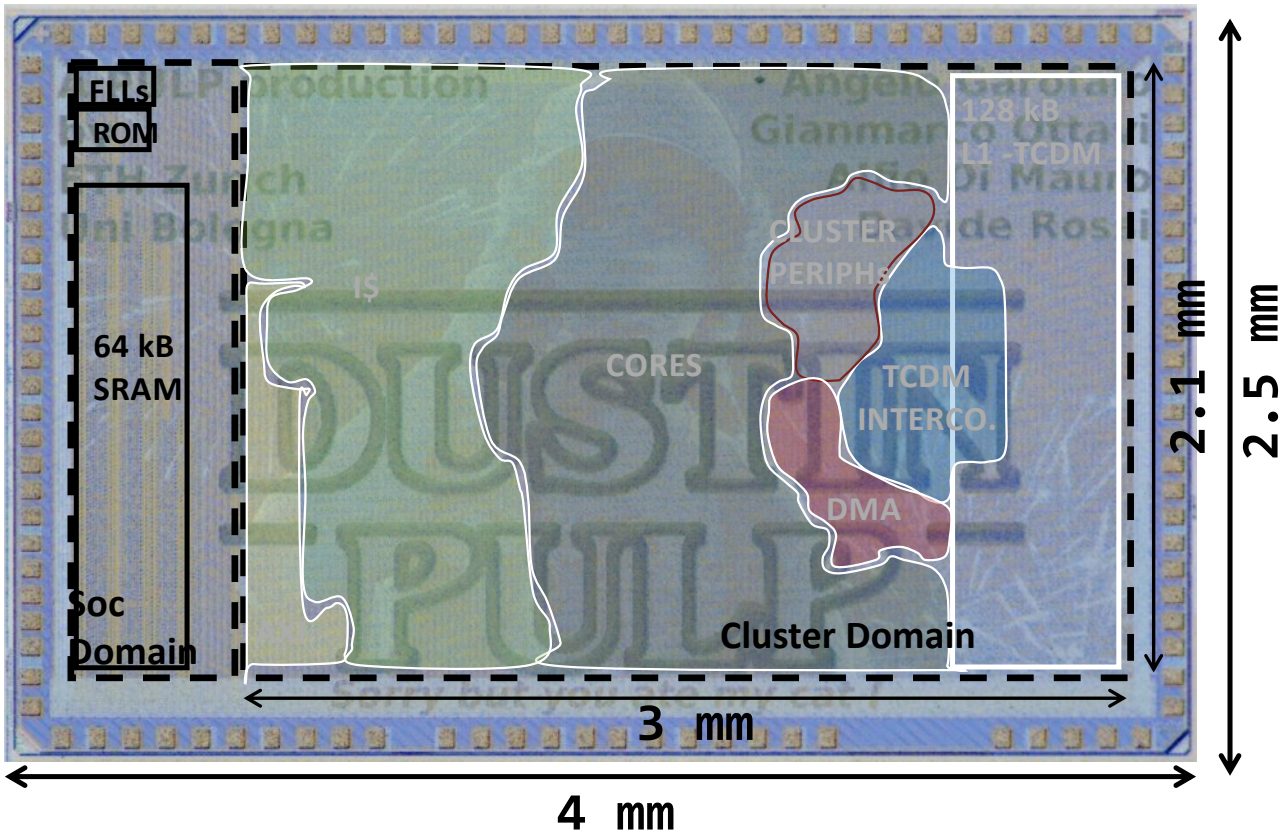




Up to **1.5x** improvement

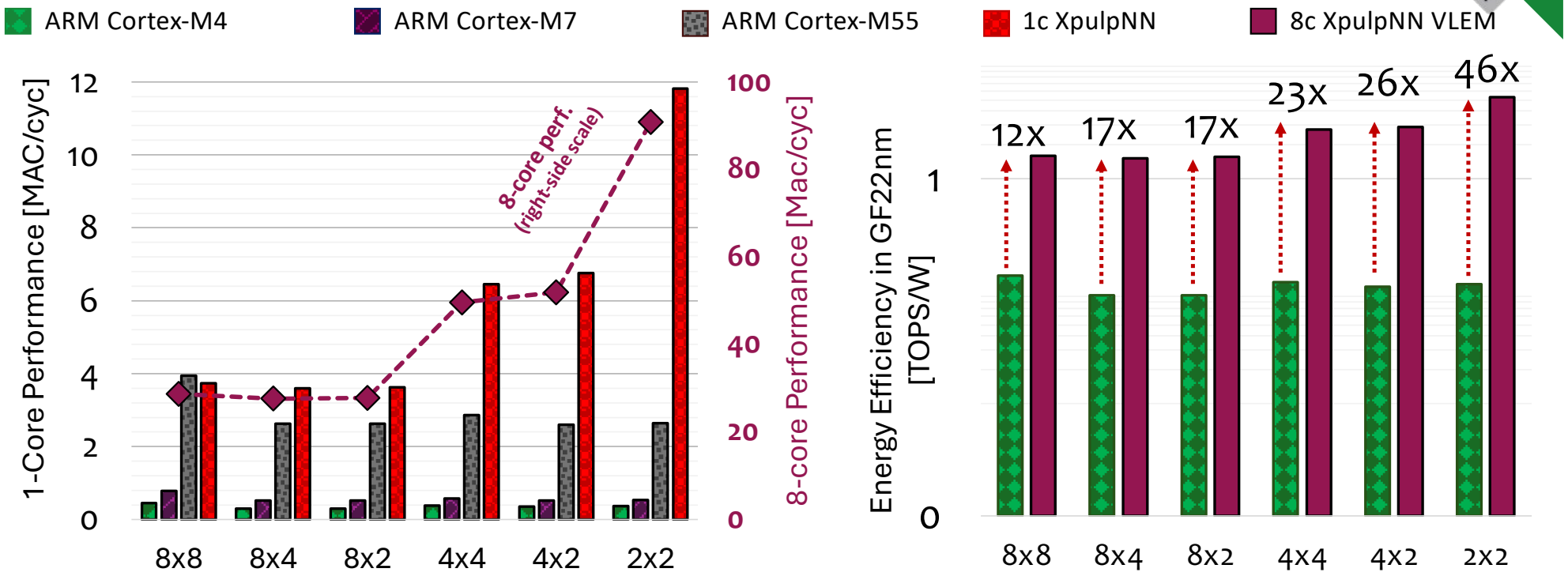
Can we scale-out with number of VLEM processors in the cluster?

These Concepts Are Silicon Proven in Dustin



Technology	CMOS TSMC 65nm
Chip area	10 mm^2
Total SRAM	208 kB
VDD range	0.8V - 1.2V
Power Envelope	156 mW
Frequency Range	60 – 205 MHz

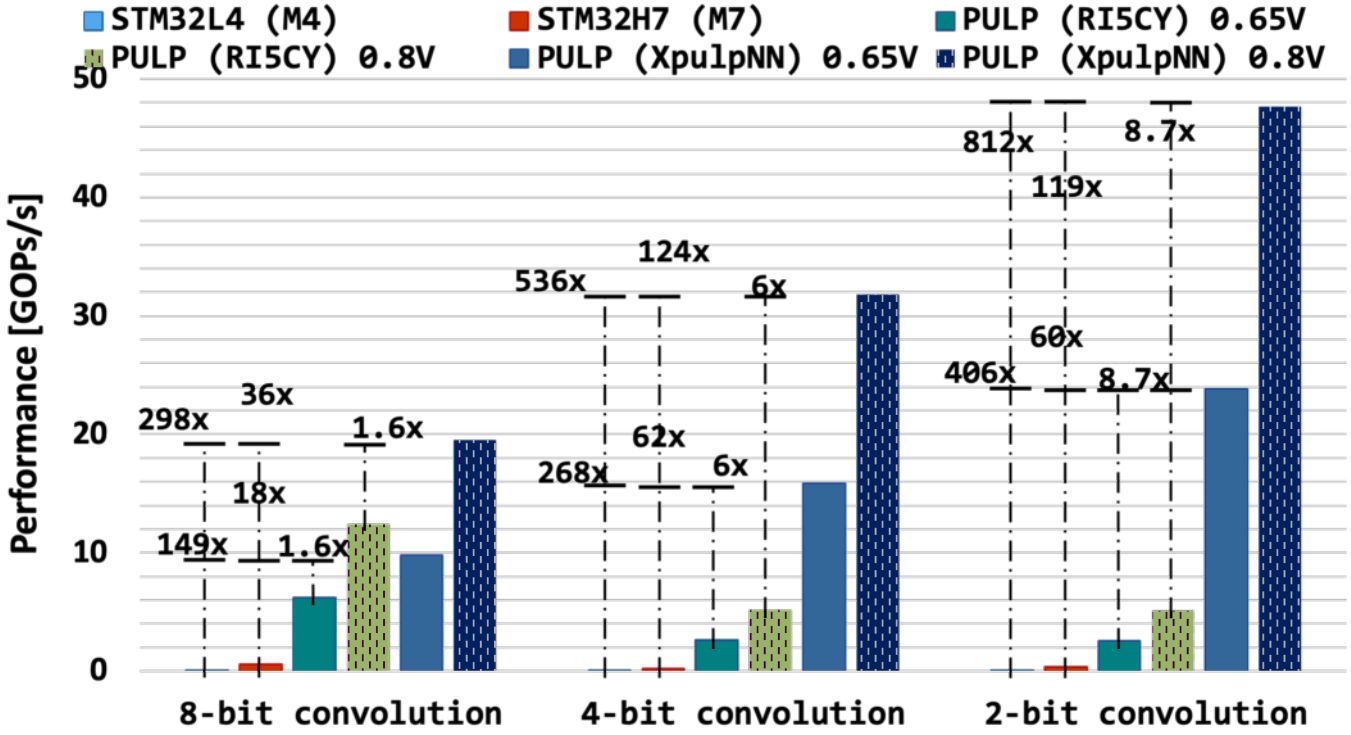
Accelerator-Like Energy Efficiency



- 8-c cluster in GF22: 21mW @ 400MHz (**14.1 mW** in VLEM) – **0.8 mm²**
- **Energy-efficiency comparable with custom accelerators: >5 TOPS/W**

[Garofalo et al. IEEE D&T 25]

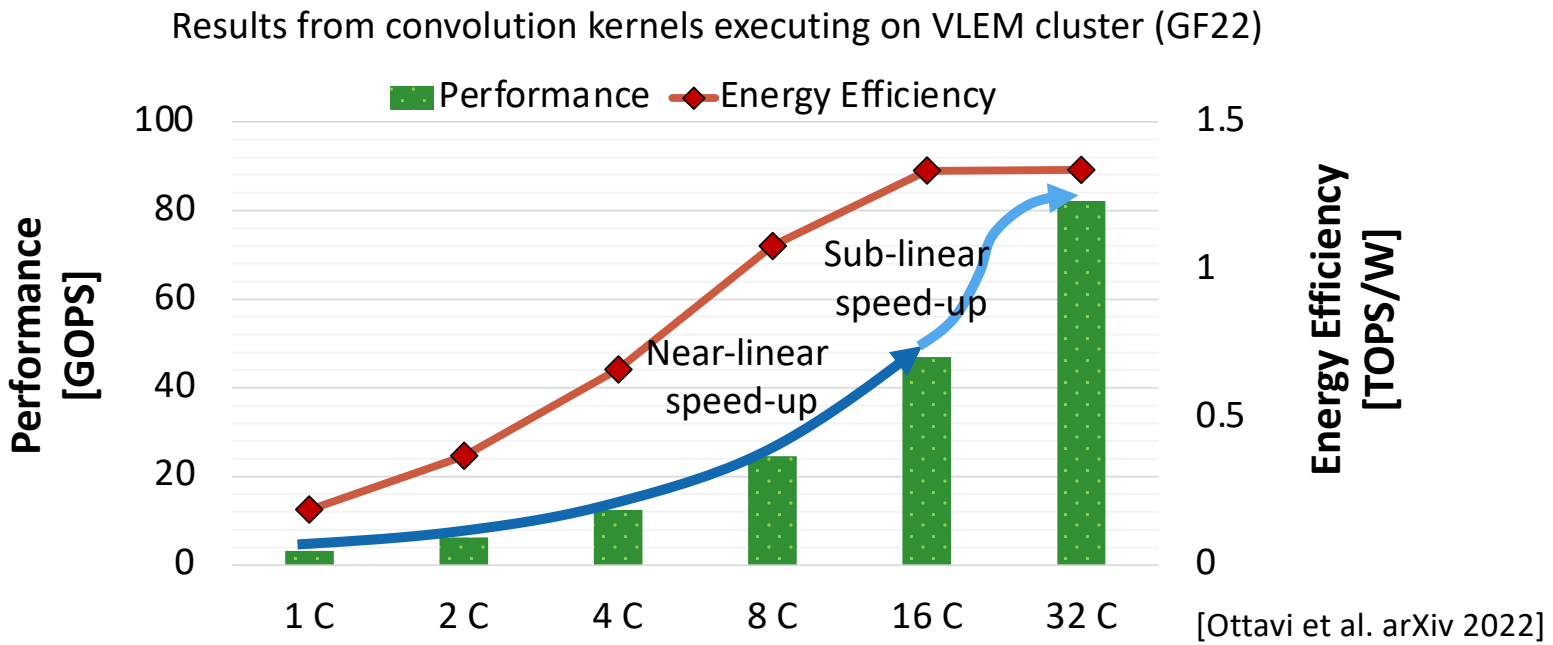
Performance Comparison Against Commercial Devices



8 cores results in GF22

- **Reasonable Performance for TinyML Applications**
  - >100 GOPS w/ 16 cores @ 650MHz, 0.8V in GF22 FDX Technology
- **Can we scale more to target more complex AI workloads?**

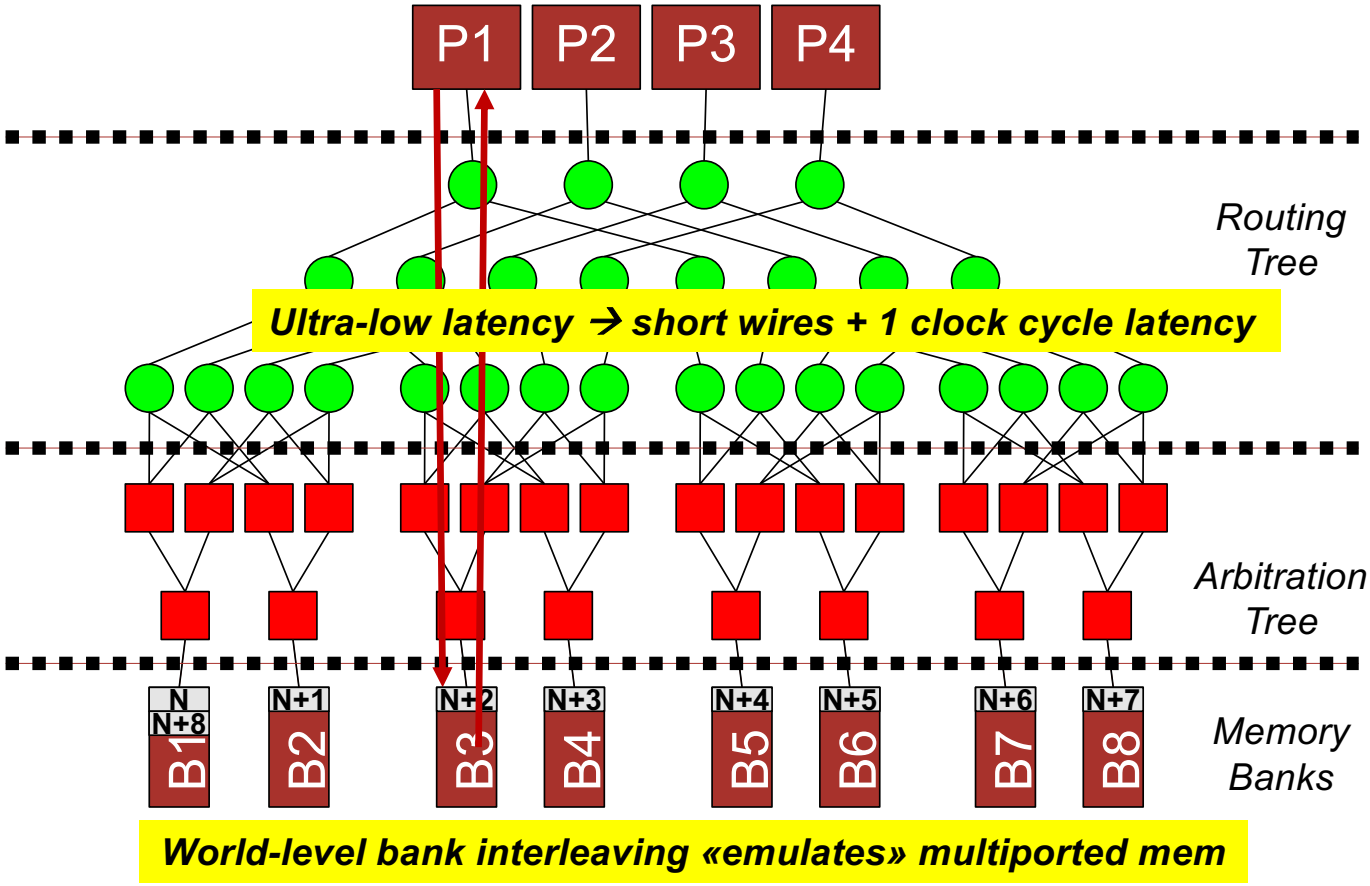
Scaling Issues of the PULP Cluster



[Ottavi et al. arXiv 2022]

- Performance bottleneck at the **core's boundaries** (single 32-bit data port)
- Energy efficiency bounded by **limited scalability of low-latency local interconnect**
- How to address these scalability issues?

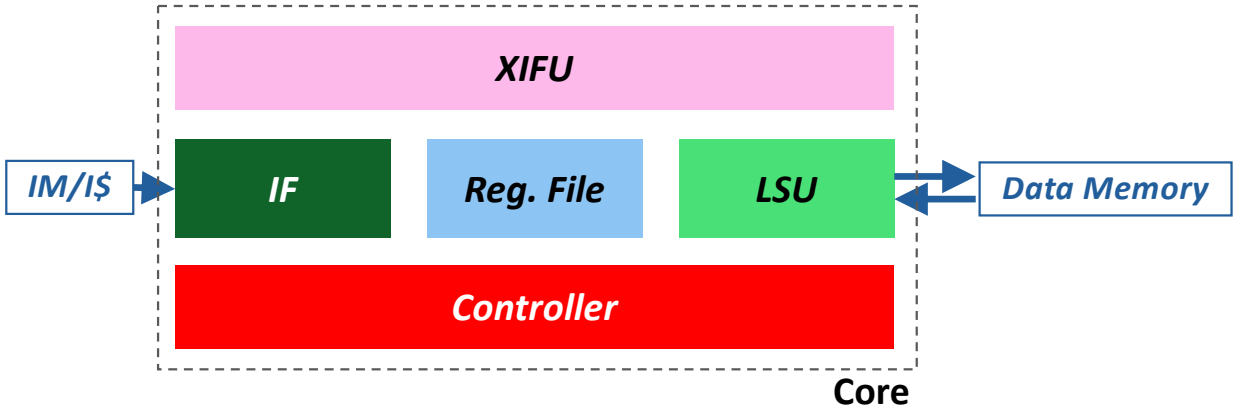
Scaling Issues: The Interconnect



A. Rahimi, I. Loi, M. R. Kakoee and L. Benini, "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters," *2011 Design, Automation & Test in Europe*, 2011, pp. 1-6.

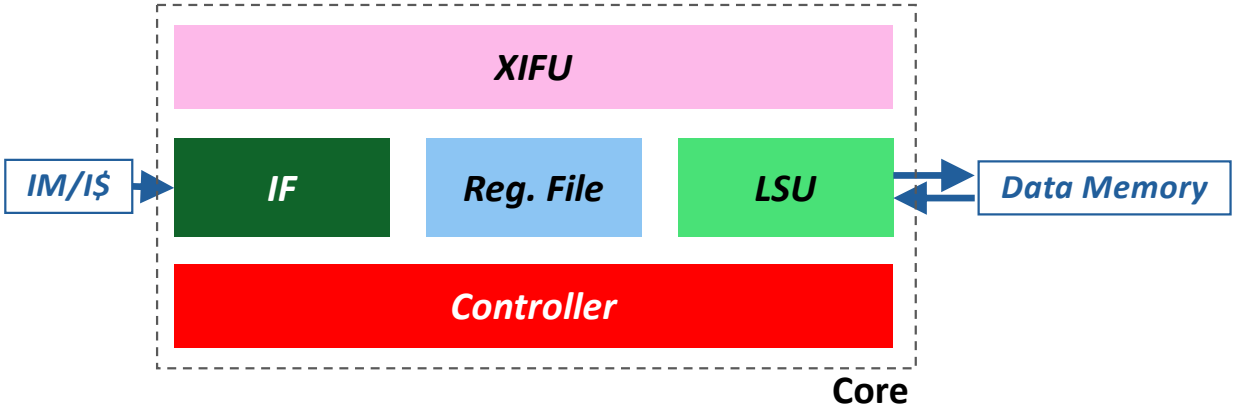
Limits of Load-Store Architectures

- **Let's consider the various flavors of ISA extensions we have seen so far**
- **We will consider few bottlenecks and related workarounds**



The Register File Bottleneck

- **Arithmetic instructions are REG-REG operations**
  - **Reg-Reg** instructions are bottlenecked by the **architectural register file**
    - **Limited size**, i.e. the number of architecturally exposed registers is small (=32 in RISC-V)
    - **Limited bandwidth**, 3 read ports and 2 write points in RISCY core we have seen

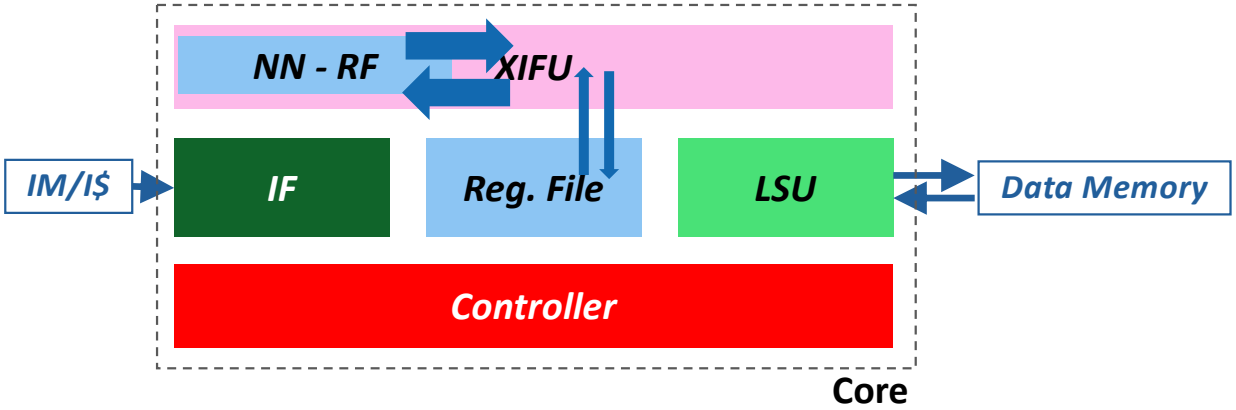


  - These **architectural limitations** limit the **data reuse**
    - At best, PULP-NN MatMul kernels cannot scale beyond **4x2 geometric size** in RI5CY
    - **Frequent back-and-forth** between Register File and L1 Memory
    - Register File is a shared resource with all other instructions! → **sub-optimal** to be used for intense computation

The Register File Bottleneck – a Workaround

Core with Special RF (XpulpNN)

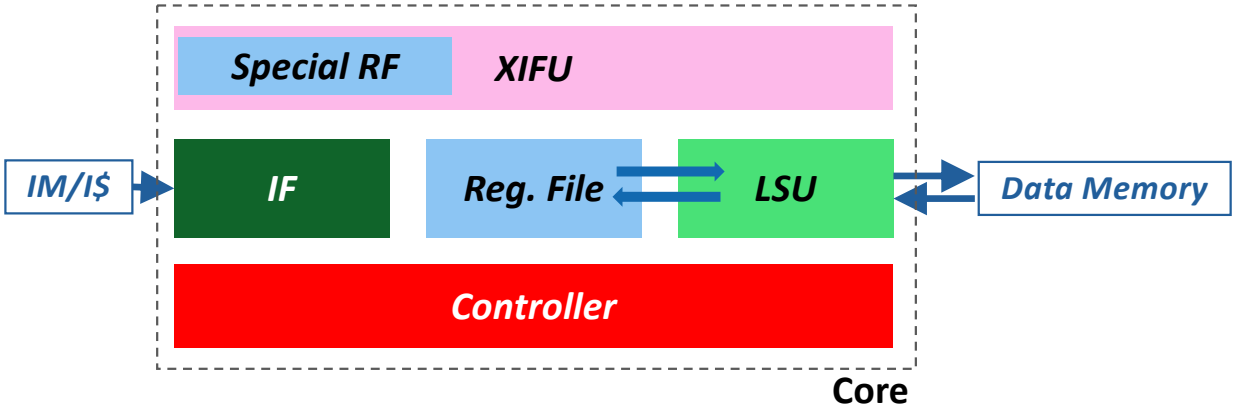
- **Create a special-purpose register file alleviates the bottlenecks**
  - Special Register File, e.g. NN-RF in XpulpNN, is **not shared with SW**, and it can be arbitrarily large (but at a cost!)
    - **Data reuse limited only by the overall NN-RF size**, which can potentially be > 32 elements
    - NN-RF can also be organized in different buffers, delivering **higher bandwidth**



  - However, there are still some limitations..
    - **Less flexibility**: Special RF is by definition less architecturally visible than architectural RF ☹
    - Alla data ultimately will be **bottlenecked by LSU** and memory accesses ☹

The LSU Bottleneck

- **The core's LSU has limited bandwidth**
  - Maximum 32b/cycle in RI5CY
    - …assuming we issue 1 load/store instruction per cycle (or mac-load)
    - In PULP cluster **LSU's bandwidth** shared with other cores → reduced bandwidth in case of conflicts

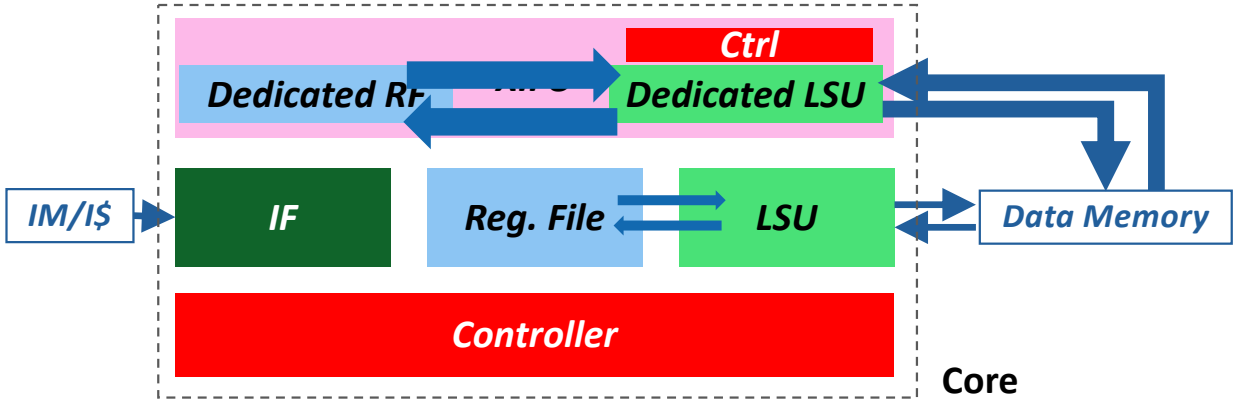


  - Such a limited L1 memory bandwidth limits large acceleration schemes

The LSU Bottleneck – a Workaround

Streaming co-processors w/ dedicated RF

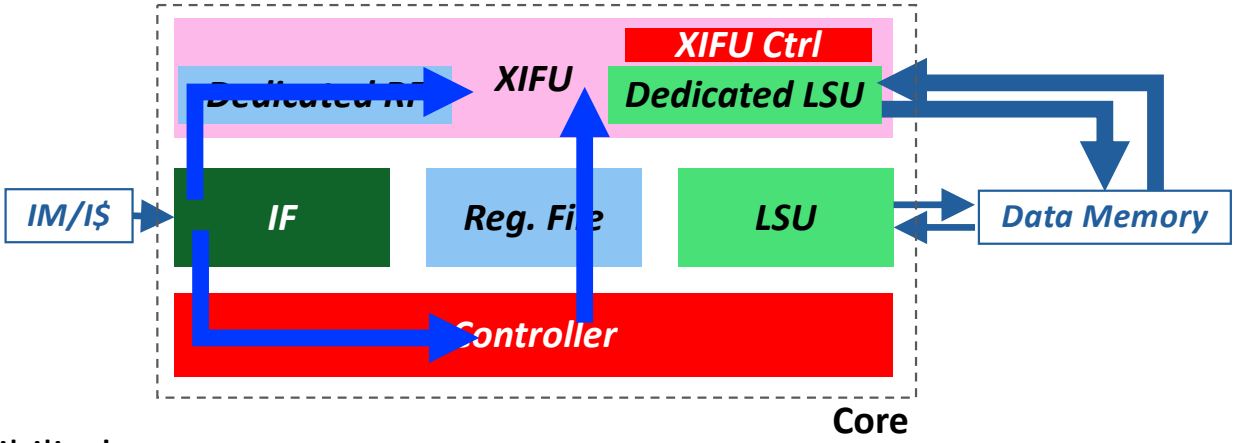
- **Provide the co-processor or ISA extension with its own LSU**
  - Can provide the **required bandwidth** (at a cost)
  - Does not occupy the core LSU
    - LD/ST on extension & main core can proceed in parallel
    - We need a small controller to control the extension LSU



  - Can create **memory consistency problems** if used in a very "tightly coupled" way
    - Especially if extension operation lasts more cycles than the core pipeline stages

The Von Neumann Bottleneck

- **An ISA extension still executes *instructions*, leading to a control bottleneck**
  - It needs to **fetch/decode/execute** ("Von Neumann bottleneck")
    - Keeps the core occupied with "menial" control tasks
    - Either relax "RISC" assumption or accept very fine-grain computation (more difficult to accelerate)

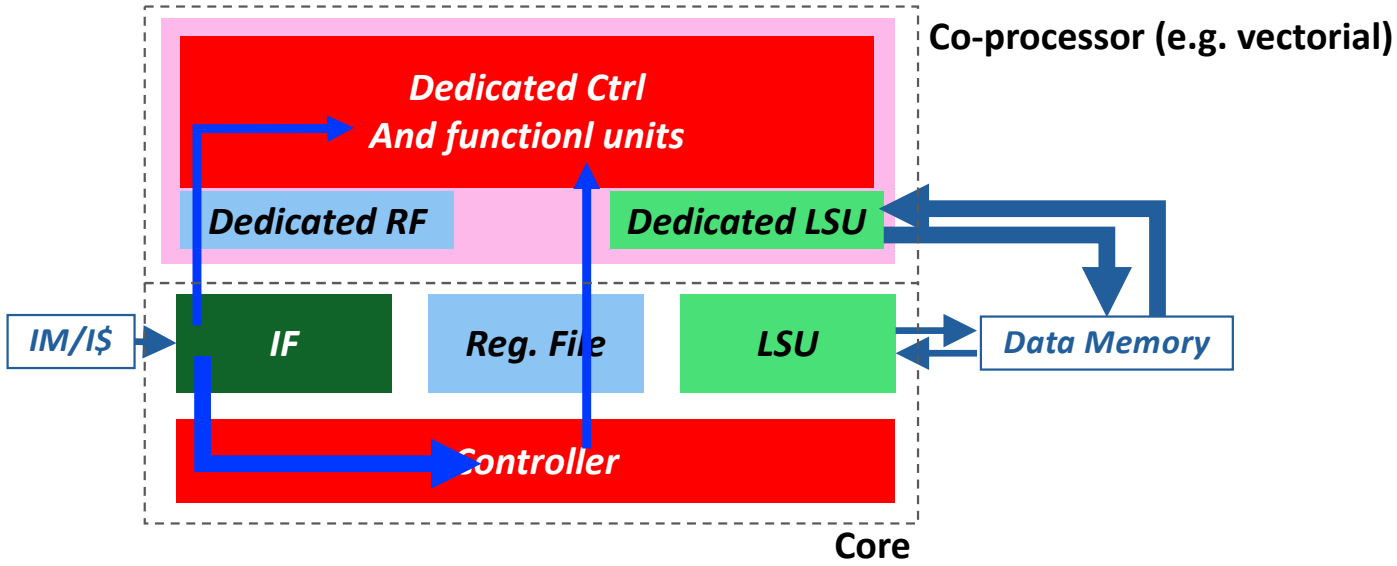


  - The price of flexibility!

Tightly-coupled Co-processors

Tightly-coupled Coprocessor (RV Vector co-prorcessor)

- **Delegate more control to the extension itself - Coprocessor**
  - Coarser grain, less fine-grained instructions

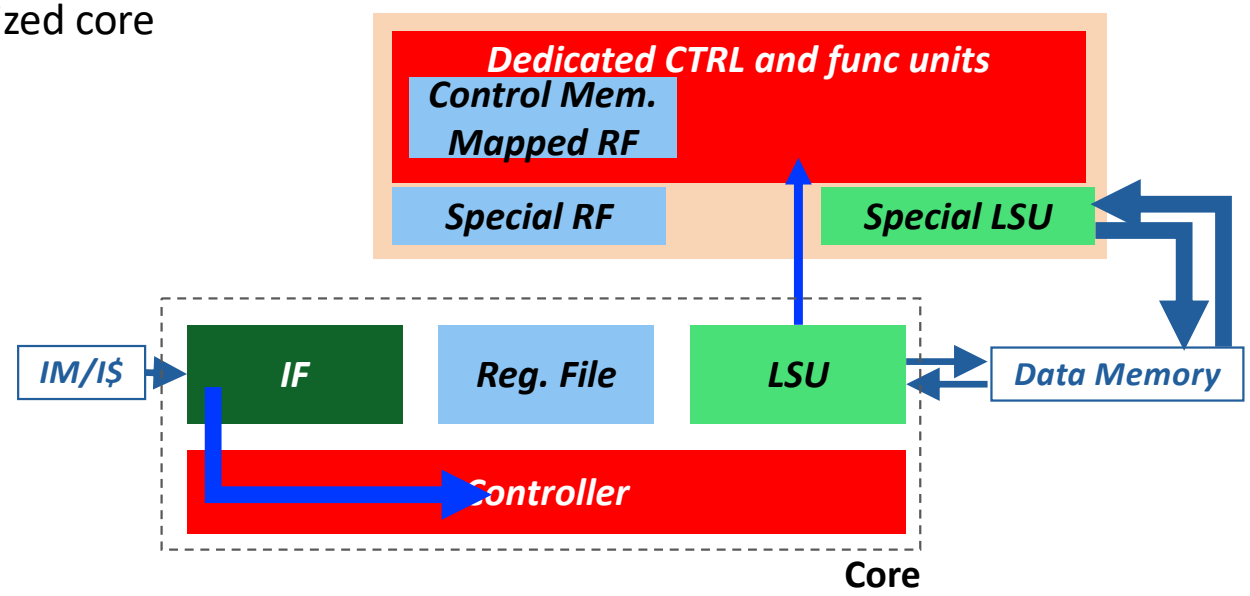


- **Dedicated local register file, LSU, controller**

Loosening the Core-coupling

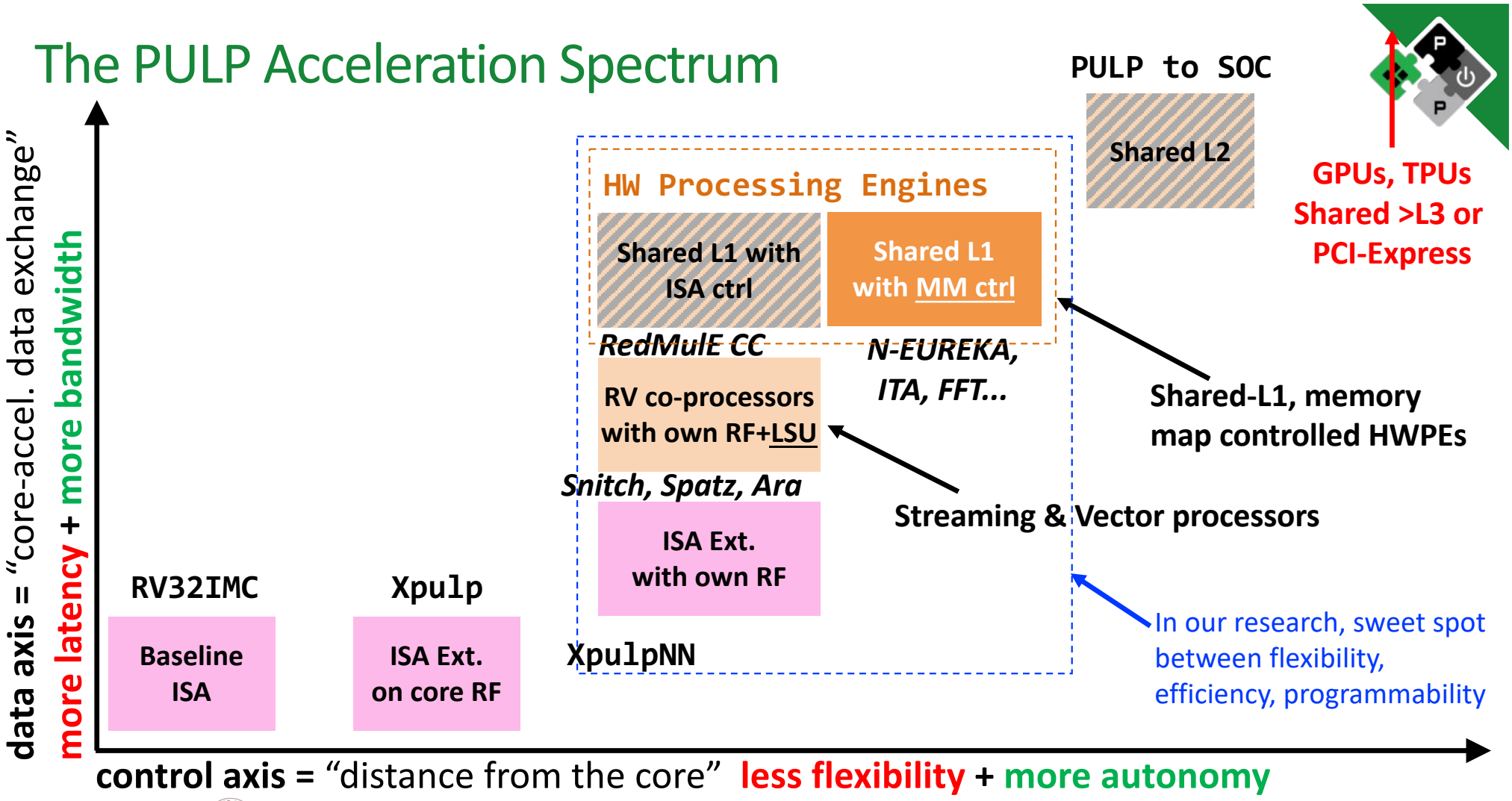
Memory-mapped Accelerator in a Heterogeneous Cluster

- **Move the only remaining coupling (controller) to memory-mapped control**
  - Not really a coprocessor any more, but something else entirely: an **independent engine** / specialized core



  - This can be anything ranging from a *really* **loosely-coupled core** (100-1000s of cycles to move data core to accelerator) to **a PULP cluster-coupled engine** (sharing memory at L1)

The PULP Acceleration Spectrum



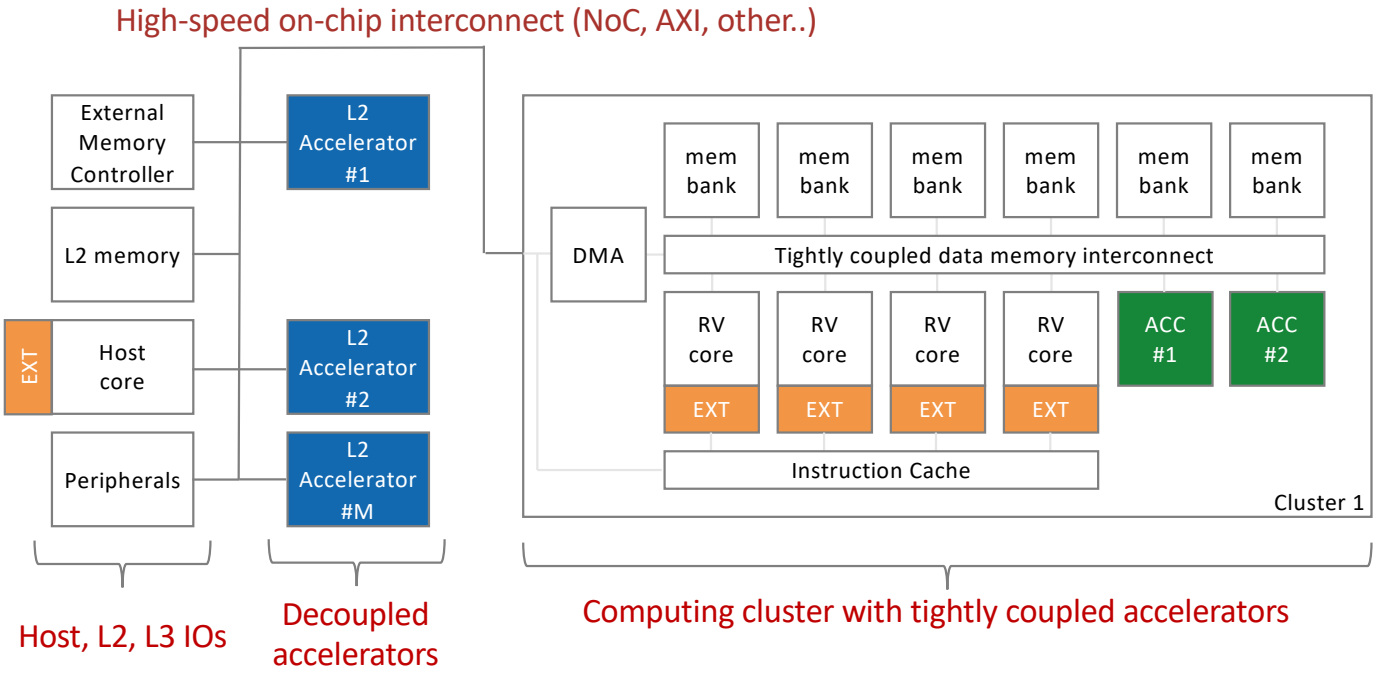
Heterogeneous, Multiscale Accelerated Computing

Multiple Scales of acceleration

Extensions to processor cores
- Explore new extensions
- Efficient implementations

Shared-memory Accelerators
- Domain specific
- Local memory

Multiple Decoupled Accelerators
- Communication
- Synchronization



RISC-V is a key enabler → max agility, enabling SW build-up, without vendor lock-in

PULP: Open Hardware Toolbox for DSAs

RISC-V Cores and Vector Units

RI5CY *CV32E*	Zero R *Ibex*	Snitch	Spatz	Ariane *CVA6*	ARA
RV32	RV32	RV32	RVV	RV64	RVV

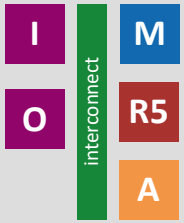
Peripherals

JTAG, SPI, UART, I2S, DMA, GPIO

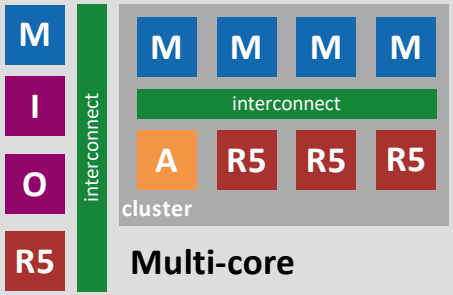
Interconnects

LIC, HCI, APB, FlooNoC, AXI4

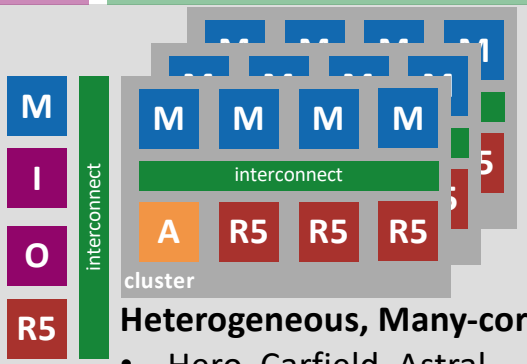
Platforms



Single core
- PULPino, PULPissimo
- Cheshire



Multi-core
- OpenPULP
- ControlPULP



Heterogeneous, Many-core
- Hero, Carfield, Astral
- Occamy, Mempool

IOT https://github.com/pulp-platform HPC

Accelerators and ISA extensions

XpulpNN, XpulpTNN	ITA (Transformers)	RBE, NEUREKA (QNNs)	FFT (DSP)	REDMULE (FP-Tensor)

Summary

- **Low-power energy-efficient TinyML cluster architectures**
  - Multi-core cluster of tiny RISC-V processors
  - Single-cycle latency interco + multi-banked scratchpad memory managed by DMA for efficient data movements
  - Sincronization in HW for fast and low-power barriers
- **Vector Lockstep Execution Mode (VLEM) to reduce unnecessary power**
  - Similar to the concept of GPUs (still, not equal!)
  - Drawbacks of VLEM solvable with HW/SW co-operation
- **Scalability issues:**
  - The MIMD/VLEM approach not scalable beyond 16 cores
    - Von Neumann Bottleneck + physical constraints on the interconnect
- **Next:**
  - RISC-V Solutions to address scalability issues: 1) Heterogeneity; 2) Streaming; 3) Vector paradigm

ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

HW/SW Co-design of Energy Efficient RISC-V Edge AI Platforms

ACACES 2026, Fiuggi (Italy), 12-17 July 2026

Part 4: Addressing the Von Neumann Bottleneck with Heterogeneous Clusters and Streaming Clusters

Angelo Garofalo angelo.garofalo@unibo.it
agarofalo@iis.ee.ethz.ch

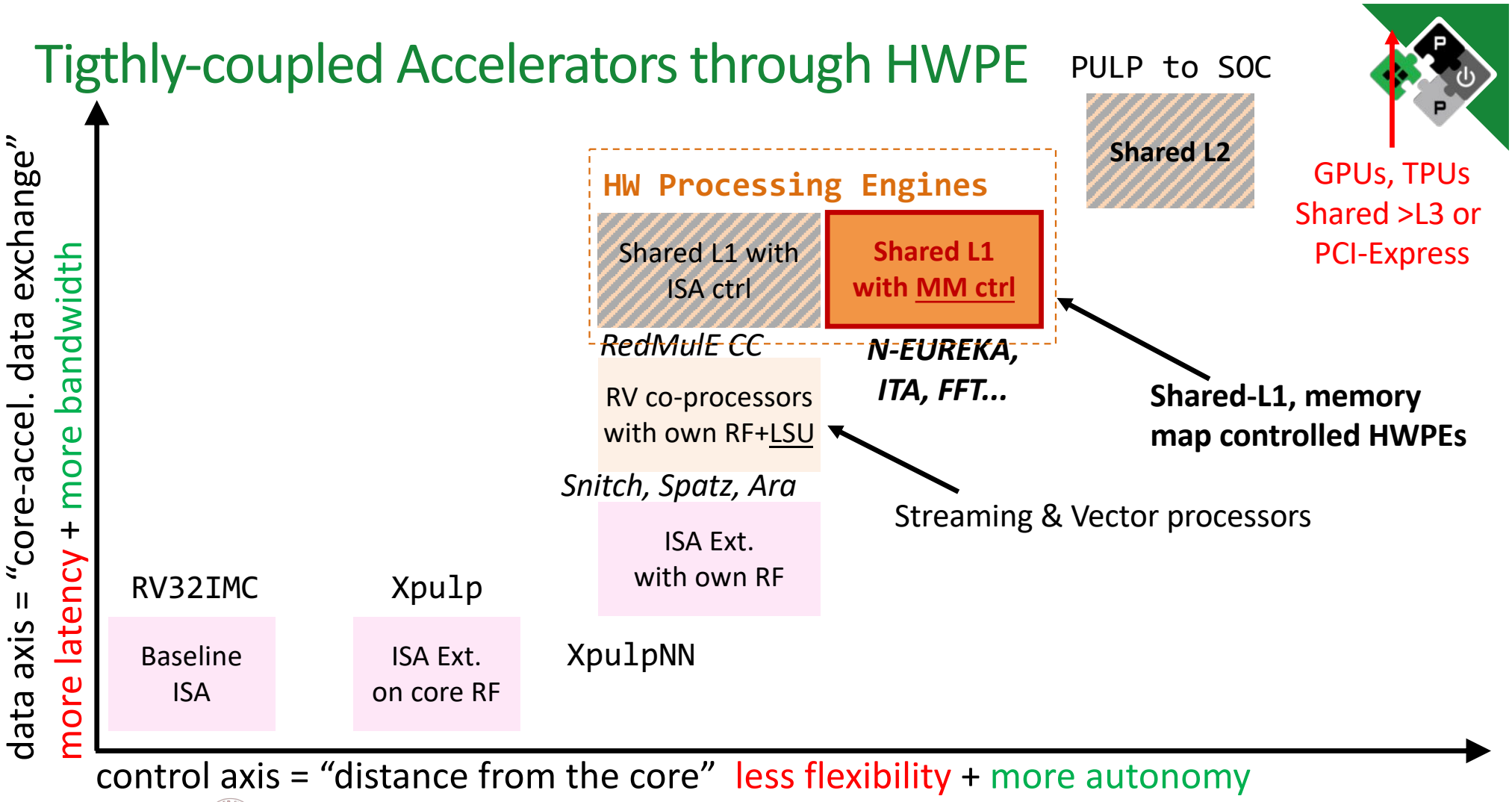
pulp-platform.org
@pulp_platform
company/pulp-platform
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

Outline

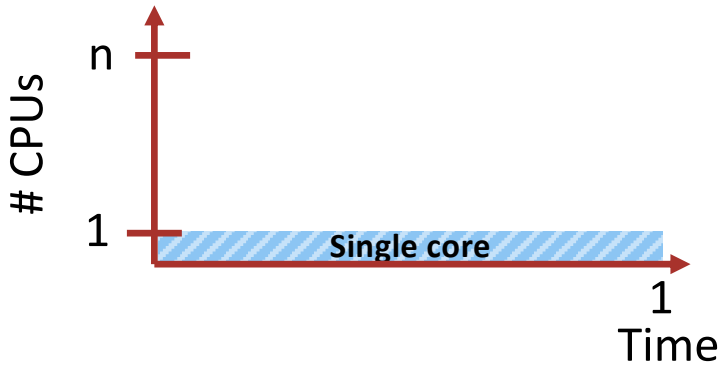
- Why Heterogeneous Clusters?
  - Challenges and Solutions
- Tighly-coupled heterogeneous accelerators
  - The Hardware Processing Engines (HWPE)
  - The Heterogeneous cluster interconnect
  - Fast heterogeneous synchronization
- In-memory computing centric analog/digital RISC-V PULP cluster
  - Integration challenges of analog in-memory computing accelerators
  - The role of RISC-V cores to mitigate Amdhal's Effects

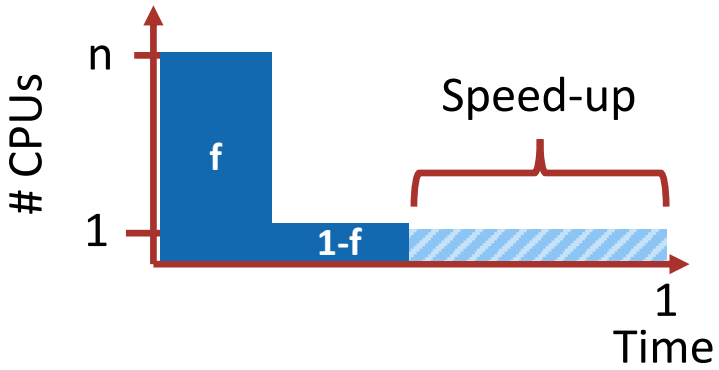
Tigthly-coupled Accelerators through HWPE



The Need of Heterogeneous Systems

- Recap of Amdahl's Law
  - f → code that can run in // ; (1-f) → code that must run serially; n → # of PEs/CPUs





$$Speedup = \frac{1}{(1-f) + \frac{f}{n}}$$

$$\lim_{n\to\infty} \frac{1}{1-f+\frac{f}{n}} = \frac{1}{1-f}$$

The Need of Heterogeneous Systems

- Amdahl's Law applied to accelerators
  - A) Sequential dataflow ; b) parallel dataflow

- **Revised Amdahl's Law**

$$\text{(a) Speed-up (SU)} = \frac{f_{host}+f_{acc1}+f_{acc2}}{f_{host}+f^{A}_{acc1}+f^{A}_{acc2}+OH} < \frac{f_{host}+f_{acc1}+f_{acc2}}{f_{host}}$$

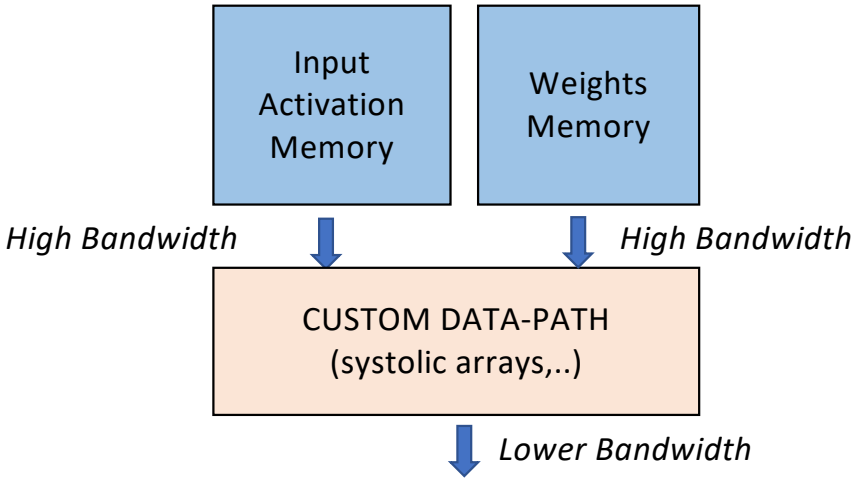
$$\text{(a) Speed-up (SU)} = \frac{f_{host}+f_{acc1}+f_{acc2}}{f_{host}+\text{Max}(f^{A}_{acc1}+f^{A}_{acc2})+OH} < \frac{f_{host}+f_{acc1}+f_{acc2}}{f_{host}}$$

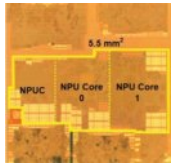
The Need of Heterogeneous Systems

- **Performance bottleneck keeps shifting toward non-accelerated kernels**
  - Solutions
  - Balance workload through **many acceleration sub-systems**
  - Residual computation perfomed in **SW** on domain-specialized RISC-V Cores

- **Hardware challenge becomes efficient data movements within the system**
  - Solutions
  - **Tightly-coupled integration** schemes
  - High-bandwidth **low-latency** interconnect
  - **Synchronization in hardware** (among PEs) for lower latency and energy

Specialized AI Accelerators - Digital

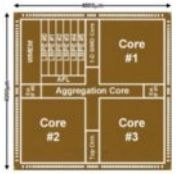




[Song et al., ISSCC19, 8nm]

[Desoli et al., ISSCC17, 28nm]

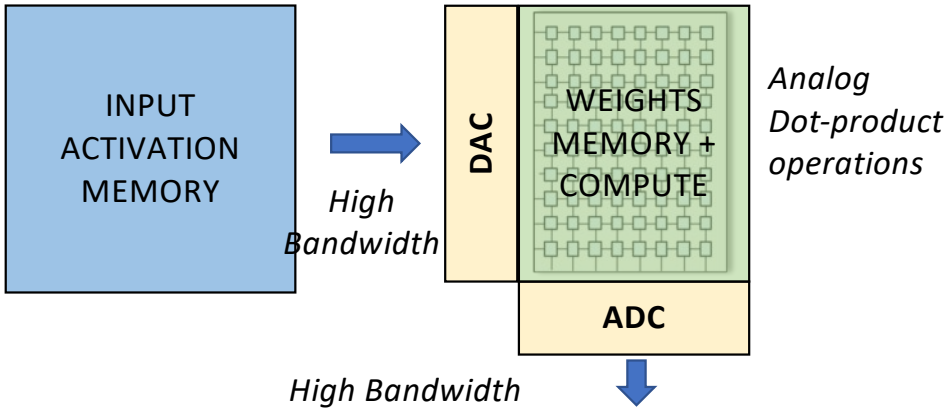
[Moons et al. ISSCC 2017]



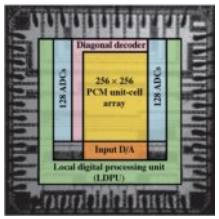
[Lee et al., ISSCC18, 65nm]

- Custom MAC units, data-paths, data precision (binary, ternary, 2-8 bits)
- Perf. **[0.1, 1] TOPS**, En. Eff. **[1, 100] TOPS/W**
- Hardware challenge moved at the boundaries, on data movements

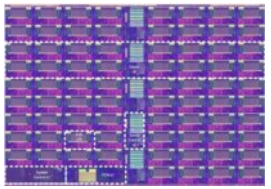
Specialized AI Accelerators - Analog



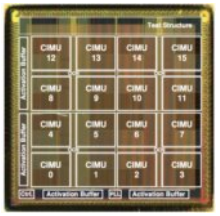
Not in scale



[Khaddam et al. JSSCC, 2022]



[Fick et al. ISSCC 2022]



[Jia et al. ISSCC 2021]

- Perf.: **[10, 100] TOPS**
- En. Eff. **[100, 1000] TOPS/W**

- Weights are stored where the computation happens (nvAIMC arrays)
- Analog MAC more efficient than digital MAC (10x better perf. and en. eff.)
- Energy efficiency vs. accuracy
- Compute density vs. re-programmability

Edge AI Processors

ISA-based acceleration

Custom digital data-paths

Analog In-Memory Computing cores

1-5pJ/OP

0.05-1 pJ/OP

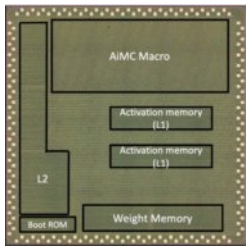
10-100 fJ/OP

SW programmable

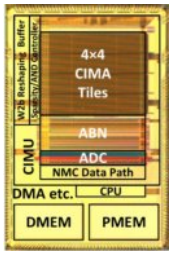
Re-configurable

Only MVMs

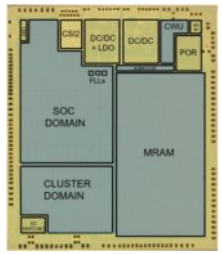
- Heterogeneous integration for flexible and efficient AI computation
  - But this introduces some challenges…



[Ueyoshi et al. ISSCC 2022]

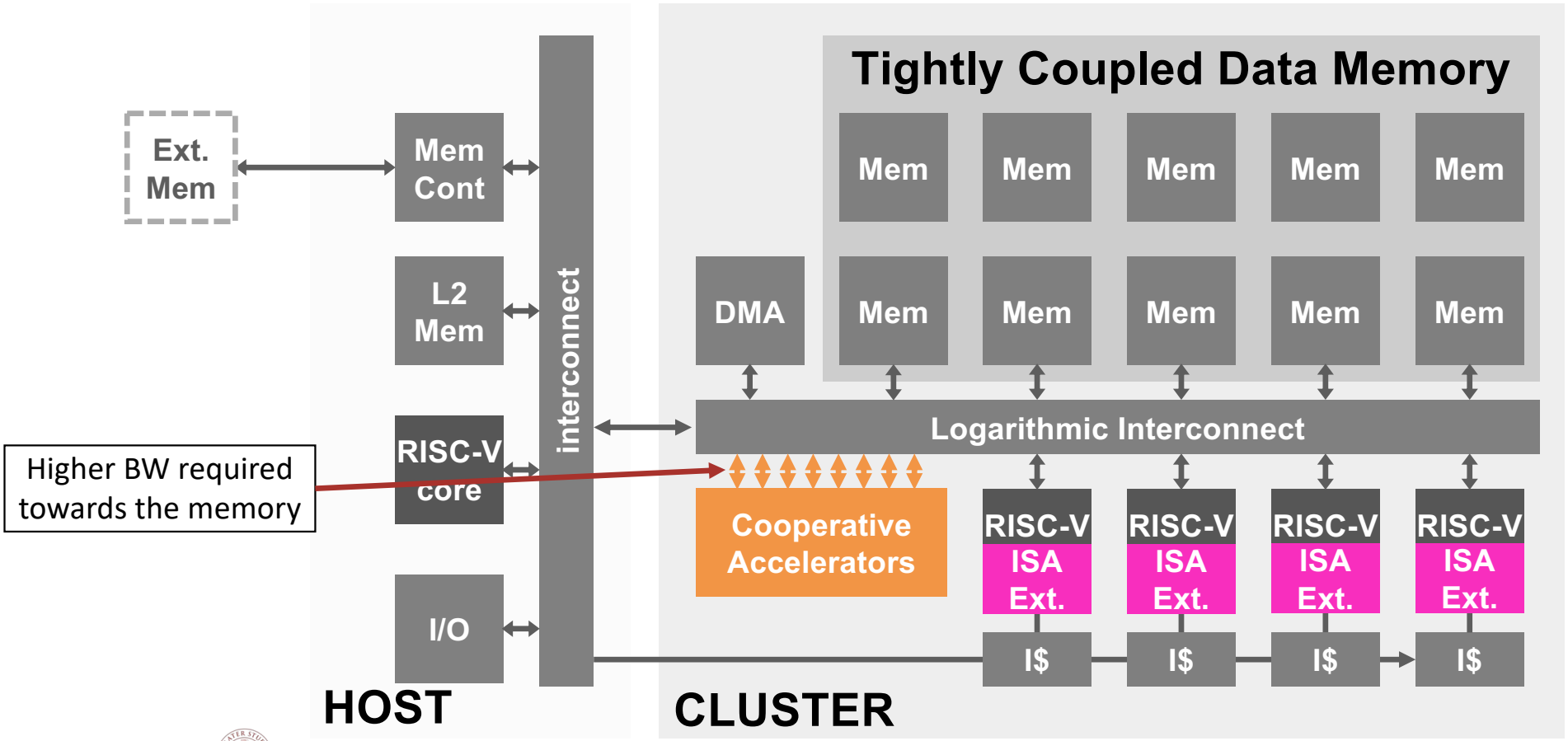


[Jia et al. JSSC 2020]



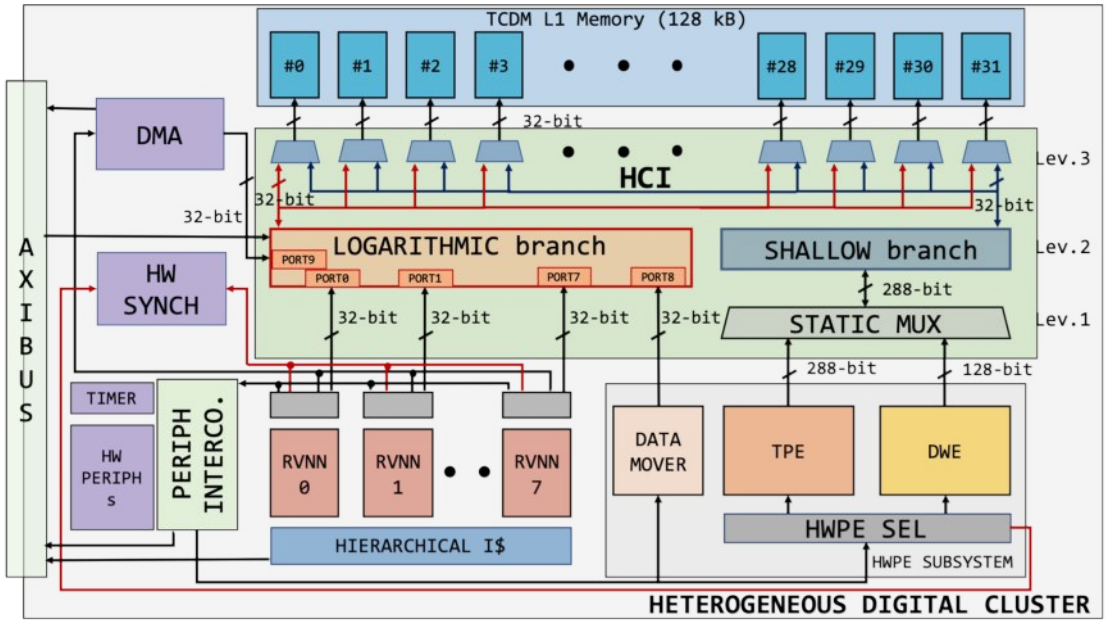
[Rossi et al. JSSC 2022]

PULP as a template for Heteorgeneous Parallel SoC

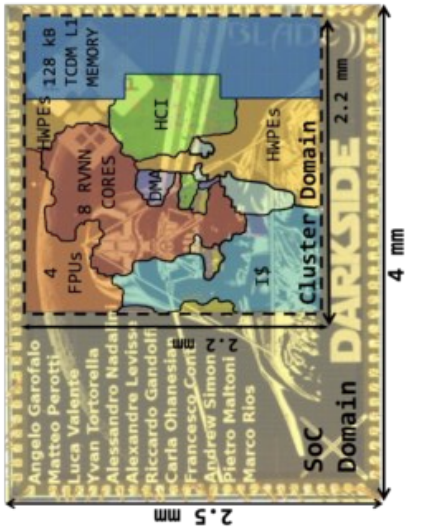


PULP Tightly-Coupled Heterogeneous Accelerators

- **The Darkside use-case**



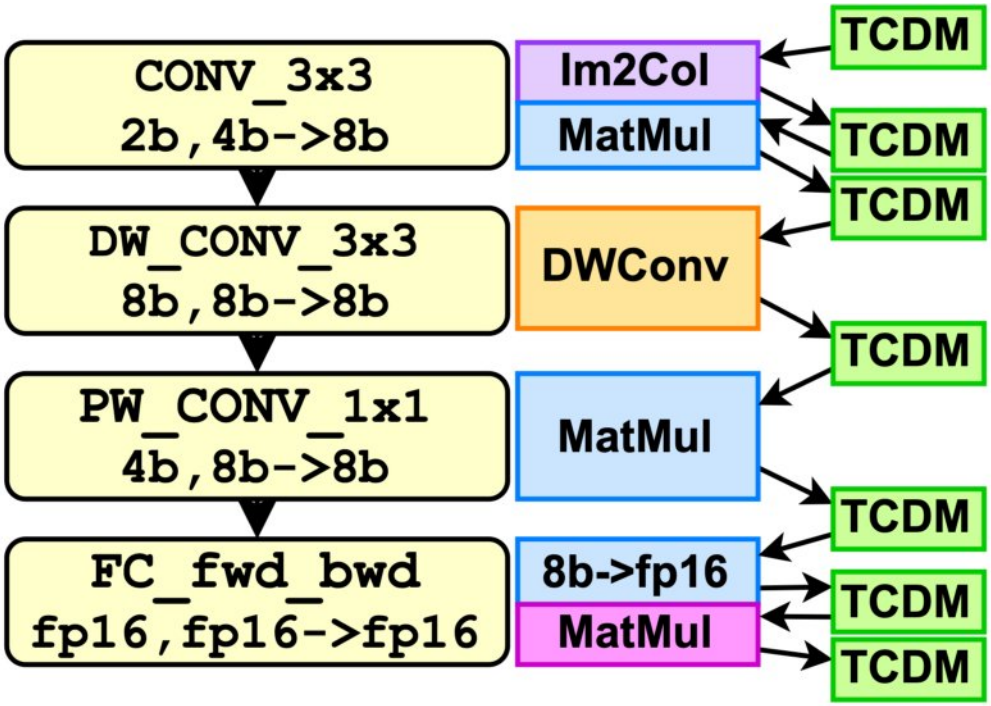
SoC layout, 65nm



[Garofalo et al. OJSSCS, 2022]

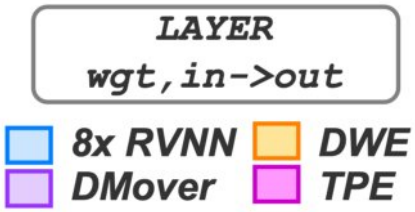
- Acceleration with flexibility: zero-copy HW/SW cooperation
- Many tightly-coupled accelerators for Amdahl's effects mitigation
- HCI: Low-Latency (one clk cyc), High-Bandwidth (76B/cyc)

Darkside's AI Execution Model

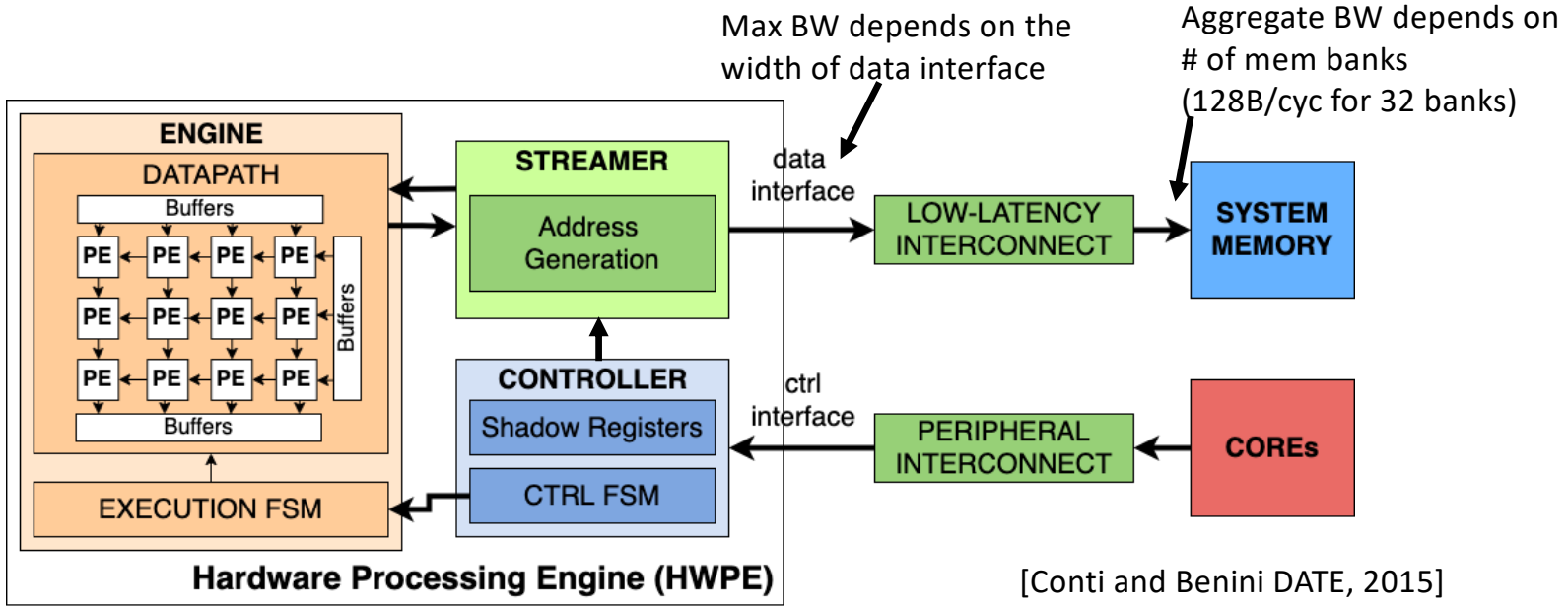


How do we handle:
1) High Bandwidth access to TCDM? (Interco)
2) Fast synchronization of the workload?

- TCDM as tightly-coupled memory buffer for all the compute units of the cluster:
- Efficient cooperation among hardware compute units;
- Support complex ML and DL execution models (e.g. full MobileNetV2, FC Autoencoder);



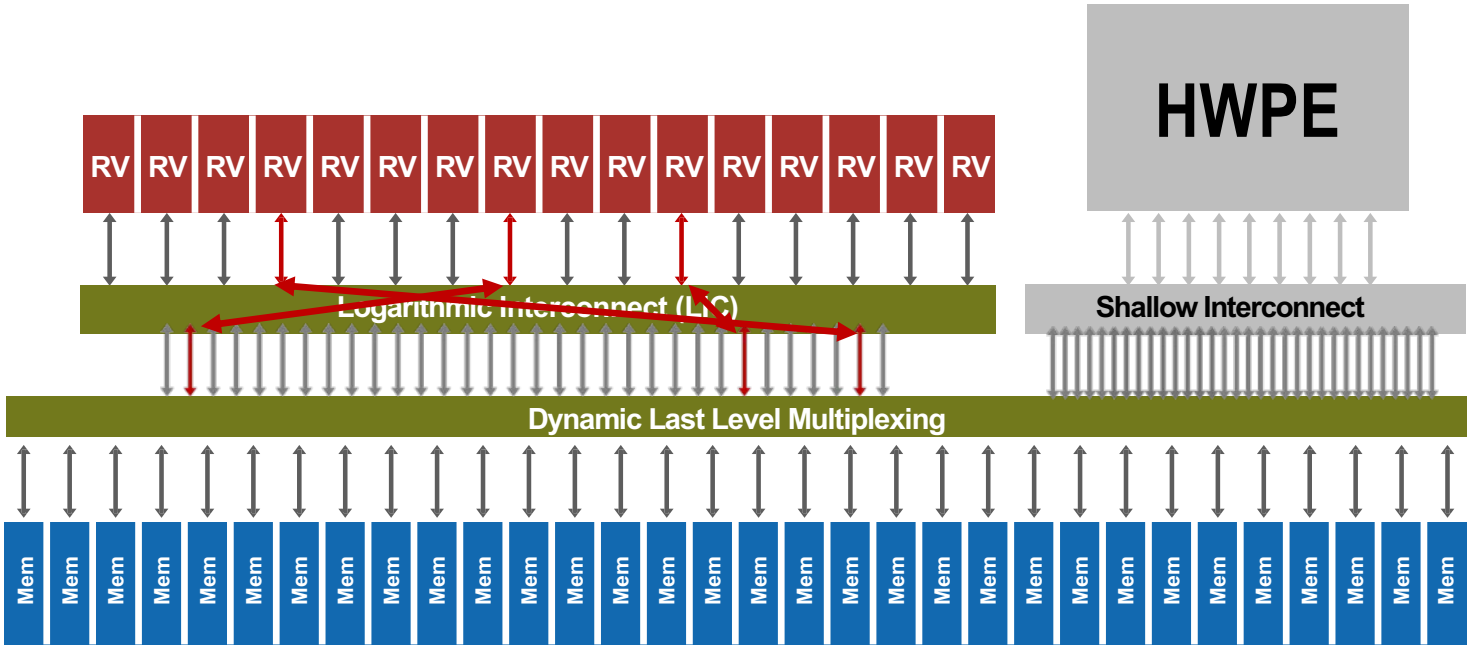
The Hardware Processing Engines (HWPEs)



[Conti and Benini DATE, 2015]

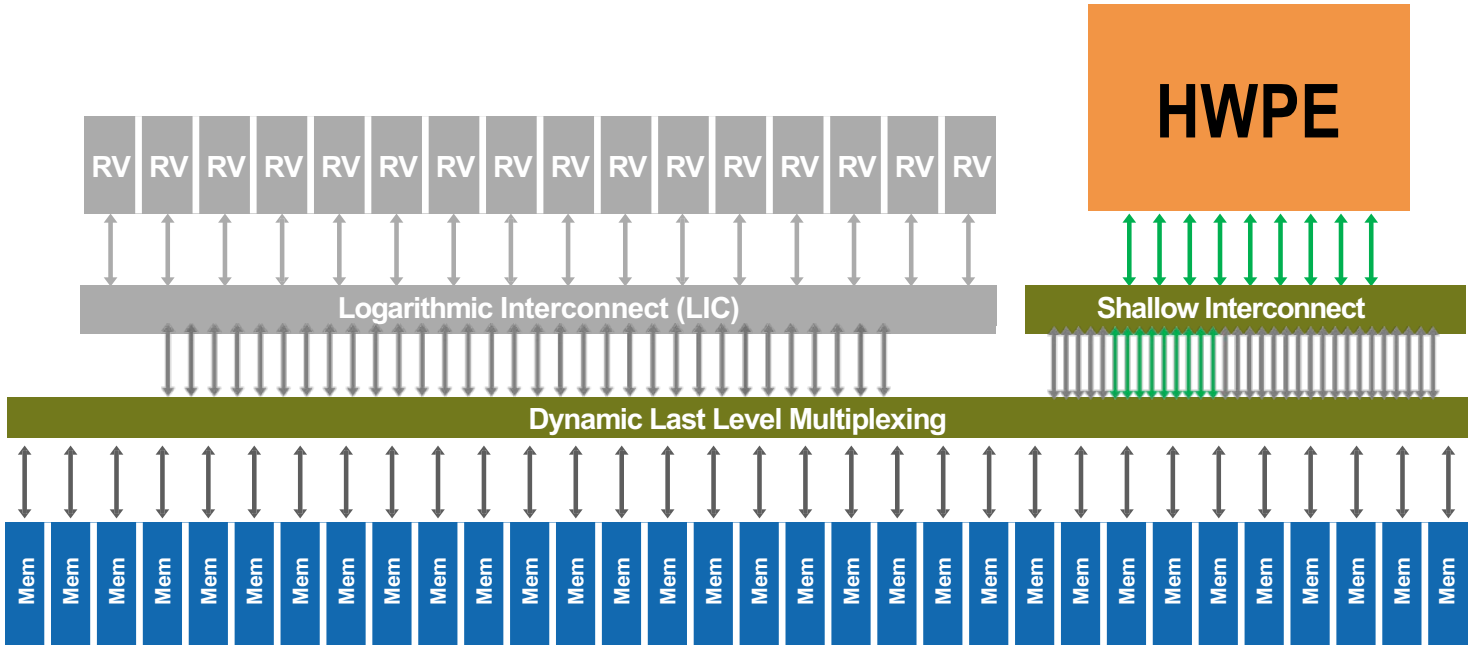
- HWPE efficiency:
  - Specialized compute datapath & minimal internal storage (e.g. line buffers)
  - Specialized high-BW interconnect towards L1 buffer (on data interface)
  - Dedicated control (no I-fetch) with shadow registers (overlapped config-exec)

The Heterogeneous Cluster Interconnect



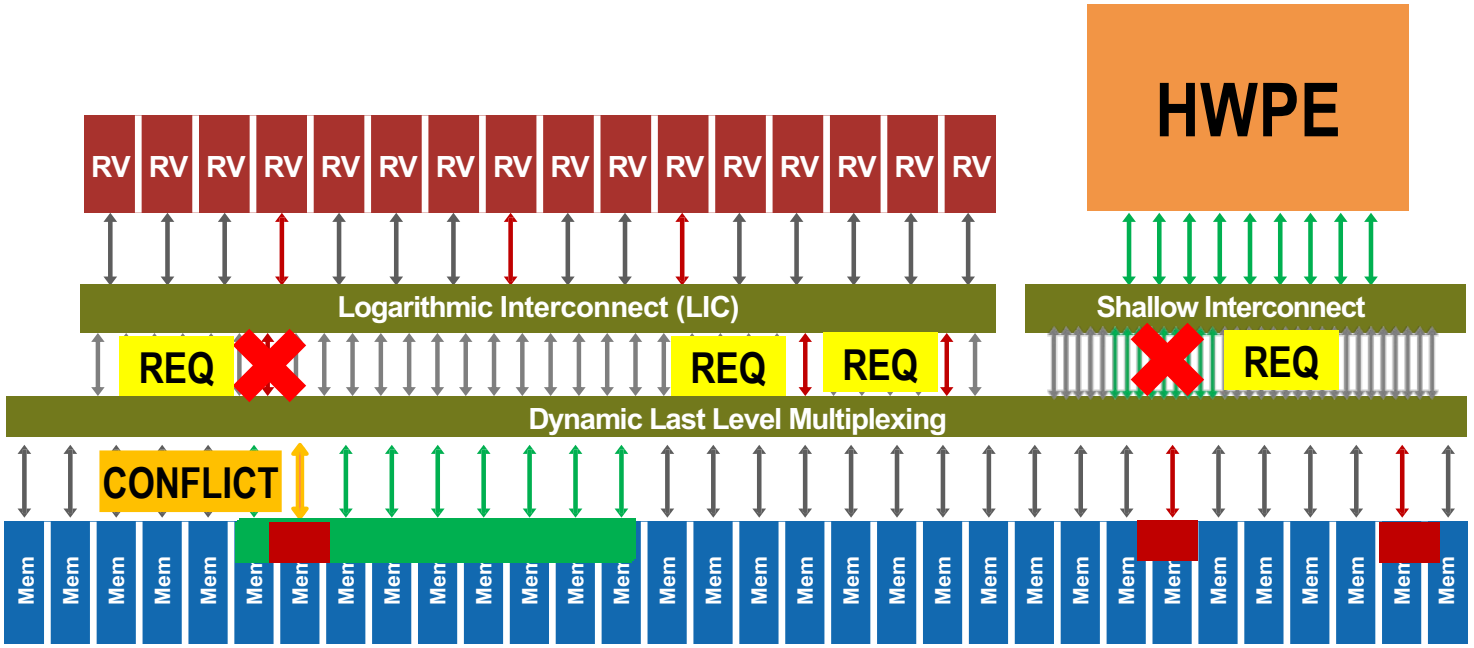
- Processors with their narrow memory ports are arbitrated in the LIC

The Heterogeneous Cluster Interconnect - 2



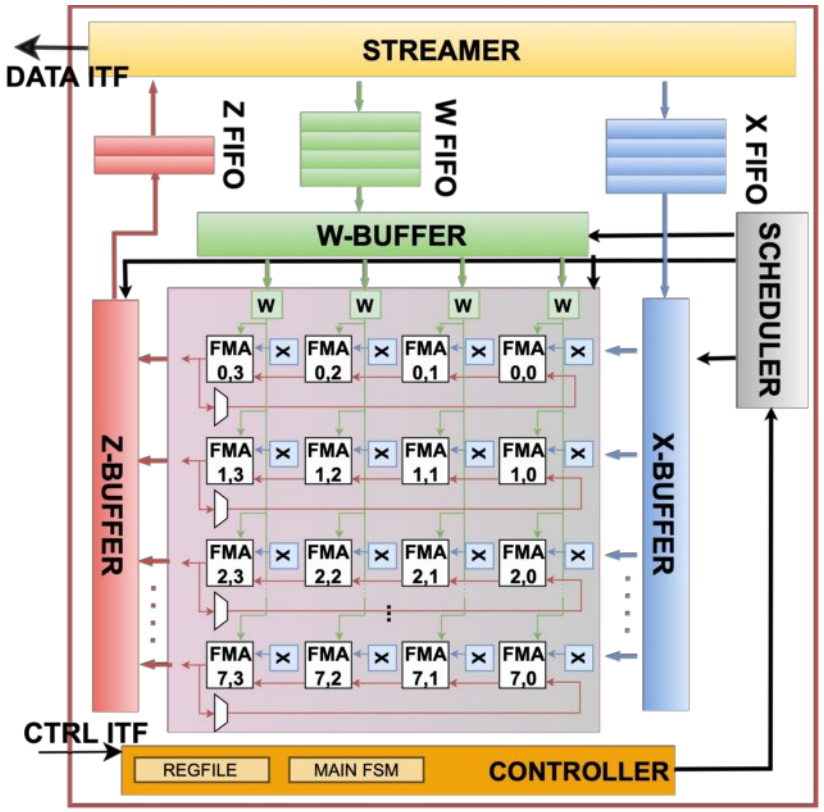
- HWPE exposes a unified high-bandwidth port (e.g. 256-bit) towards the shallow interconnect (simple ”wide” port reordering block without self-contention)

The Heterogeneous Cluster Interconnect - 3



- Final level dispatches accesses to single 32-bit memory banks and manages conflicts by means of rotating configurable priority (e.g. max 3 stall cycles);
- If no conflicts, HWPEs and cores can access concurrently the memory.

The Tensor Product Engine (RedMule)

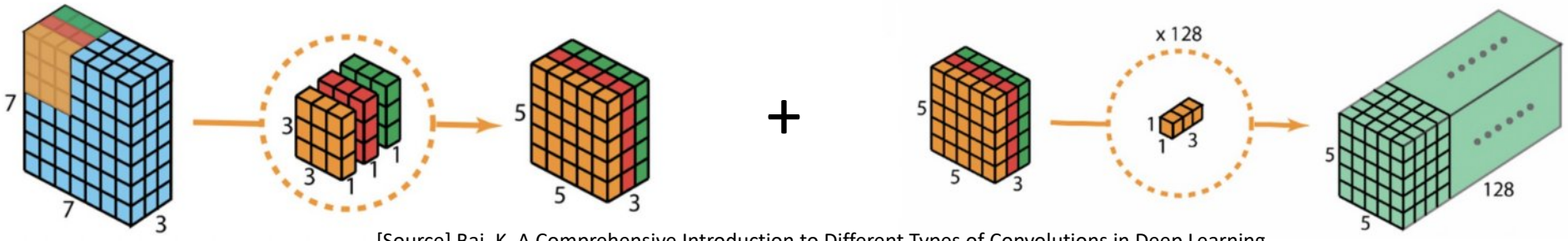


- **Boost IEEE FP16 Matrix Multiplications;**
- **Datapath:**
  - Scalable array of FMA Units, up to 8x8;
  - FMAs cascaded along the rows;
  - Trade-off between performance and area;
- **Execution scheduling optimized:**
  - Streaming always overlaps computation;
  - Data reuse is maximized;
- **98% of FMAs Utilization;**
- **Near-to-ideal performance: 189 OPs/cycle (ideal is 192 OPs/cycle) in 12x4 config;**
- **Up to 22x Speed-Up over SW MatMuls execution;**
- **292 kGE for a 12x4 configuration (>4x xpulpnn core area!).**

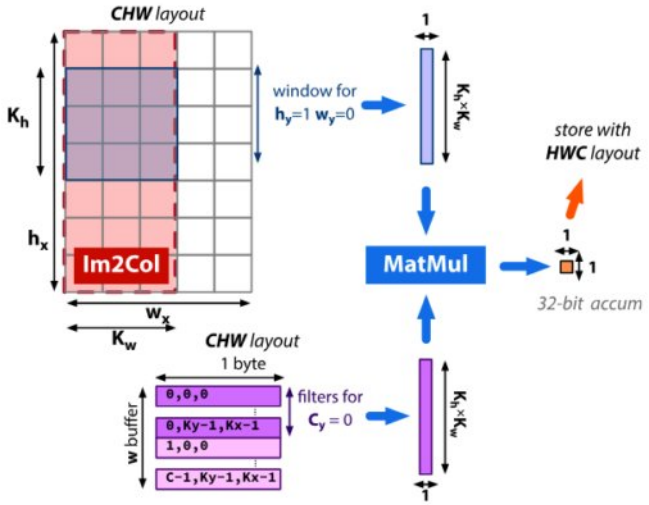
Use-case: The Depthwise Convolutions of MobileNets

Depthwise Separable Convolution

Pointwise (1x1) Convolution



[Source] Bai, K. A Comprehensive Introduction to Different Types of Convolutions in Deep Learning.

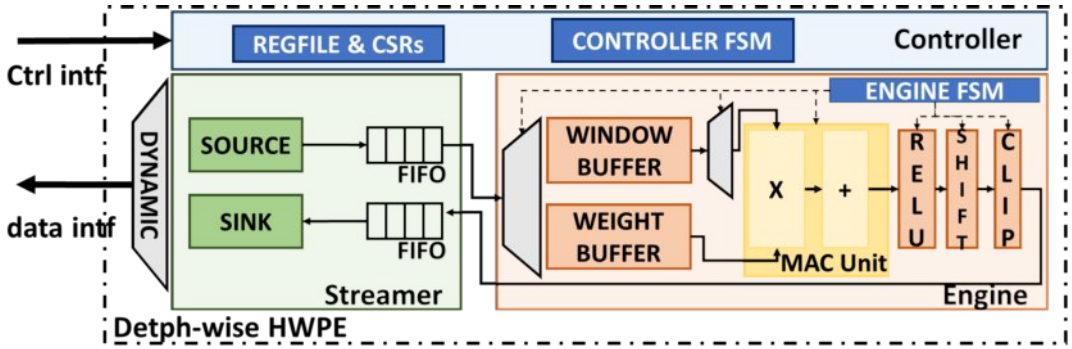


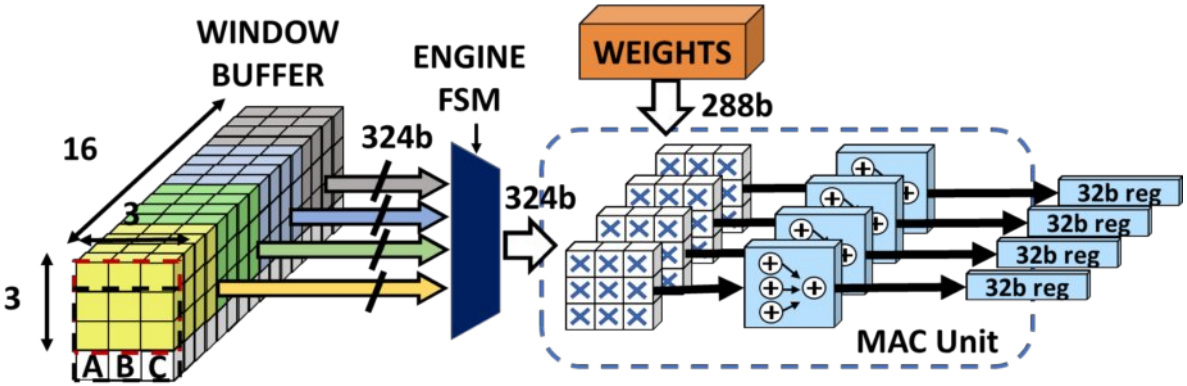
- **HWC layout not efficient** for Depthwise convolutions
- → no operations along channels
- PULP-NN integrated a **CHW-based implementation** of depthwise
- → very low performance
- Better to delegate this workload to a digital function-specific accelerator

ETH zürich

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Accelerating DepthWise With Custom Engine

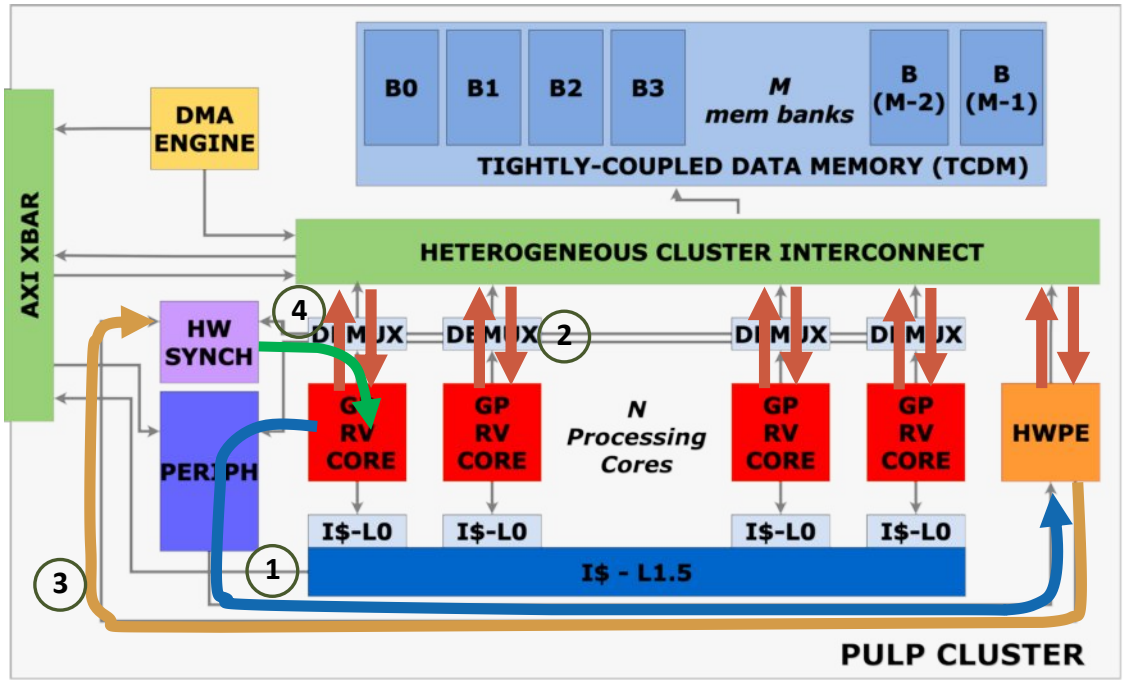




Depth-Wise Engine (DWE):

- Boost low-reuse 8-bit (integer) 3x3 Depth-wise convolutions;
- It works on HWC layout (no need to reshuffle data!)
- Weight-Stationary Data Flow to maximize data reuse;
- It automatically performs 8-b requantization on the outputs
- Fully exploit memory bandwidth of **36B/cyc** through *shallow* HCI branch;
- Peak performance of **30 MAC/cycle**;
- **131 kGE (2x xpulpnn core area)**.

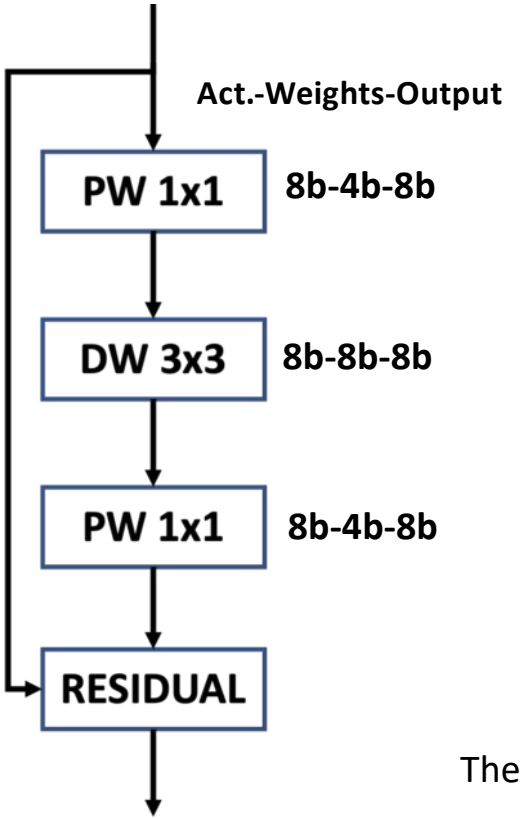
HW Synchronizer for Cores, DMAs, Accelerators

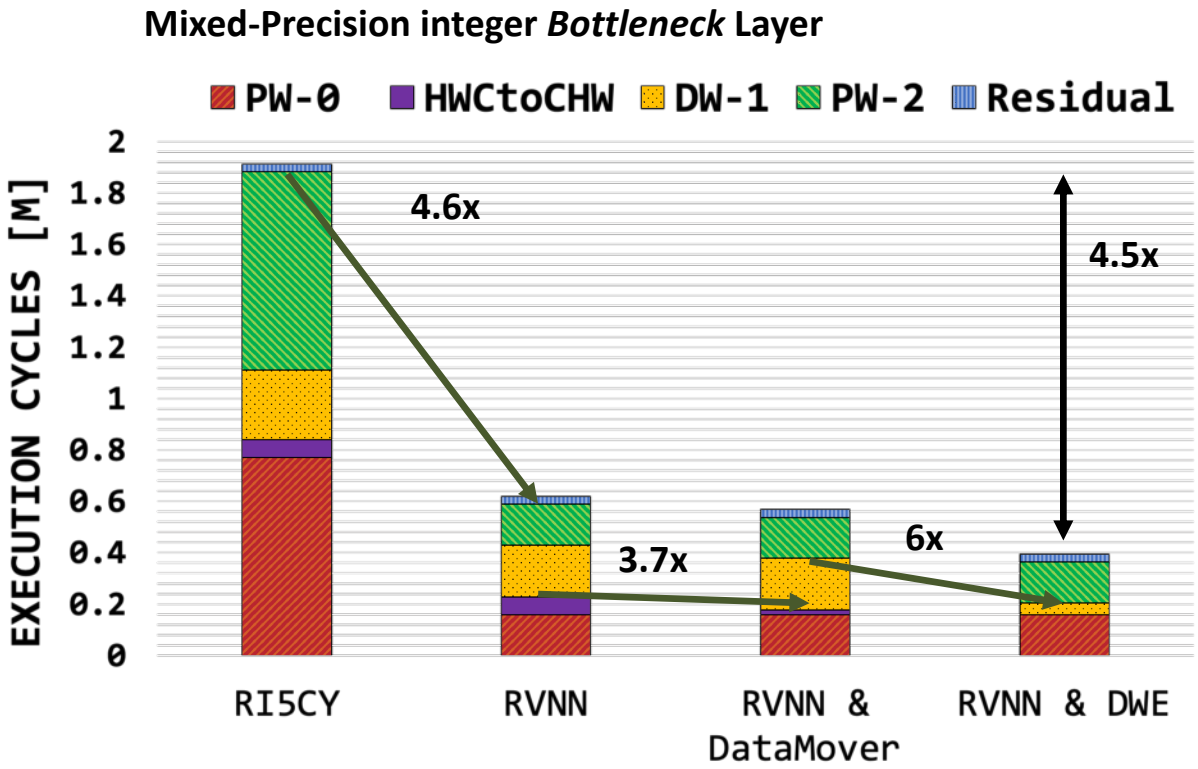


[Glaser et al. IEEE TPDS 2020]

- Fast and energy-efficient thread dispatching and synchronization
- Automatic clock gating of PEs waiting on a barrier
- Event-based computation (between accelerators/DMAs and GP cores)

MobileNetV2 Execution on Darkise



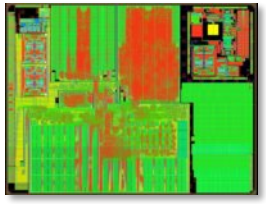


The Heterogeneous approach mitigates Amdahl's effects on heterogeneous workloads

Tightly-Coupled Acceleration: Success Stories

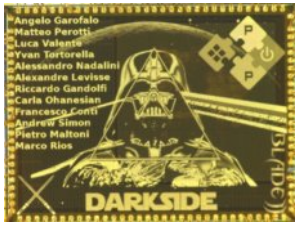
GREENWAVES TECHNOLOGIES

GAP9 SoC with **NE16**
(commercial)

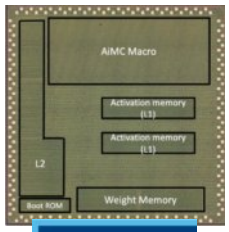


DOLPHIN INTEGRATION
The enabler of low-power Systems-on-Chip

Marsellus SoC with **RBE**
ISSCC 2023

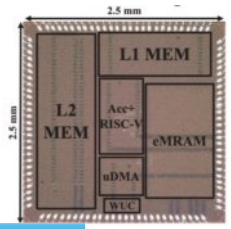


Darkside SoC with many HWPEs:
transposer, systolic array, depth-wise engine..
[Garofalo et al. ESSCIRC 2022]



KU LEUVEN

Diana SoC with **AIMC**
[Ueyoshi et al. ISSCC 2022]



KU LEUVEN MAGICS

TinyVers SoC with **HWPE**
[Jain et al. IEEE VLSI 2022]

Not Only Academia: **GAP9** with **NE16**

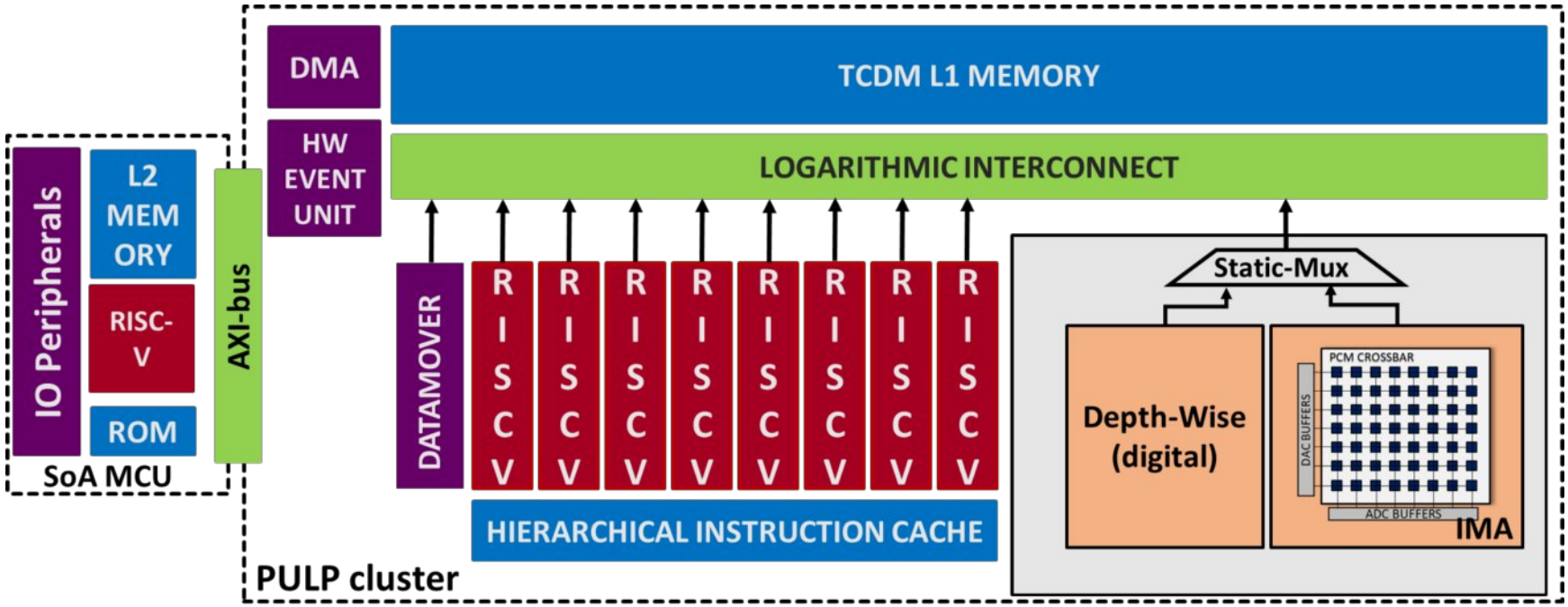
GREENWAVES TECHNOLOGIES

- **Best-in-class in latency and energy efficiency in MLPerf Tiny 1.0!**

							Benchmark Results									
							Task	Visual Wake Words		Image Classification		Keyword Spotting		Anomaly Deteciton	
							Data	Visual Wake Words Dataset		CIFAR-10		Google Speech Commands		ToyADMOS (ToyCar)	
							Model	MobileNetV1 (0.25x)		ResNet-V1		DSCNN		FC AutoEncoder	
							Accuracy	80% (top 1)		85% (top 1)		90% (top 1)		0.85 (AUC)	
Submitter	Board Name	SoC Name	Processor(s) & Number	Accelerator(s) & Number	Software	Notes	Units	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ
Greenwaves Technologies	GAP9 EVK	GAP9	RISC-V Core (1+9)	NE16 (1)	GreenWaves GAPFlow	GAP9 (370MHZ, 0.8Vcore)		1.13	58.4	0.62	40.4	0.48	26.7	0.18	7.29
Greenwaves Technologies	GAP9 EVK	GAP9	RISC-V Core (1+9)	NE16 (1)	GreenWaves GAPFlow	GAP9 (240MHZ, 0.65Vcore)		1.73	40.8	0.95	27.7	0.73	18.6	0.27	5.25
OctoML	NRF5340DK	nRF5340	Arm® Cortex®-M33		microTVM using CMSIS-NN backend	128MHz		232.0		316.1		76.1		6.27	
OctoML	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		microTVM using CMSIS-NN backend	120MHz, 1.8Vbat		301.2	15531.4	389.5	20236.3	99.8	5230.3	8.60	443.2
OctoML	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		microTVM using native codegen	120MHz, 1.8Vbat		336.5	7131.6	389.2	21342.3	144.0	7950.5	11.7	633.7
Plumerai	B_U585I_IOT02A	STM32U585	Arm® Cortex®-M33		Plumerai Inference Engine 2022.09	160MHz		107.0		107.1		35.4		4.90	
Plumerai	CY8CPROTO-062-4343w	PSoC 62 MCU	Arm® Cortex®-M4		Plumerai Inference Engine 2022.09	150MHz		192.5		193.1		61.4		6.70	
Plumerai	DISCO-F746NG	STM32F746	Arm® Cortex®-M7		Plumerai Inference Engine 2022.09	216MHz		57.0		64.8		19.1		2.30	
Plumerai	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		Plumerai Inference Engine 2022.09	120MHz		208.6		173.2		71.7		5.60	
Silicon Labs	xG24-DK2601B	EFR32MG24	Arm® Cortex®-M33	Silicon Labs MVP(1)	TensorFlowLite for Microcontrollers, CMSIS-NN, Silicon Labs Gecko SDK			111.6	1139.2	120.9	1234.7	36.3	401.9	5.43	47.3
STMicroelectronics	NUCLEO-H7A3ZI-Q	STM32H7A3ZIT6Q	Arm® Cortex®-M7		X-CUBE-AI v7.3.0	280MHz, 3.3Vbat		50.7	7978.1	54.3	8707.3	16.8	2721.8	1.82	266.5
STMicroelectronics	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		X-CUBE-AI v7.3.0	120MHz, 1.8Vbat		230.5	10066.6	226.9	10681.6	75.1	3371.7	7.57	323.0
STMicroelectronics	NUCLEO-U575ZI-Q	STM32U575ZIT6Q	Arm® Cortex®-M33		X-CUBE-AI v7.3.0	160MHz, 1.8Vbat		133.4	3364.5	139.7	3642.0	44.2	1138.5	4.84	119.1
Syntiant	NDP9120-EVL	NDP120	M0 + HiFi	Syntiant Core 2 (98MH	Syntiant TDK	Syntiant Core 2 (98MHz, 1		4.10	97.2	5.12	139.4	1.48	43.8		
Syntiant	NDP9120-EVL	NDP120	M0 + HiFi	Syntiant Core 2 (30MH	Syntiant TDK	Syntiant Core 2 (30MHz, 0		12.7	71.7	16.0	101.8	4.37	31.5		
Qualcomm Innovation Center	Next Generation Snapdragon Mobile Platform HDK	Next Generation Snapdragon Mobile Platform	Qualcomm Kryo CPU(1)	Qualcomm Sensing Hub(1)	Qualcomm AI Stack										

Latency 1.73ms (against ~29ms on GAP8)
Energy 41uJ (against 1450uJ on GAP8)

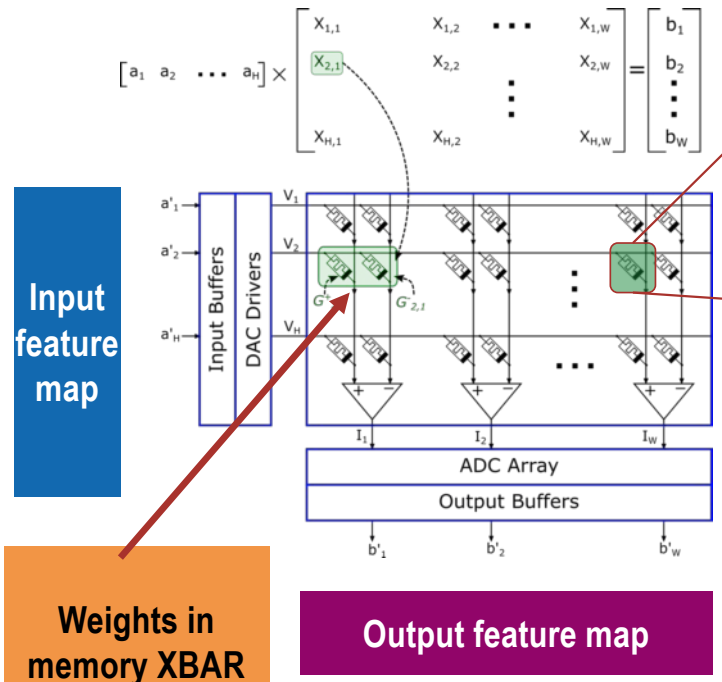
The Analog/Digital PULP Cluster

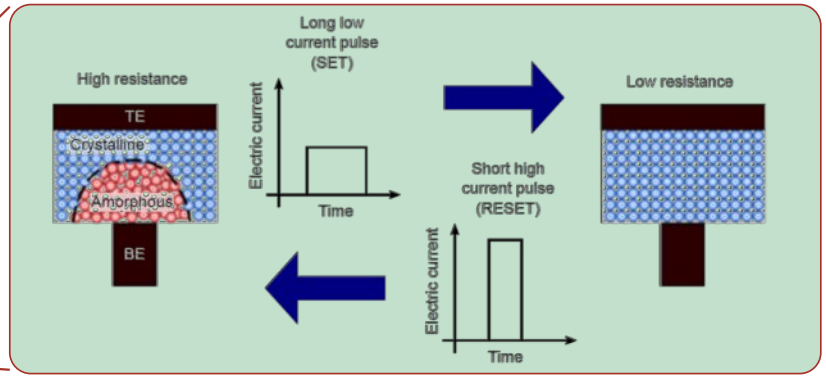


A. Garofalo et al., "A Heterogeneous In-Memory Computing Cluster For Flexible End-to-End Inference of Real-World Deep Neural Networks," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, doi: 10.1109/JETCAS.2022.3170152.

Emerging Technologies for Deep AI Acceleration

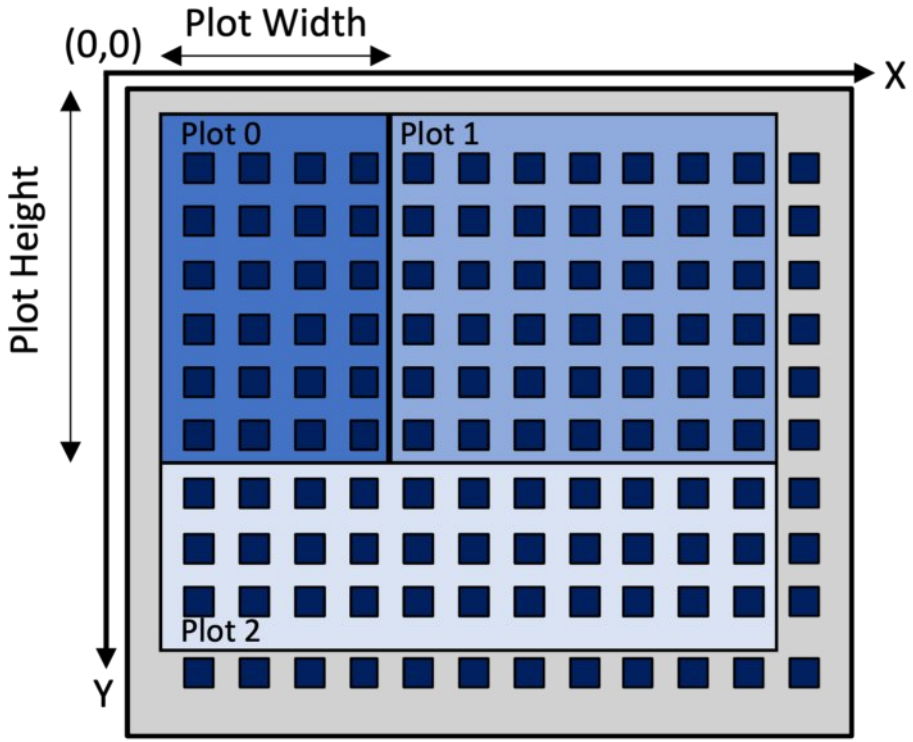
- **Analog In-Memory Computing (A-IMC) based on Phase Change Memory (PCM)**





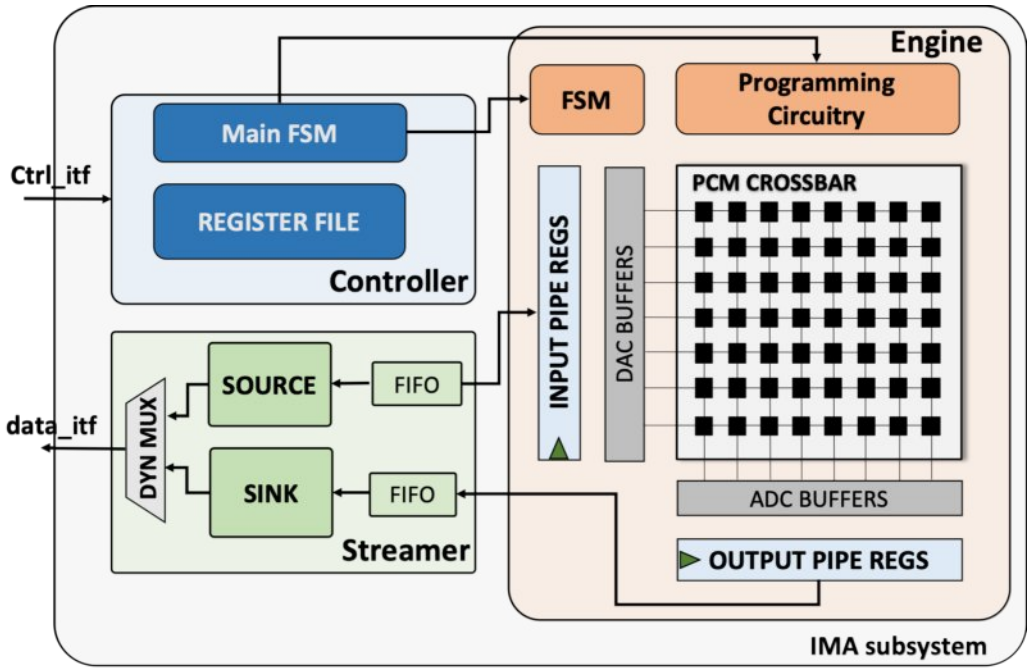
- An Efficient way to implement many parallel MAC operations found in DNNs
- Store weights in conductance elements $G_{N\#}$
  - Many options available for $\mathbf{G_{N\#}}$ → Typical Equivalent Precision: 4-bit
- Relies on analog computation "nearly free"
  - Multiplication (e.g., $\mathbf{I_{A0} = G_{A0} * V_0}$) and
  - Addition (e.g. $\mathbf{I_{A0} + I_{A1} + I_{A2}}$)

Challenge: Weights Mapped Statically



- Weights have to be mapped statically
- Application Cannot be changed on the fly
- Plot configuration allows selecting a sub-set of the IMA array (e.g. a layer of a DNN).
- Increases average utilization (multiple layes can fit a single array).

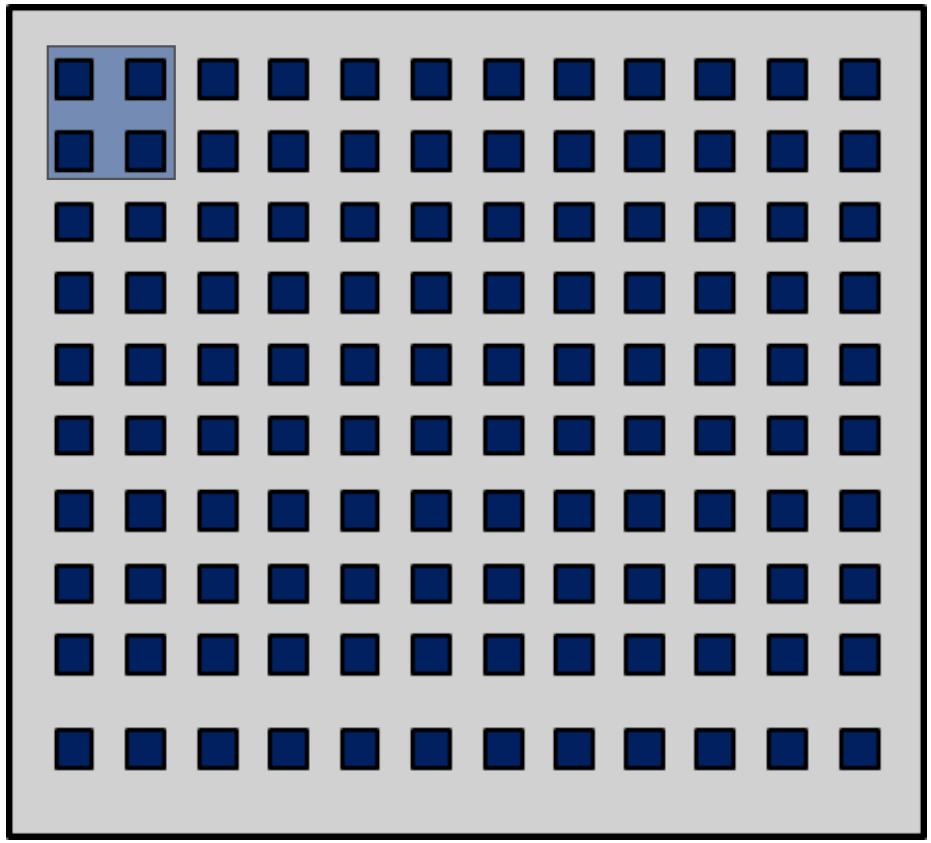
Heterogeneous In-Memory Computing Cluster



[Ottavi et al. AICAS, 2021]
[Khaddam et al. JSSCC, 2022]

- **256x256** PCM-based IMC array
- Weights precision: **4-bit;** Input/Output precision: **8-bit**
- **The IMC runs in its own clock domain and has a fixed exec latency (independently of the # of operations we perform)**
- MVM operations: 130K in 130ns
  - Peak performance **1TOPS**
- Peak Efficiency **11.7 TOPS/W**
- Integrated into HWPE
- Data_itf sized to sustain BW requirement of the IMC core

Challenge: Performance Depends on Utilization!



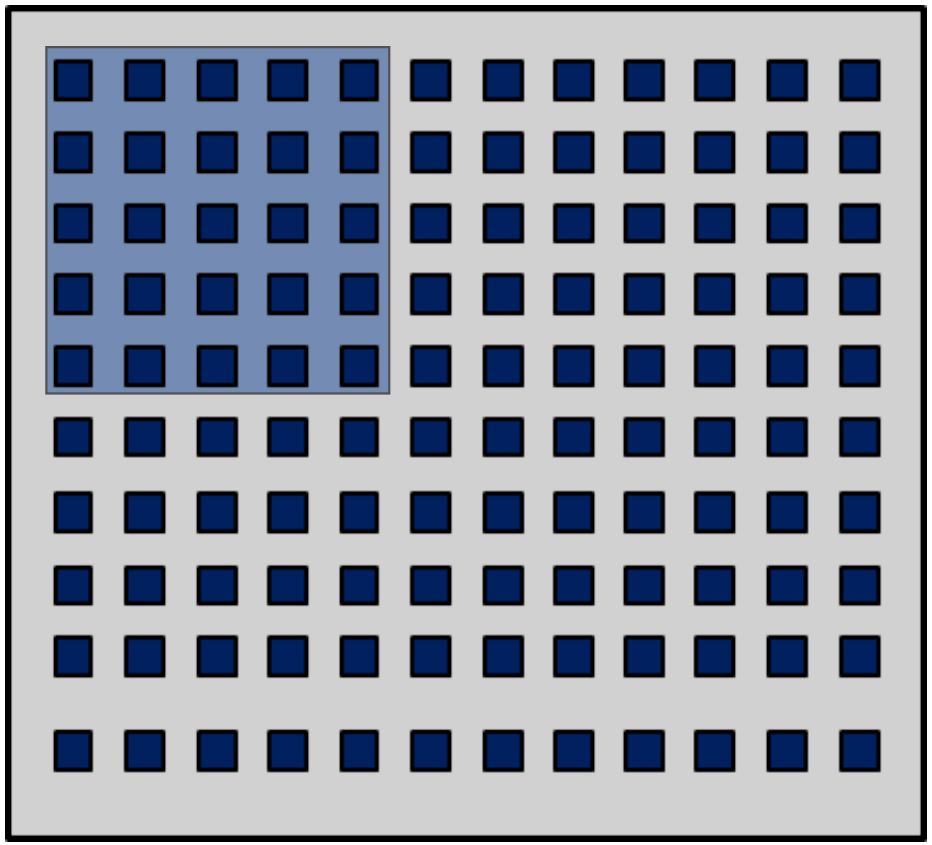
Numerical Example

- Plot Size: 4x4
- Latency: 130ns (Fixed by tech)
- **Performance =**

  16 MAC/130ns =

  240 OP/s (1 MAC = 2 OP)

Challenge: Performance Depends on Utilization!



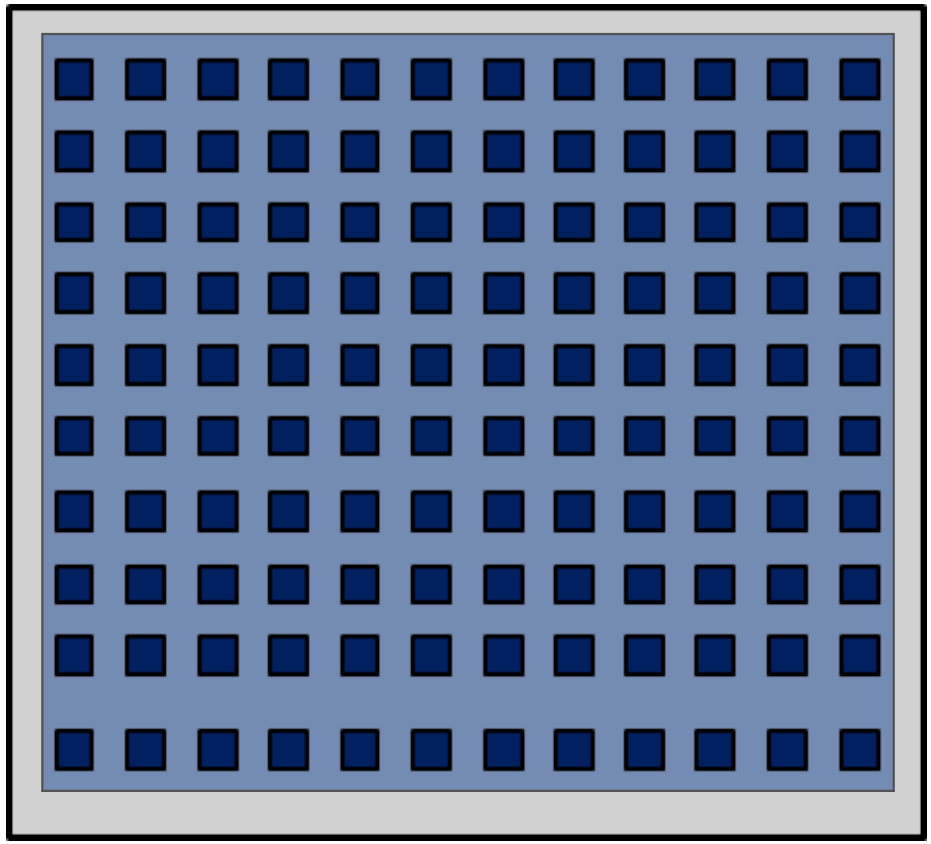
Numerical Example

- Plot Size: 16x16
- Latency: 130ns (fixed by tech)
- **Performance =**

  256 MAC/130ns =

  4000 OP/s (1 MAC = 2 OP)

Challenge: Performance Depends on Utilization!



Numerical Example

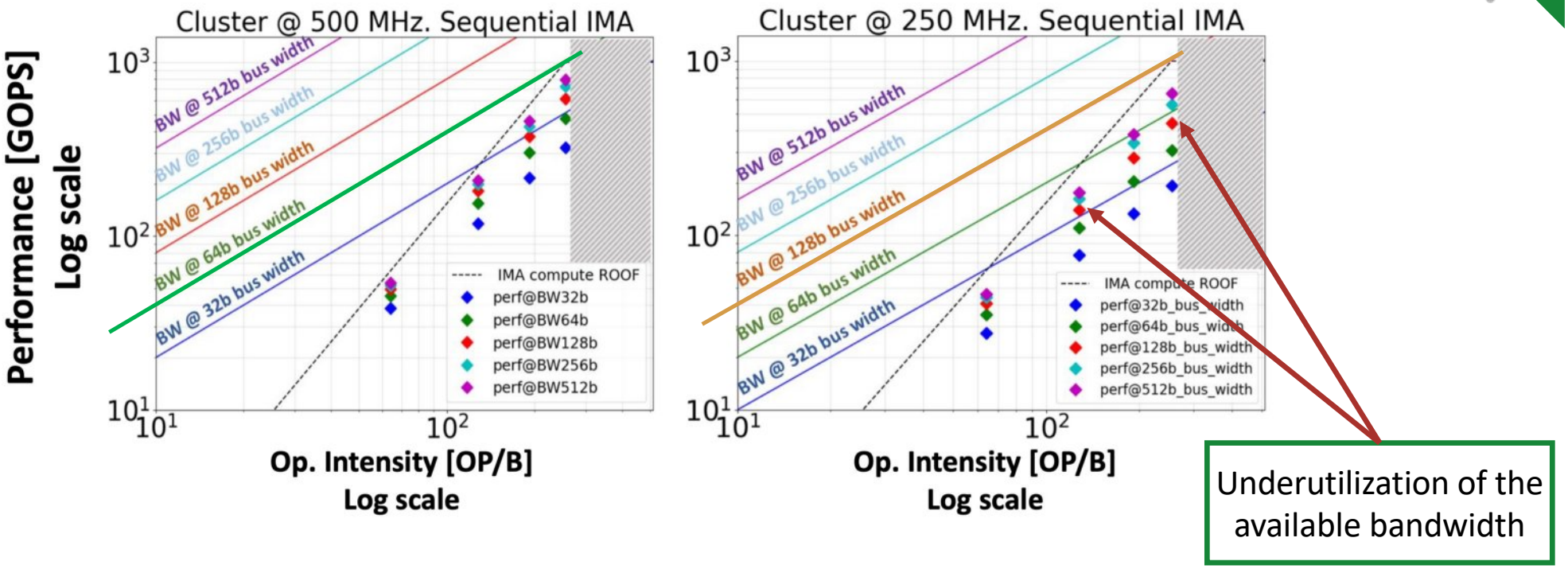
- Plot Size: 256x256
- Latency: 130ns (fixed by tech)
- **Performance =**

  65536 MAC/130ns =

  500 GMAC/s =

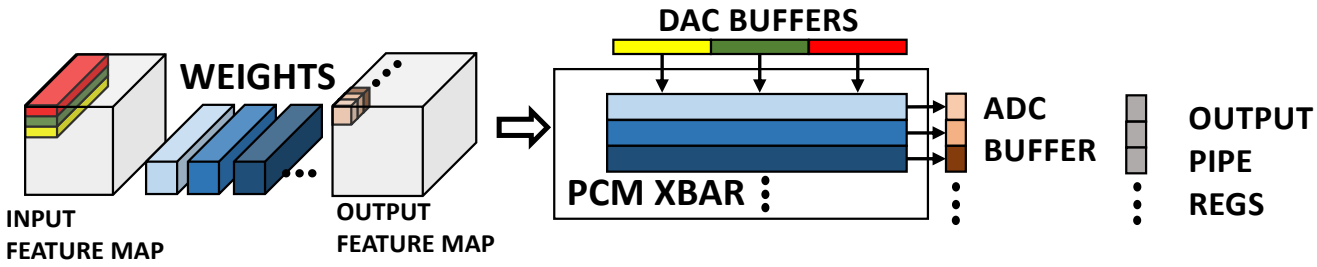
  1 TOP/s (1 MAC = 2 OP)

Roofline Plot Not Constant!

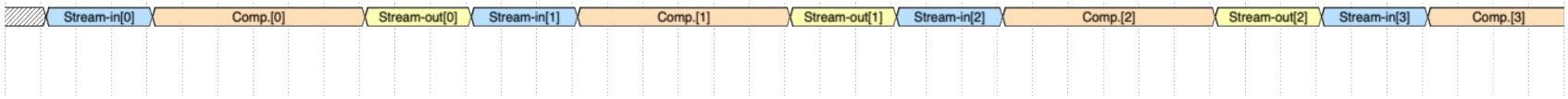


- Compute roof determined by MVM op. latency, XBAR size and array utilization
- BW determined by architecture and physical implementation of digital system

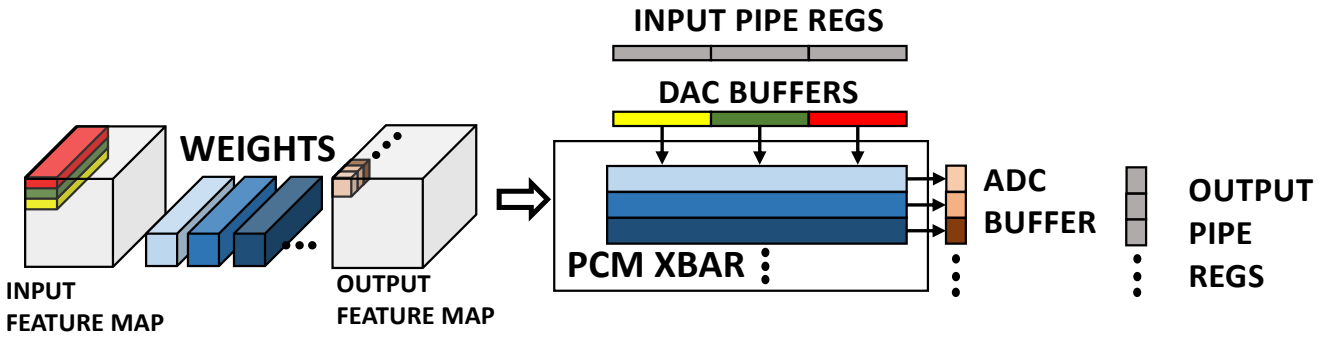
Pipelined Computational Model



Sequential Execution Model (Sub-Optimal)



Pipelined Computational Model

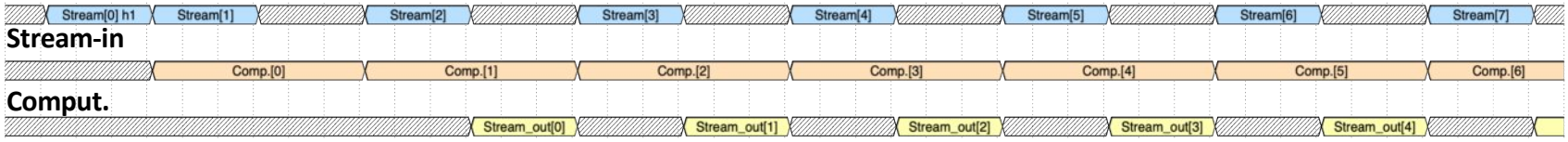


Pipelining:

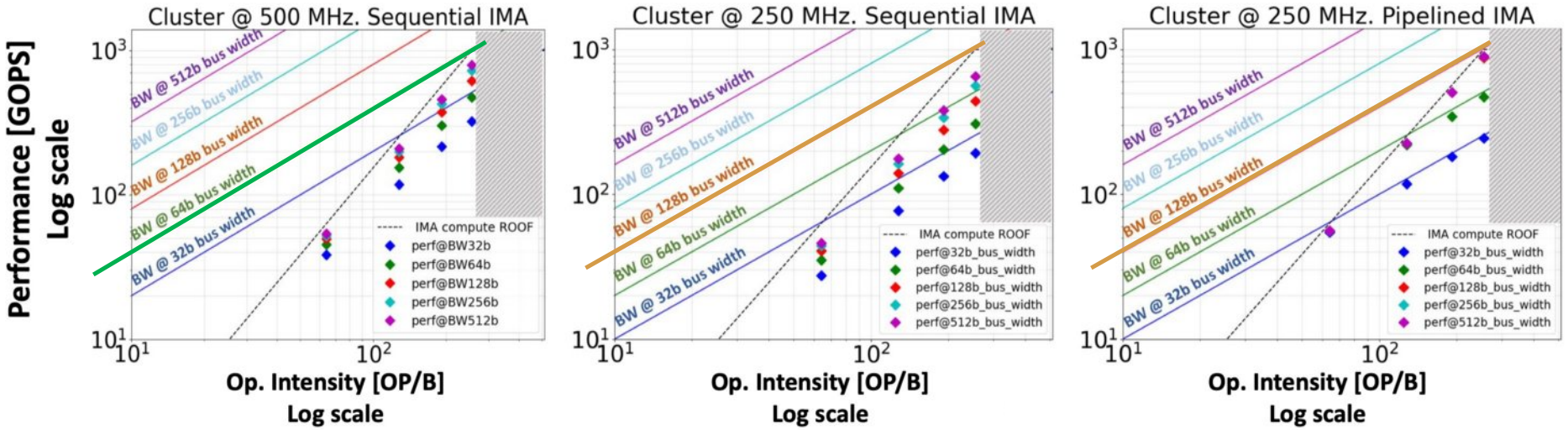
- While Job 0 being processed by IMA job 1 being loaded into input buffers
- Requires doubling IMA I/O registers

Sequential Execution Model (Sub-Optimal)

Pipelined Execution Model



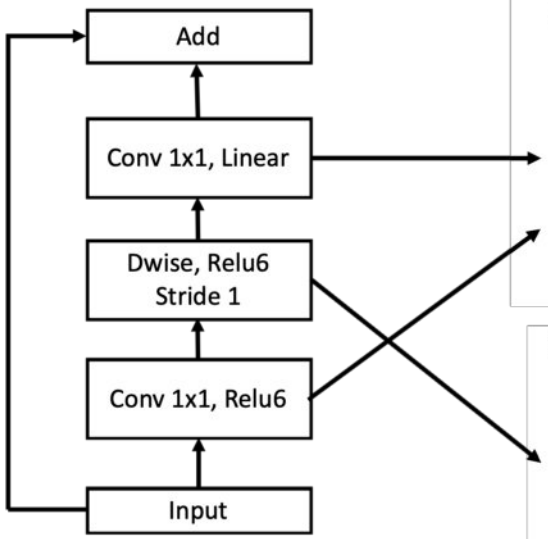
Integration Challenges Solved with Pipelining



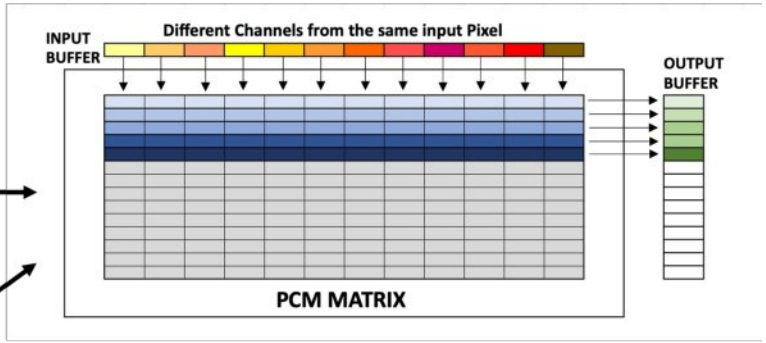
- Compute roof determined by MVM op. latency, XBAR size and array utilization
- BW determined by architecture and physical implementation of digital system
- Pipelined execution (overlap computation with streaming) to fully utilize BW

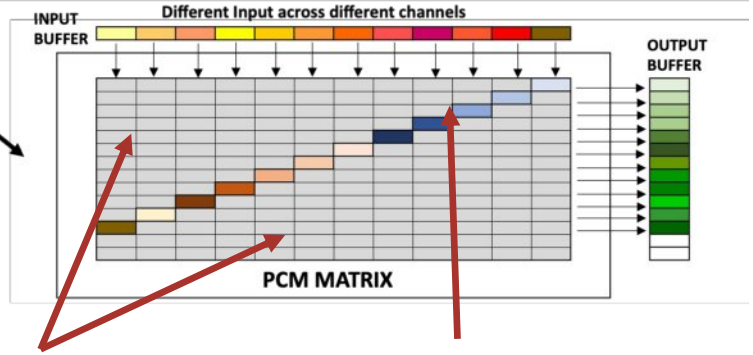
MobileNetV2 Bottleneck Layer Mapping on IMA

- **1x1 convol layer**
  - High data reuse
  - Efficiently mapped on IMA at high utilization
- **DW layer**
  - Not efficiently mappable on IMA
    - Lots of data padding (zeros) to adapt to DW exec flow
    - Drastic underutilization of the IMA
      - Wastes area, power
      - Reduces perf and energy efficiency
  - Heterogeneity needed!
    - Digital DW Engine!



Layer	Size	IN-CH	OUT-CH
Conv 1x1	28x28	32	192
Dwise	28x28	192	192
Conv 1x1	28x28	192	32

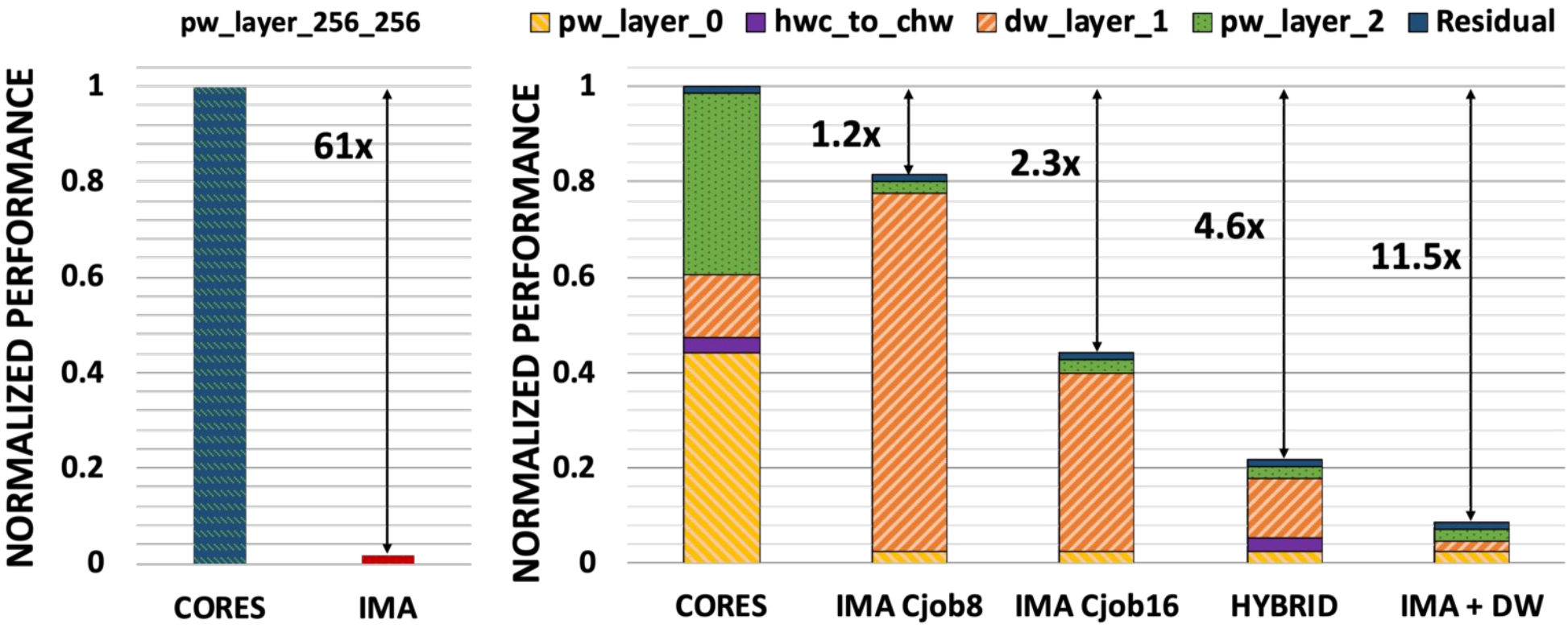




Zero padding

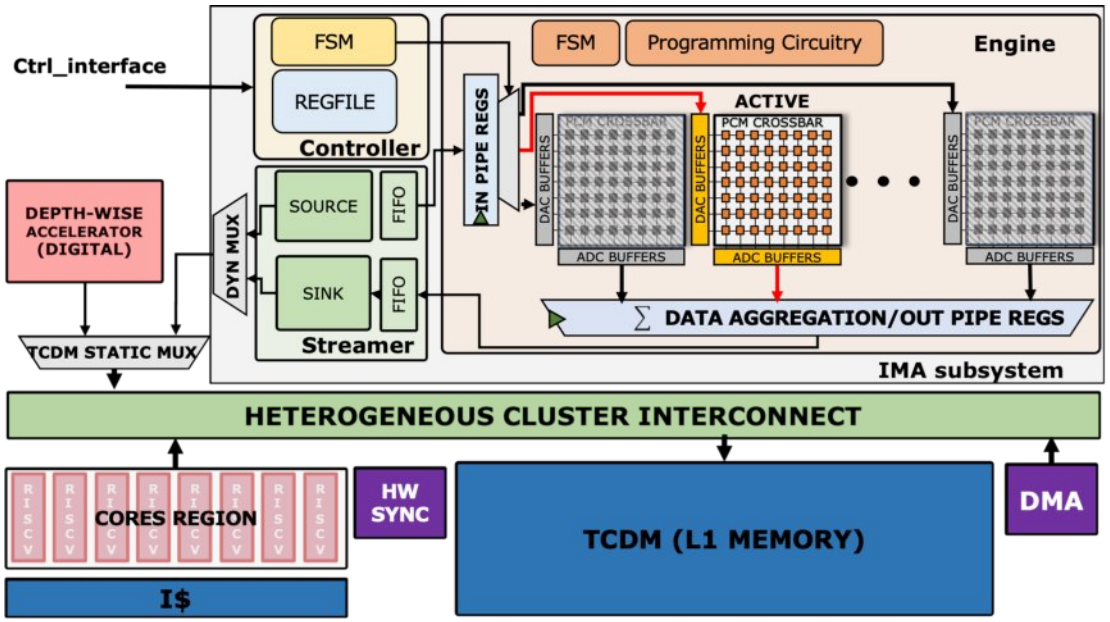
DW weights have to be "sparsly" mapped to IMA → not efficient!

Amdahl's Law Mitigation



Full MobileNetV2 Mapping

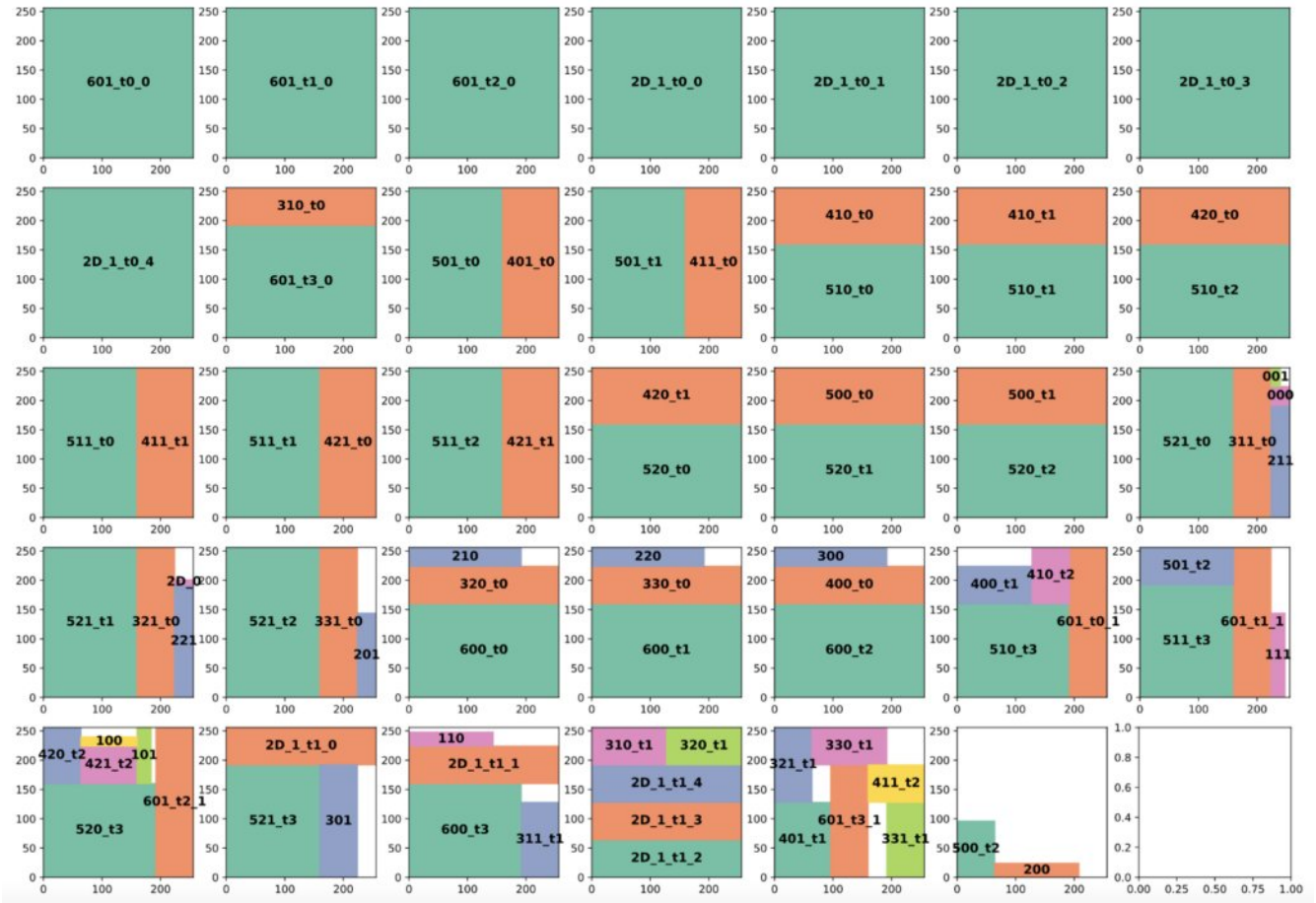
- **Due to PCM-based AIMC characteristics more AIMC tiles must be integrated into the cluster to execute a full MobileNetV2**
  - We need to store the entire model (pointwise + linear layers) in AIMC statically



- 34 AIMC Xbars, 25mm^2 layout in 22nm FDX technology
- Enough storage capacity to host a full MobileNetV2 (~1MB) execution
- Sequential execution model tailored for the needs of an advanced IoT device

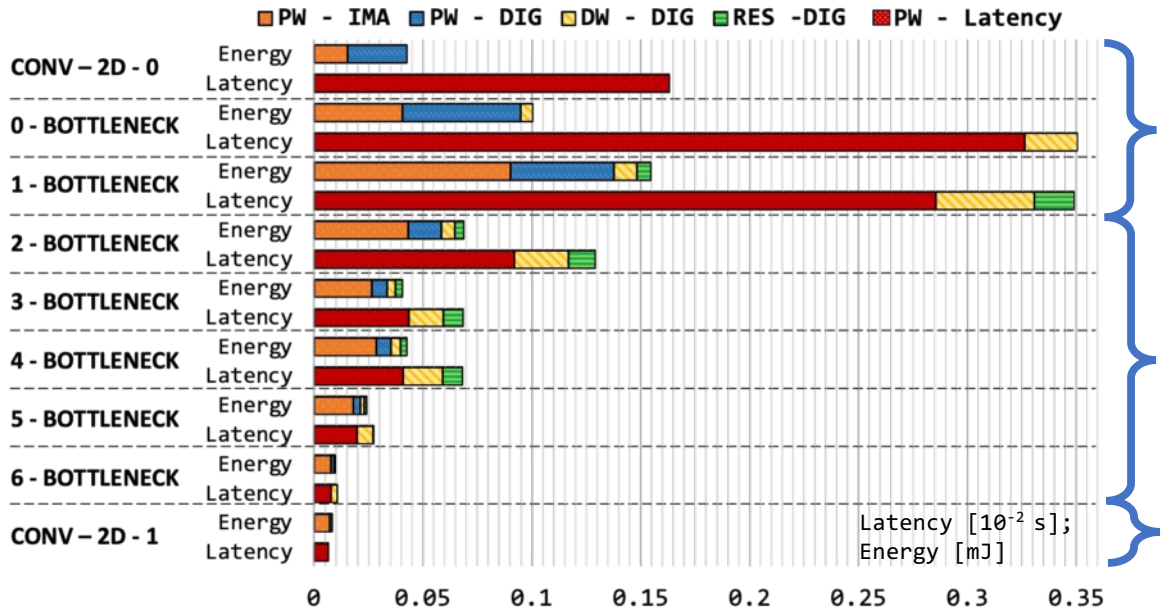
[Garofalo et al. IEEE JETCAS, 2022]

Full MobileNetV2 Mapping



- 34x 256x256 IMA Needed
- > 18 mm^2 in 22nm
- Large layers need to be split among several IMAs
- No Communication Overhead

End-To-End Latency and Energy

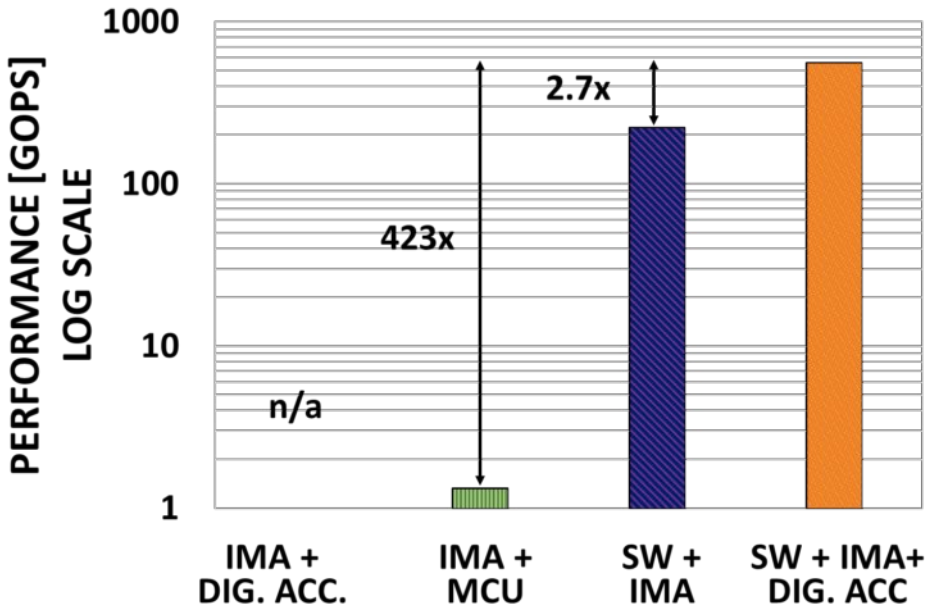


- Layers with few weights, but large input images: energy dominated by data transfers (input image data)
- Intermediate Bottleneck layers: significant DW and Residual workload
- Last compute intensive convolution layers: outstanding IMA acceleration

Demonstrated that AIMC alone cannot efficiently handle End-to-End Execution → Heterogeneous systems do! → RISC-V specialized processors as effective solutions to mitigate Amdhal's effects

Final Considerations on IMA-Centric Hetereogeneous Sys

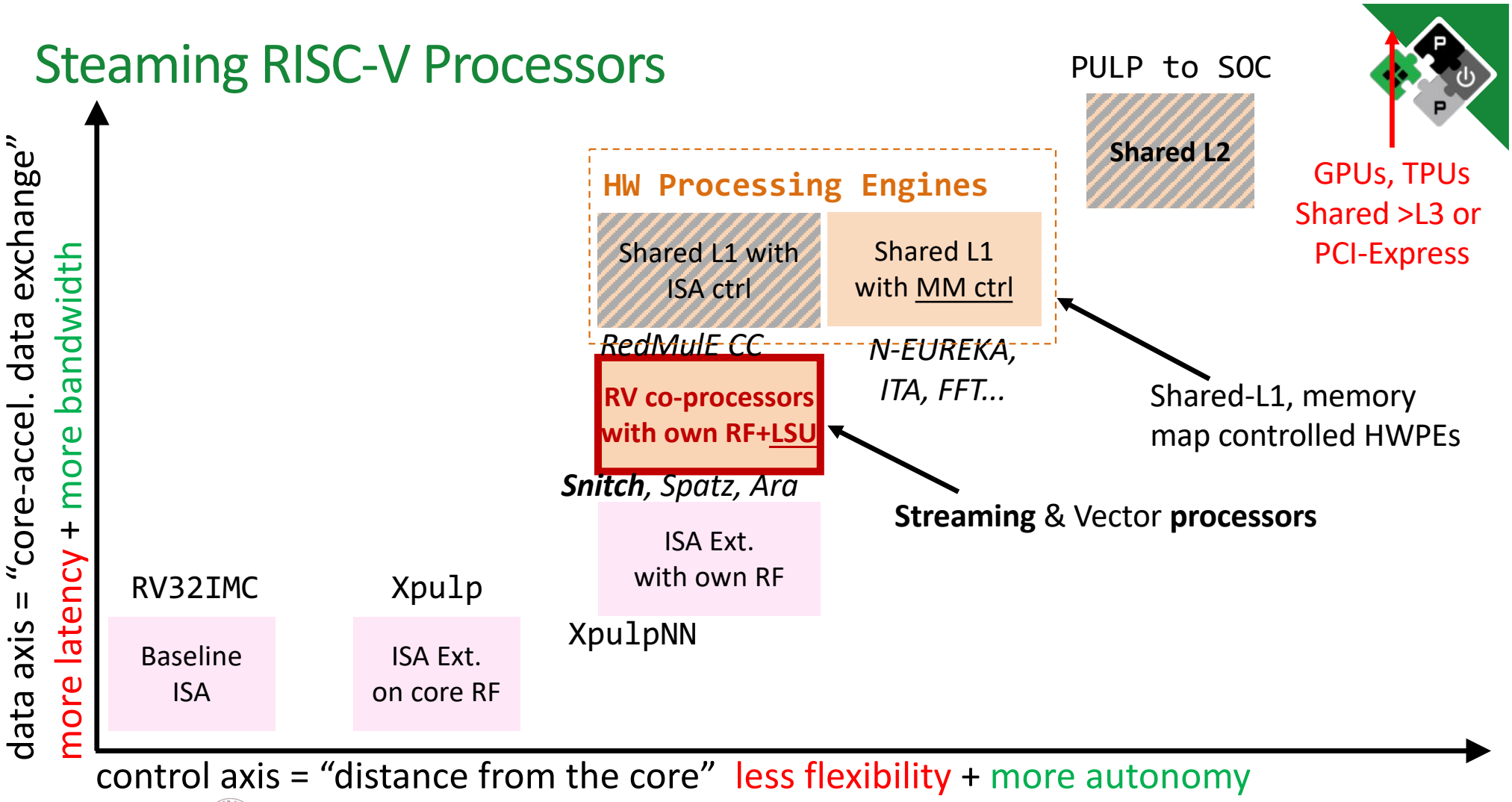
- IMA accelerator can only perform dense MVM
- IMA + function-specific accelerators are not general enough
  - → in real-world there will always be a kernel that your accelerators cannot handle
- IMA + MCU
  - → flexible enough, but at very poor performance on non-MVM kernels
- Heterogeneous IMA-centric programmable clusters
  - → high flexibility, good mitigation of Amdahl's effects



Summary

- **Trading-off performance with flexibility**
  - A good compromise with HWPEs
  - Tightly-coupled integration of accelerators and cores for “zero-copy” cooperation
  - Critical to architect interconnect and synchronization
    - ..to provide enough bandwidth to the accelerator and keep its utilization ~100%
- **Tightly-coupled schemes work for In-memory centric architecture as well**
  - ..but more integration challenges due to AIMC characteristics
- **In these architectures, RISC-V plays a key role**
  - SW programmability for orchestration, DMA, etc
  - Domain-specialized RISC-V processors to complement AIMC/accelerators compute features

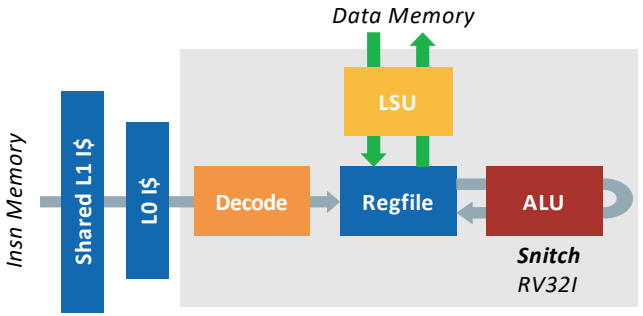
Steaming RISC-V Processors



Outline

- **Addressing the Von Neumann Bottleneck with Streaming Processors**
- **Snitch core-complex**
  - Latency-tolerant
  - Decoupled Functional units (with ded RF) through ISA-controlled acceleration port
  - Dedicated “LSU” with streaming extensions + acceleration of compute loops
- **Multi-precision FP units for AI**
  - AI FP formats + Micro-scaling
  - FP EXP acceleration and HW-SW acceleration of softmax/Gelu
- **Snitch cluster**
- **Scale-out for Physical AI**

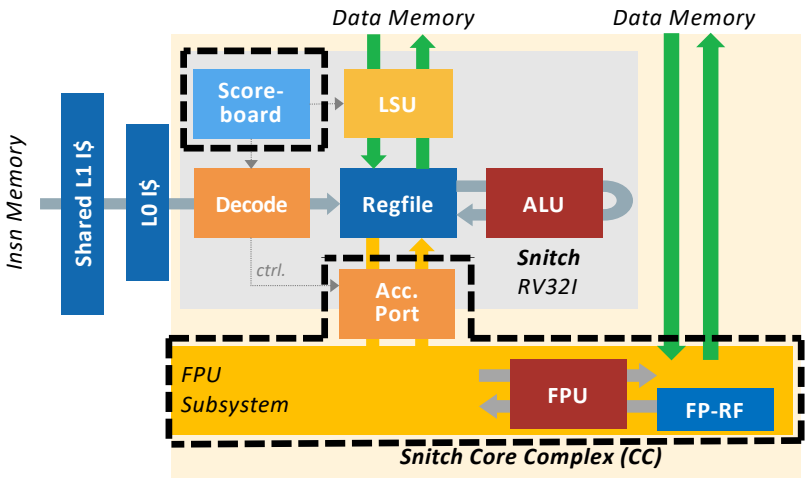
Addressing the Von Neumann Bottleneck



20kGE

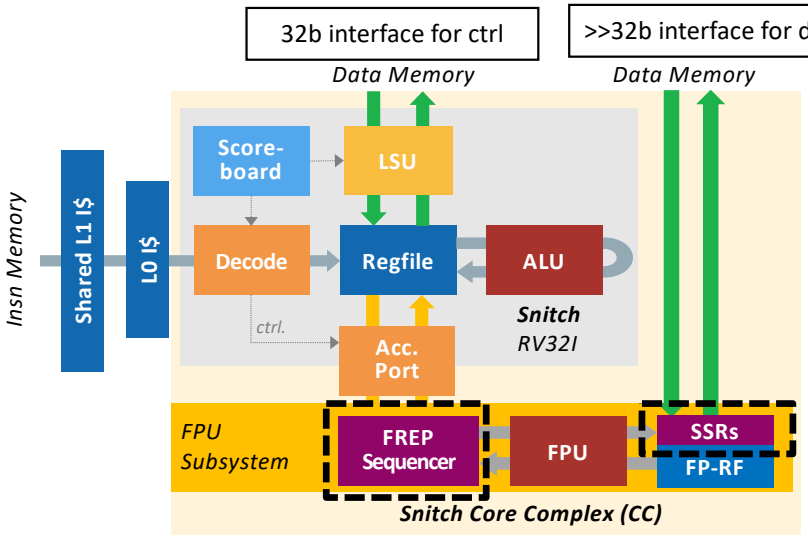
[Zaruba et al. IEEE TC 20]

Addressing the Von Neumann Bottleneck



- **Decouple PEs from control core through ISA-controlled Acceleration port**
  - **Multi-precision Floating-Point Units,** RVV, Tensor units, AME extensions

Addressing the Von Neumann Bottleneck



Streaming Semantic Registers (SSRs)

LD/ST elision

```
loop:
fld r0, %[a]
fld r1, %[b]
fmadd r2, r0, r1
```

→

```
scfg 0, %[a], ldA
scfg 1, %[b], ldB
loop:
fmadd r2, ssr0, ssr1
```

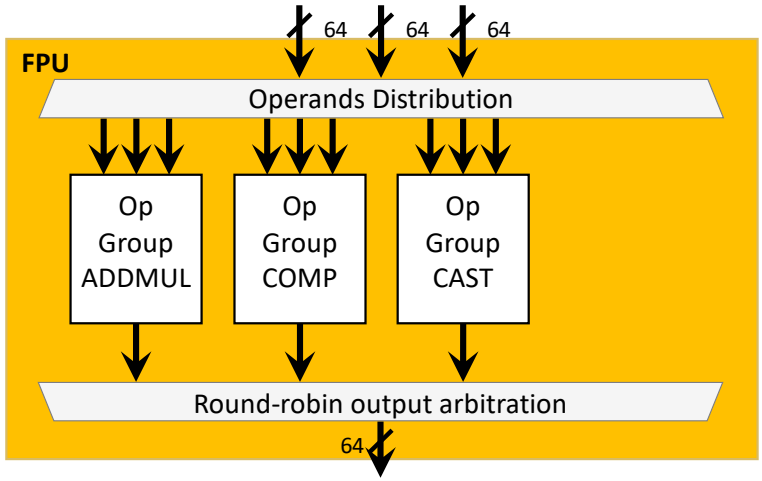
100% FPU utilization

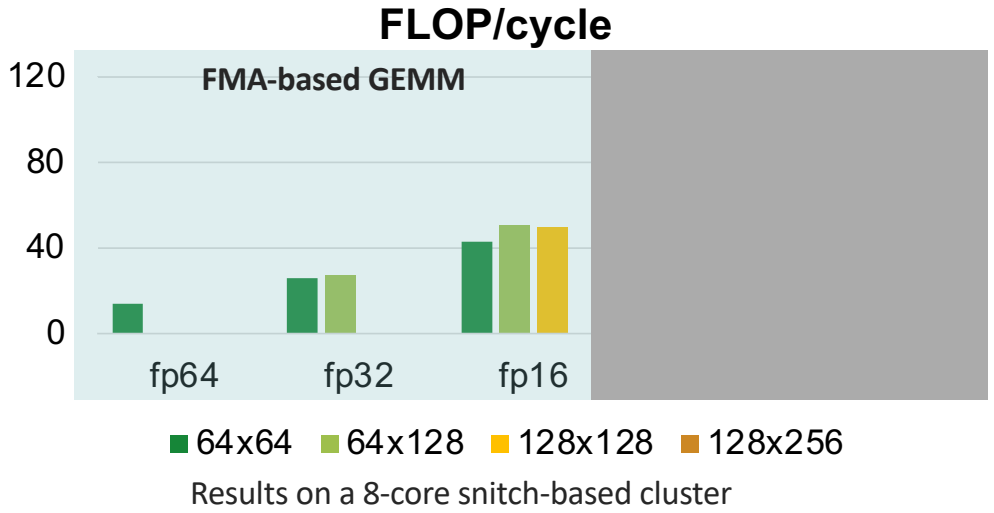
[Scheffler et al. IEEE JSSCC 25]

[Zaruba et al. IEEE TC 20]

- **Decouple PEs from control core through ISA-controlled Acceleration port**
  - **Multi-precision Floating-Point Units,** RVV, Tensor units, AME extensions
- **Accelerating `hot loops' with coarser-grained control/memory instructions**
  - **FREP** → Hardware loop controller (repeats FPU ops autonomously)
  - **SSR** → directly map *memory streams* to FP-RF with dense/sparse patterns (>>32-b bandwidth)

Multi-precision Floating-point for Transformers

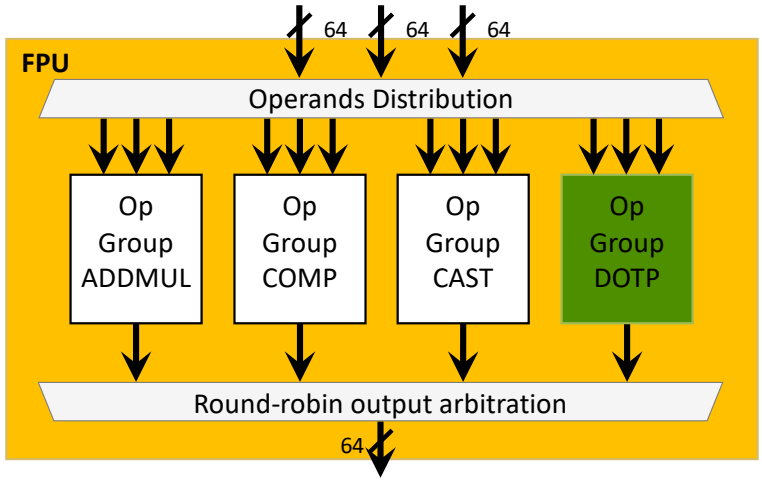


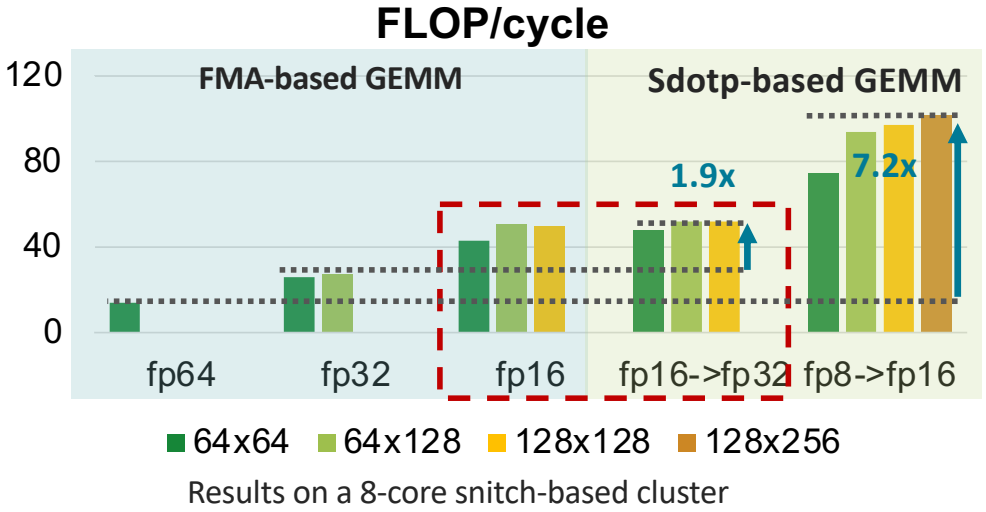


[Bertaccini et al. IEEE TETC 24]

- **Support for FP64, FP32 + AI formats FP16 (BF16), FP8 (BF8) (<u>with SIMD</u>)**

Multi-precision Floating-point for AI





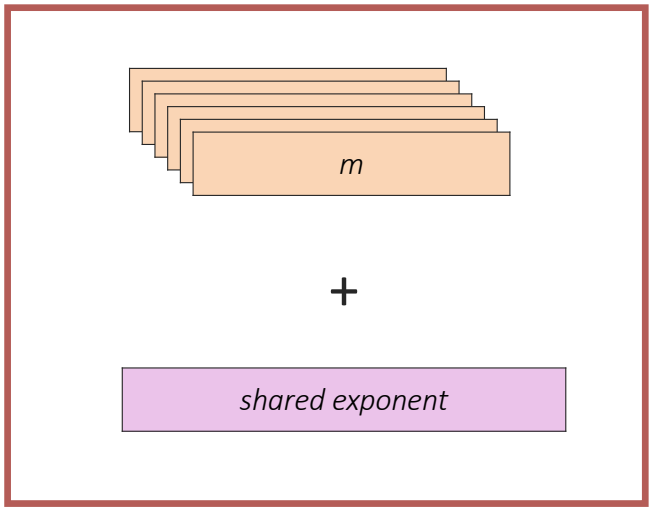
Results on a 8-core snitch-based cluster

[Bertaccini et al. IEEE TETC 24]

- **Support for FP64, FP32 + AI formats FP16 (BF16), FP8 (BF8) (<u>with SIMD</u>)**
- **Expanding DOTP Unit (27% area overhead)**
  - Minimize precision loss in accumulators (compared to FMA)
  - Dedicated OpGroup: two 16-to-32 (**2 dotp/cyc**) and four 8-to-16 units (**4 dotp/cyc**)

Low-Precision to the Rescue: Why Microscaling (MX)?

- Low-bit formats (e.g., FP8) reduce mem BW and compute cost
- But significant trade-off between range and precision.
  - Smaller exponent → **narrower dynamic range**
  - Smaller mantissa → **reduced precision**
  - Outliers force a shared per-tensor scale → all non-outliers may underflow/overflow and be represented as '0'/'max val', losing info
- **Microscaling (MX) data formats**, proposed as a standard by Open Compute Project[1]:
  - Offers finer-grained scaling by sharing an exponent over a block of 32 low-precision FP elements
  - Dot-product computation by expanding FP formats in fixed-point → accumulate "precisely" → scale the block with shared exponent
  - Enables low-bit formats while preserving accuracy[2,3].
- **Inefficient if not supported in HW/ISA**

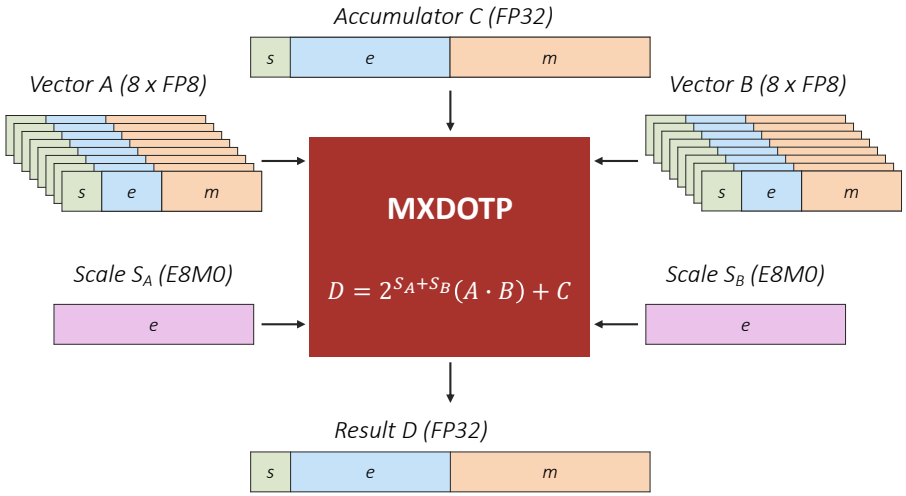


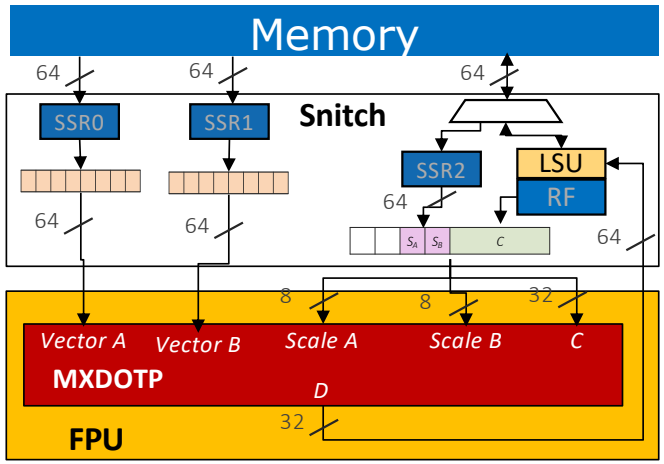
Block FP

Limited hardware support!

1. B. D. Rouhani et al., "OCP Microscaling Formats (MX) Specification," 2023.
2. B. D. Rouhani et al., "Microscaling Data Formats for Deep Learning," 2023.
3. H. Yang et al., "An Empirical Study of Microscaling Formats for Low-Precision LLM Training," 2025.

Micro-scaling (MX) Formats for Transformers





[İslamoğlu et al. IEEE ASAP 25]

- **Scaled 8-way dotp (MXDOTP) insn for MXFP8 (E5M2, E4M3), MXFP4, MXINT8**
  - Accumulation over a large ***Kulish*** accumulator (e.g., 96b)
    - **exactly** represent the sum of all partial products → avoids swamping and rounding errors
  - Performs final block scaling with shared exponents
  - Limited **17% extra FPU area** overhead **(<10%** at Core complex level)

MXDOTP on MatMuls

Baseline-FP32

MatMul kernel with FP32 inputs/outputs.

```
vfmac.s %[c], ft0, ft1
```

~ 0.5 instructions/MAC

Baseline-SW

MX MatMul kernel with SW emulation (FP32 output).

```
lbu t1, 0(%[Sa])
lbu t2, 0(%[Sb])
add t3, t1, t2
addi t3, t3, -127
flh ft0, 0(%[A])
vfcvt.s.b ft0, ft0
flh ft4, 0(%[B])
vfcvt.s.b ft4, ft4
vfmac.s ft8, ft0, ft1
slli t3, t3, 23
fmv.w.x ft0, t3
fmadd.s fa1, fa2, ft0, fa1
```

~ 3.9 instructions/MAC

MXDOTP ISA

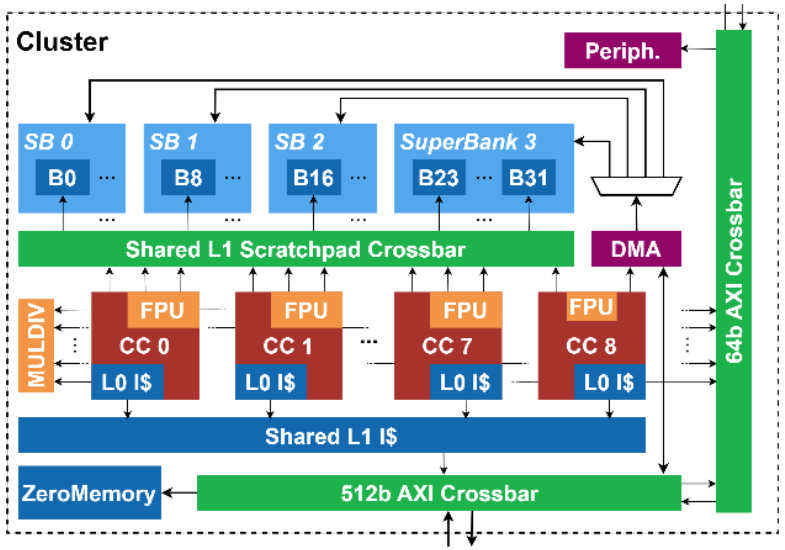
MX MatMul kernel using **MXDOTP** instruction (FP32 out).

```
mxdotp %[c], ft0, ft1, ft2, sel
```

~ 0.16 instructions/MAC

[İslamoğlu et al. IEEE ASAP 25]

Snitch-based Compute Cluster

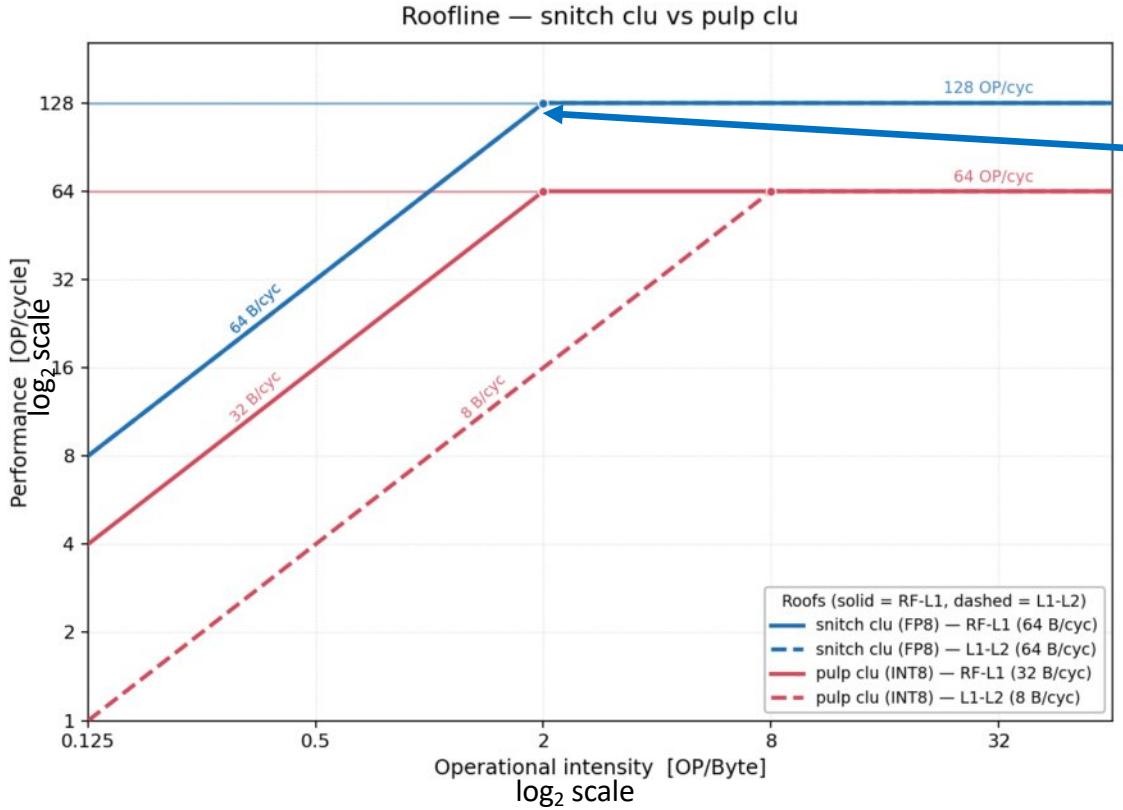


[Scheffler et al. IEEE JSSCC 25]

- **8 compute cores: 64b FPU + SSR + FREP**
- **9th DMA core: for large (512-b AXI) cluster-to-cluster/memory 1D/2D transfers in HW (3D or more in SW), latency-tolerant blk transfers**
- **128kB multi-banked scratchpad memory with wide DMA ports (SuperBanks)**

`github.com/pulp-platform/snitch_cluster`

PULP-Snitch Roofline Comparison



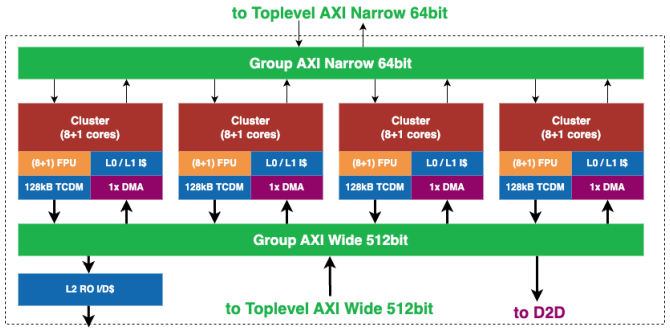
Higher perf @FP8

Higher BW (L2-L1 & RF-L1)

- **Peak perf (GF22):** PULP ~500 MHz; snitch ~ 1GHz
- **Power range (GF22):** PULP ~40-80mW ; snitch ~150-170mW
- **Area :** PULP ~ 1MGE; snitch ~ 3MGE

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

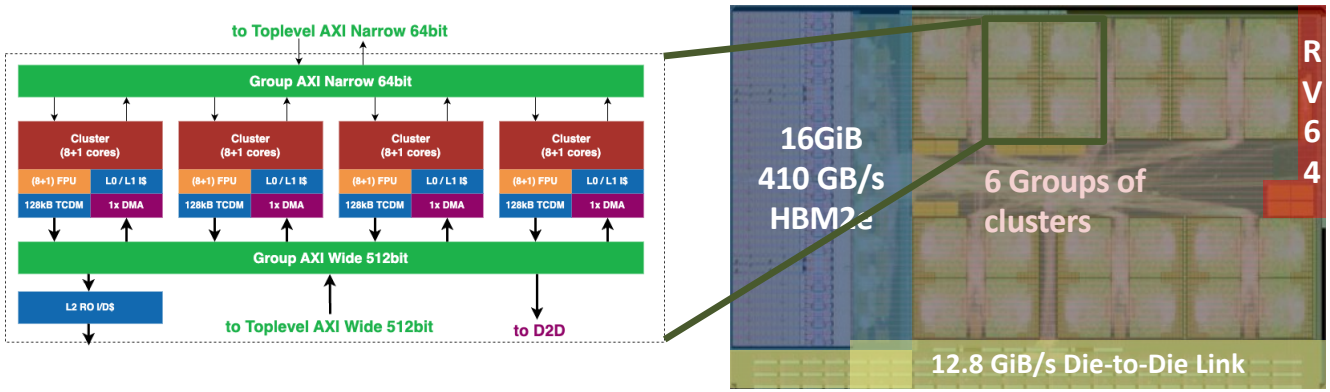
Hierarchical Scale-Out To Chiplets for GenAI



- **4 clusters organized in one "Group" w/ hierarchically shared bandwidth**
  - Fully-connected 512-b AXI for high BW local data exchange
  - Single 512-b AXI port for group-to-group and group-to-HBM communication

[Scheffler et al. IEEE JSSCC 25]

Hierarchical Scale-Out To Chiplets for GenAI



- **4 clusters organized in one "Group" w/ hierarchically shared bandwidth**
  - Fully-connected 512-b AXI for high BW local data exchange
  - Single 512-b AXI port for group-to-group and group-to-HBM communication
- **6 groups fully-connected at chiplet level (512b AXI for data, 64b AXI for ctrl)**
- **Heavy data reuse across clusters/groups to mitigate high HBM access latency**
  - Smartly distribute workload → tiling btw clusters/groups, layer pipelining btw chiplets
- **216 "snitch" cores/chiplet, 768 GFLOP/s and 3072FP8-GFLOP/s, 10-15W**

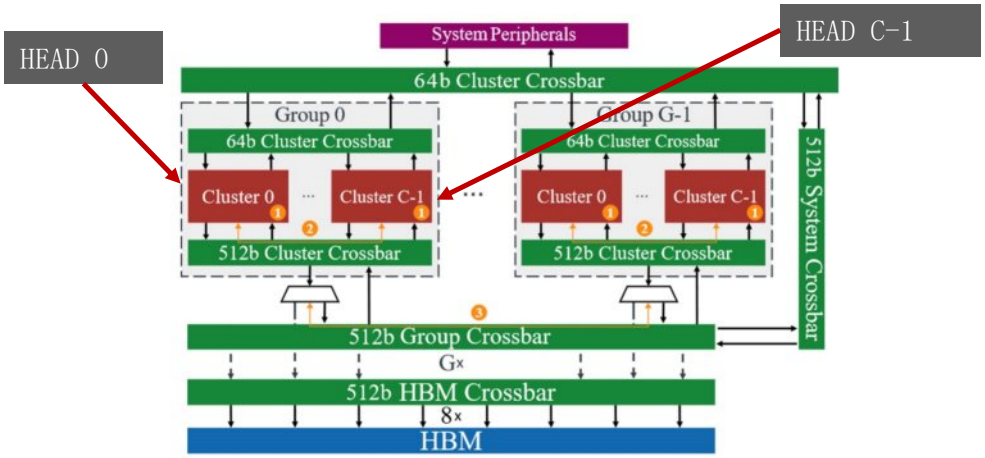
[Scheffler et al. IEEE JSSCC 25]

Hierarchical Scale-Out To Chiplets for GenAI

[Scheffler et al. IEEE JSSCC 25]

- **4 clusters organized in one "Group" w/ hierarchically shared bandwidth**
  - Fully-connected 512-b AXI for high BW local data exchange
  - Single 512-b AXI port for group-to-group and group-to-HBM communication
- **6 groups fully-connected at chiplet level (512b AXI for data, 64b AXI for ctrl)**
- **Heavy data reuse across clusters/groups to mitigate high HBM access latency**
  - Smartly distribute workload → tiling btw clusters/groups, layer pipelining btw chiplets
- **216 "snitch" cores/chiplet, 768 GFLOP/s and 3072FP8-GFLOP/s, 10-15W**

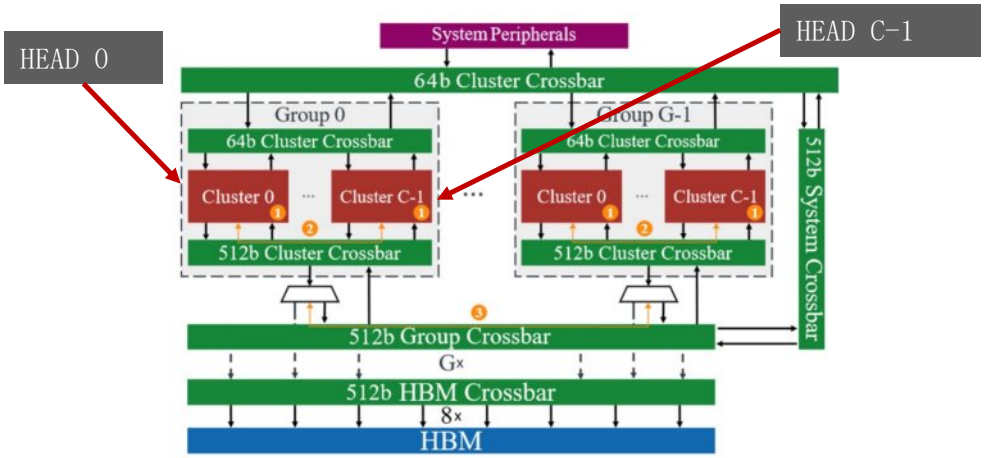
Amdahl's Strikes Again: FlashAttention

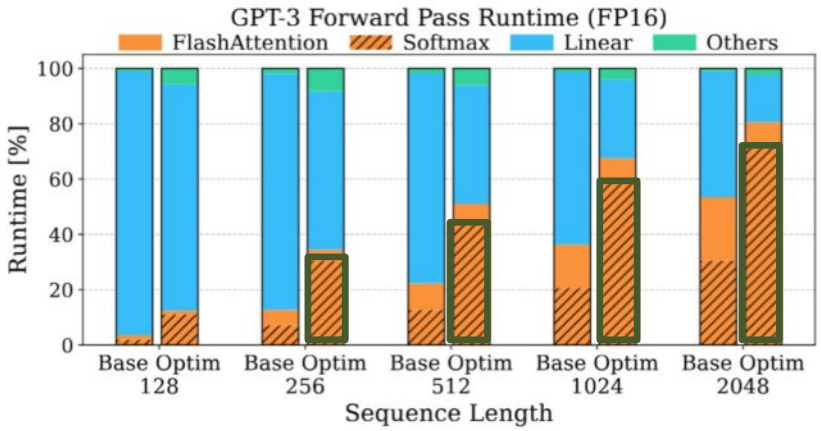


[Wang et al. ARITH 25]
[Potocnik et al. IEEE TCSAI24]

- **FlashAttention-2 workload: GEMM-dominated, Softmax ~10%**
- **Head-level tiling across the snitch clusters**
  - Each cluster computes Q/K/V tiles within assigned head

Amdahl's Strikes Again: FlashAttention





[Wang et al. ARITH 25]
[Potocnik et al. IEEE TCSAI24]

- **FlashAttention-2 workload: GEMM-dominated, Softmax ~10%**
- **Head-level tiling across the snitch clusters**
  - Each cluster computes Q/K/V tiles within assigned head
- **As GEMM is optimized (SSR, FREP, FPUs), bottleneck moves to non-linear fx**
  - Softmax: non-linear, non-paralleliable, dominated by floating-point EXP operations (glibc's exp approximation fx)

VFEXP: Lightweight FP EXP Acceleration

[Belano et al. IEEE JETCAS 25]

- **How to accelerate Softmax and other exp-based non-linear fx?**
  - Dedicated hardware → high cost/utilization (Softmax only ~10% of the workload) ☹
  - Accelerate EXP in the ISA + SW optimizations → flexible, low-cost ☺
- **EXP acceleration: Schraudolph's method + mantissa polynomial approximation**
  - **Faster** than CORDIC, **more flexible** than pure polynom. approx., **lower-cost** than LUTs
  - Implementation in **BF16** (extendable to other formats)

VFEXP: Lightweight FP EXP Acceleration

- **How to accelerate Softmax and other exp-based non-linear fx?**
  - Dedicated hardware → high cost/utilization (Softmax only ~10% workload) ☹
  - Accelerate EXP in the ISA + SW optimizations → flexible, low-cost ☺
- **EXP acceleration: Schraudolph's method + mantissa polynomial approximation**
  - **Faster** than CORDIC, **more flexible** than pure polynom. approx., **lower-cost** than LUTs
  - Implementation in **BF16** (extendable to other formats)

Schraudolph's method

$$\exps(x) \approx 2^{\text{int}(x)} \cdot \left(1 + \text{frac}(x)\right)$$

+ mantissa correction

$$\exp(x) \approx 2^{\text{int}(x)} \cdot (1 + \text{P}(\text{frac}(x)))$$

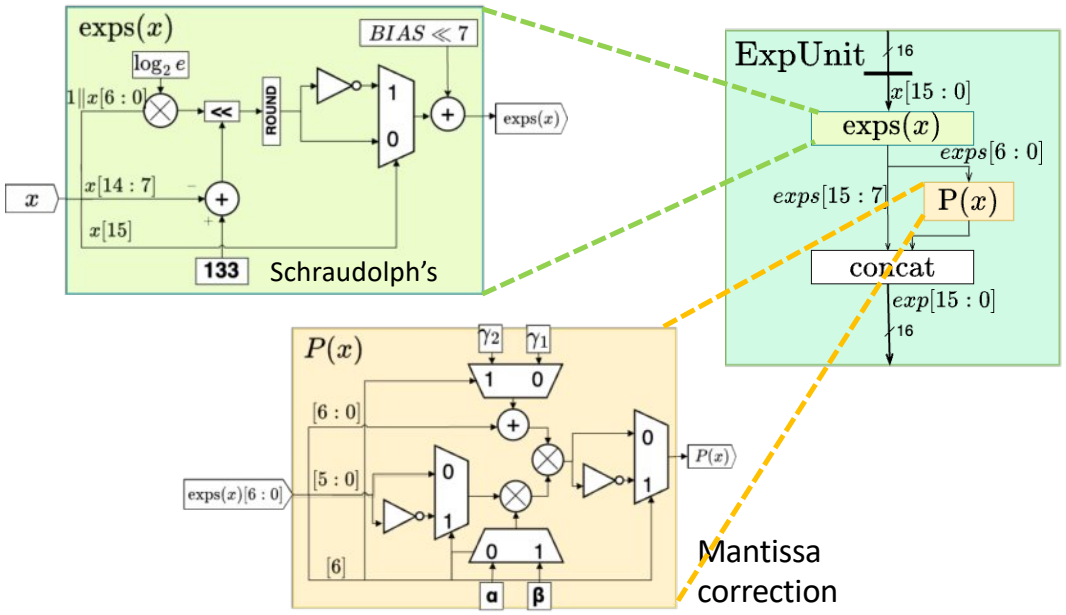
$$P(x) = \begin{cases} \alpha x\,(x + \gamma_1), & x \in [0, 0.5) \\ not\,(\beta\ not\,(x) \cdot (x + \gamma_2), & x \in [0.5, 1) \end{cases}$$

[Belano et al. IEEE JETCAS 25]

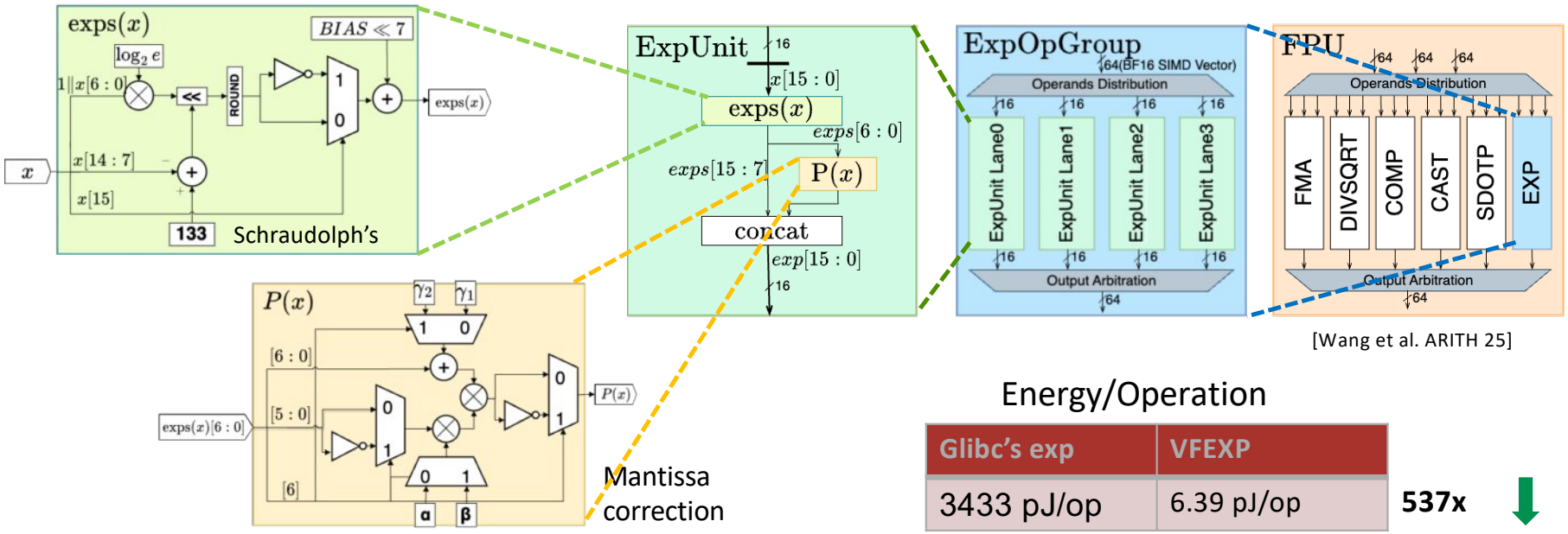
- **EXP Accuracy**
  - Relative accuracy **99.86%** (**37x** higher than Schraudolph's)

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

VFEXP: Lightweight FP EXP Acceleration



VFEXP: Lightweight FP EXP Acceleration



[Wang et al. ARITH 25]

Energy/Operation

Glibc's exp	VFEXP
3433 pJ/op	6.39 pJ/op

537x

- ExpOpGroup of 4 ExpUnit **BF16** lanes, 2 cycle of latency
- VFEXP (VEXP) instruction operating on **BF16** SIMD vectors (scalar BF16)
- **2.5%** area overhead w.r.t FPU (**8 kGE**) – **537x** less en/op wrt glibc's exp

Softmax Acceleration through ISA

FREP + **SSR** + glibc exp

```
// Find MAX
ssr config
frep.o %[n_frep], 1
vfmax.h ft3,ft3,ft1

// EXP w/ GLIBC
for (int i = 0; i < N; i++) {
y[i] = exp(x[i]- max_val)
sum += y[i];}

// NORMALIZE
ssr config
frep.o %[n_frep], 1
vfdiv.h ft2,%[sum],ft1
```

350 Cycle/N

FREP + SSR + **VFEXP**

```
// Find MAX
ssr config
frep.o %[n_frep], 1, 0, 0
vfmax.h ft3,ft3,ft0

// EXP w/ VFEXP
ssr config
frep.o %[n_frep], 4
vfsub.ah ft3, ft1, %[max]
vfexp.ah ft2, ft3
vfadd.ah ft4, ft2

// NORMALIZE
ssr config
frep.o %[n_frep], 1, 0, 0
vfmul.h ft4,%[1/sum],ft0
```

4 Cycle/N

..+ **Loop Unrolling**

```
// Find MAX
ssr config
frep.o %[n_frep],16
vfmax.h ft3,ft3,ft0
vfmax.h ft4,ft4,ft0
vfmax.h ft5,ft5,ft0
...
// EXP w/ VFEXP
ssr config
frep.o %[n_frep], 16
vfsub.ah ft3, ft1, %[max]
vfsub.ah ft4, ft1, %[max]
vfexp.ah ft3, ft3
vfexp.ah ft4, ft4
vfadd.ah ft3, ft3
vfadd.ah ft4, ft4
// NORMALIZE
ssr config
frep.o %[n_frep], 16
vfmul.h ft4,%[1/sum],ft0
vfmul.h ft4,%[1/sum],ft0
vfmul.h ft4,%[1/sum],ft0
.....
```

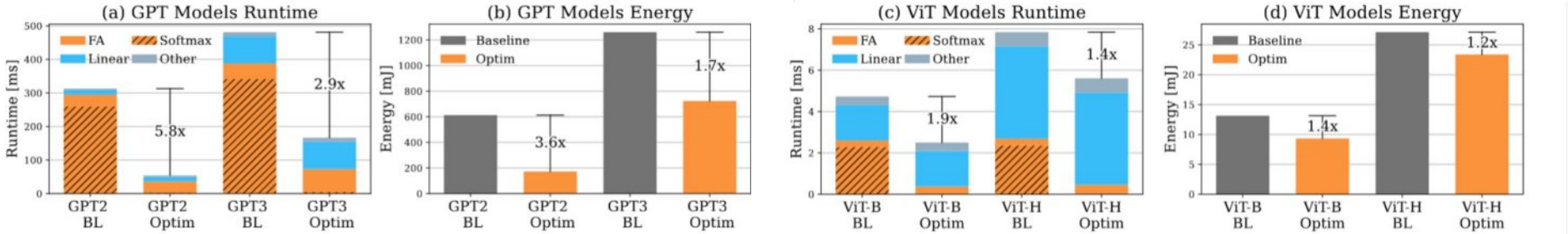
2.125 Cycle/N

- **165x** higher performance than baseline on Softmax
- **74.3x** better energy efficiency than baseline on Softmax
- Similar optimizations apply for GeLu and other exp-based non-linear fx

[Wang et al. ARITH 25]

VFEXP: End-to-End Results

End-to-End Non-autoregressive Inference



End-to-End Accuracy

Model	Dataset	Metric	FP32	BF16	BF16 EXP
GPT-2	WikiText	Perplexity (↓)	37.4	37.8	37.8
	ArcEasy	Accuracy (↑)	43.8	42.9	43.7
ViT-B	ImageNet	Accuracy (↑)	80.3	80.3	80.3
	CIFAR-10	Accuracy (↑)	98.5	98.5	98.5

[Wang et al. ARITH 25]

- **Very limited area/power cost (<1% overhead at core level)**
- **>10-100x improved performance and energy efficiency on Softmax**
- **Negligible accuracy loss on end-to-end execution of GPT-x/ViT models**
- **Up to 5.8x latency and 3.6x energy reduction on end-to-end workloads**

ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

HW/SW Co-design of Energy Efficient RISC-V Edge AI Platforms

ACACES 2026, Fiuggi (Italy), 12-17 July 2026

Part 5: RISC-V Vector Processors: Advantages and Drawbacks for Edge AI

Angelo Garofalo angelo.garofalo@unibo.it
agarofalo@iis.ee.ethz.ch

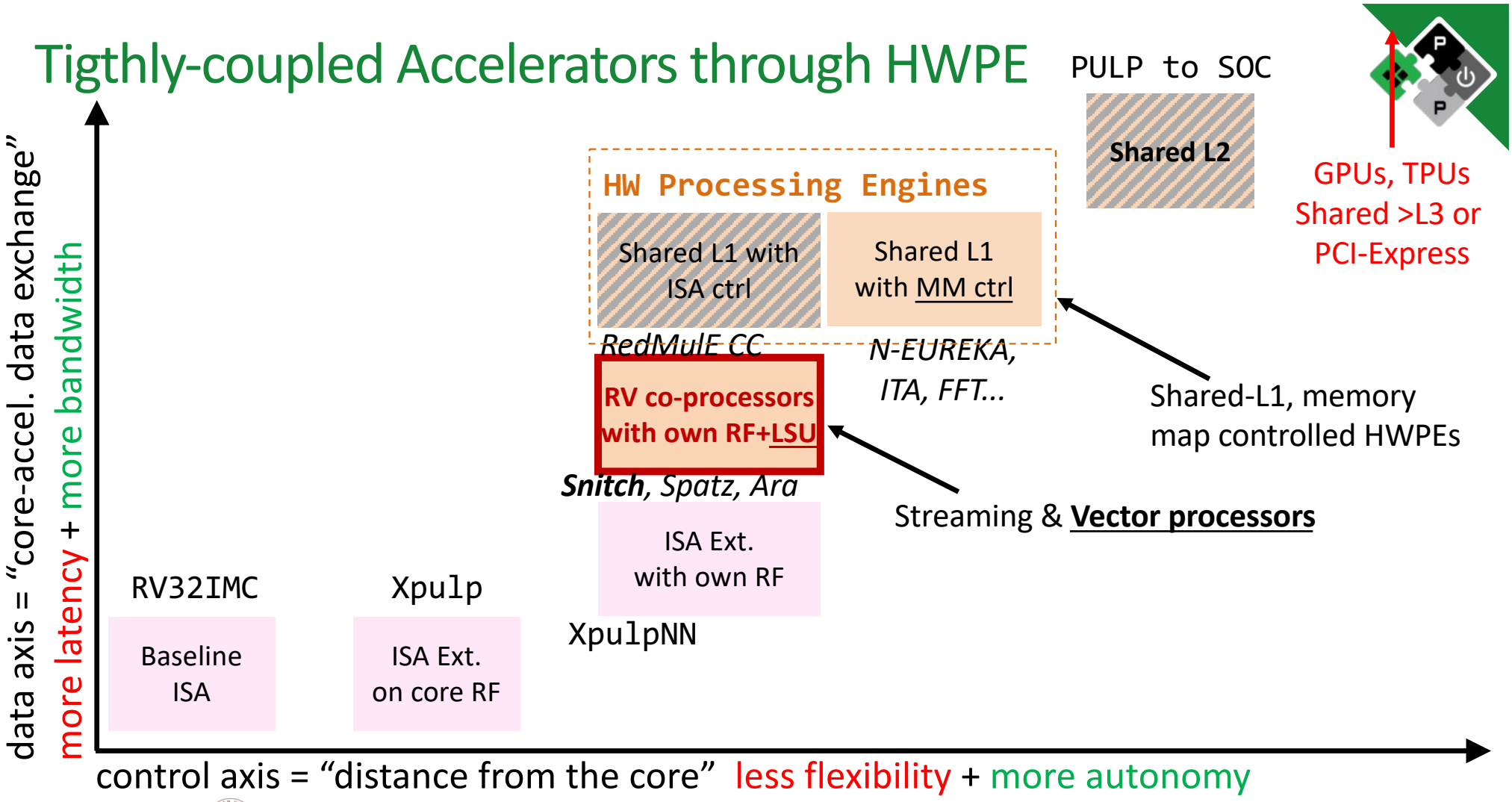
pulp-platform.org
@pulp_platform
company/pulp-platform
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

Outline

- Why Vector Processors are efficient and how they overcome the Von Neumann Bottleneck?
- Recap/Introduction to the Vector Compute Paradigm
- Introduction to Spatz: A compact, RISC-V Vector Cluster for the edge
- Comparison of Stream/Vector/Thread approach
- Conclusion

Tigthly-coupled Accelerators through HWPE

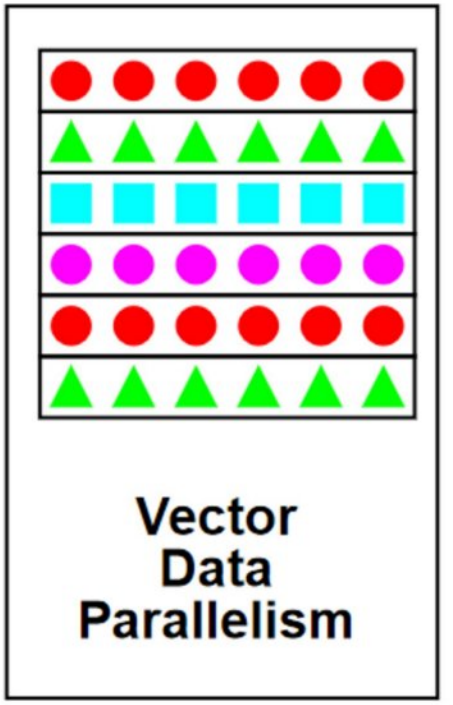


Data (or Vector) Level Parallelism

- **The least flexible form of parallelism**
  - Among Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP)
  - Anything that can be expressed to exploit DLP, can also be written to exploit ILP or TLP

…but…

- **The cheapest form of parallelism**
  - A machine that exploits DLP only needs to fetch and decode a single instruction to describe a whole array of vector operations
- **Very Energy efficient!**
  - If you can program it…
  - *Who is responsible for finding parallelism in the instruction flow?*

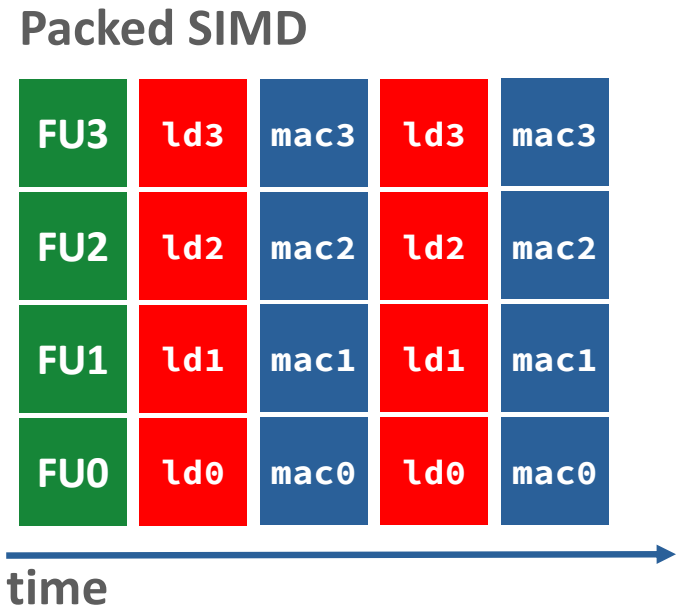


Packed-SIMD (Like Xpulp SIMD insns) Limitations

- **Let's analyze the limitations of our previous scalar cores exploiting SIMD at instructions' level**

- **Vector length is set in stone**
  - Very short vectors, e.g., Arm Neon has a VLEN of 128 bits
  - RI5CY (e.g. 4x8-bit elements with sdtop8)

- **VLEN is encoded in the instruction itself**
  - **VADD.I16 Q0, Q1, Q2/ pv.sdotp.b …**



- **Scalability problem: a wider VLEN requires a new ISA.**
  - Widest extension to date: Intel AVX-512

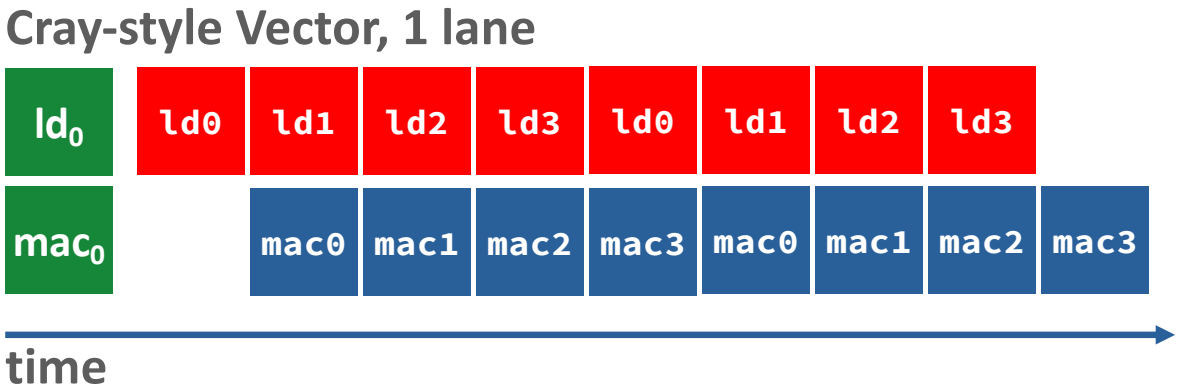
From Packed-SIMD to Vector Processors

- **Vector seen as: one-dimensional array of numbers**
- **Many programs use vectors**

```
for (i = 0; i <= 49; i++)
    C[i] = (A[i] + B[i]) / 2;
```

- **Vector Processor's instructions operate on vectors rather than scalar values**
- **Basic Requirements**
  - **Vector** register file → Needed to store/load the vectors
  - **Vector Length Register (VLEN → 'vl' in RVV)** → to set (with an insn) the lenghts of vectors on which we operate on
  - **Vector Stride Register (VSTR)** → to locate where vector elements are stored in memory → in RISC-V encoded into the insn
    - Vector elements might be stored apart from each other in memory
    - Stride → distance in memory between two elements of the same vector

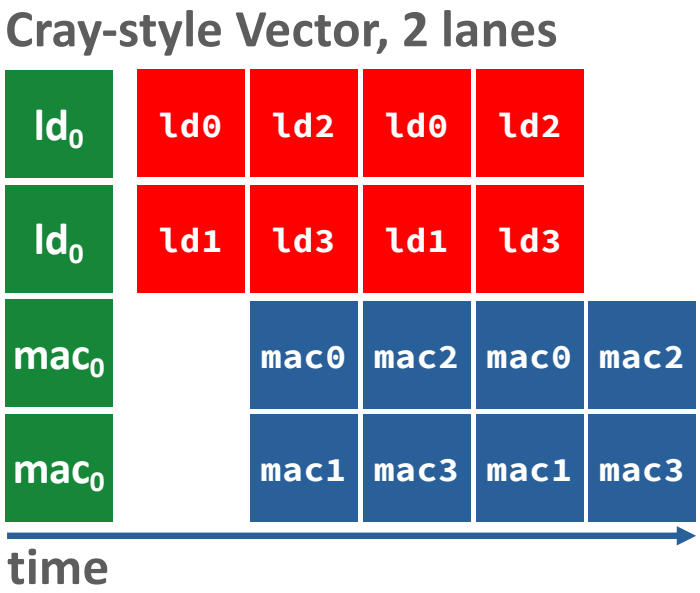
Vector Processors



- **Vector instruction performs an operation on each element in consecutive cycles**
  - Vector Functional Units can be pipelined
  - Each pipeline stage operates on a different data element
- **Vector instructions allow deeper pipelines in functional units because..**
  - **No Intra-Vector dependencies** → No hardware interlocking needed within a vector
  - **No control flow** within a vector
  - **Known stride allows easy address calculation** for all vector elements

Vector Processors

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



Vector Processor Advantages

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

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

If parallelism is irregular → Very inefficient!

Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  FLD F0, 0(R1)
  FLD F2, 0(R2)
  FADD F4, F2, F0
  FSW F4, 0(R3)
  ADDI R1, 8
  ADDI R2, 8
  ADDI R3, 8
  ADDI R4, -1
  BNEZ R4, loop
```

```
# Vector Code
  VSETVL 64
  VLE V1, R1
  VLE V2, R2
  VADD  V3, V1, V2
  VSE V3, R3
```

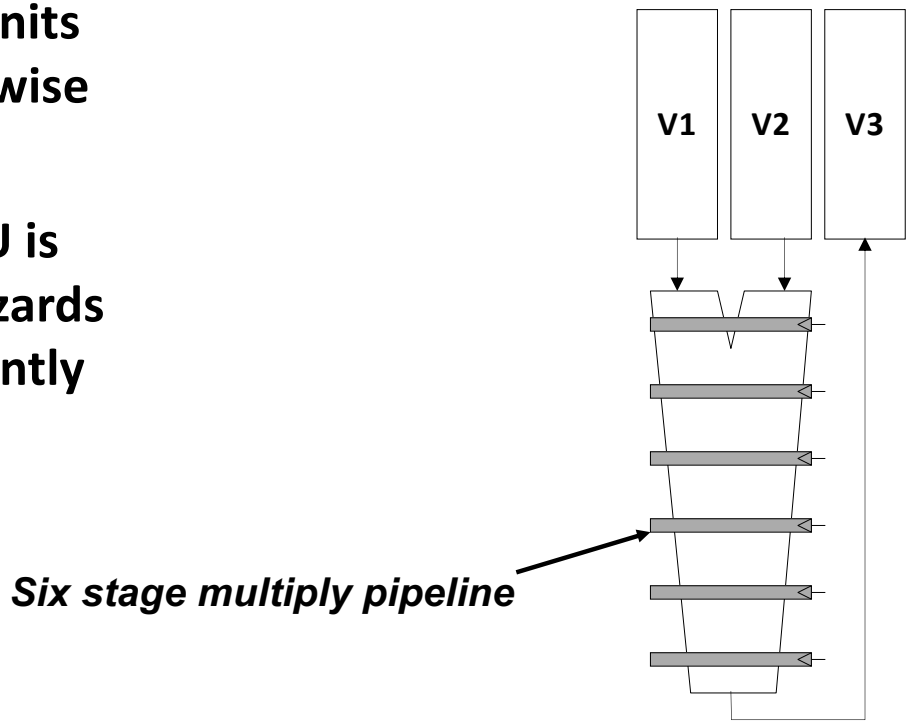
```
# RVV Code
li        t0, 64
vsetvli   t1, t0, e64
vle64.v   v1, (R1)
vle64.v   v2, (R2)
vfadd.vv  v3, v1, v2
vse64.v   v3, (R3)
```

Vector Instruction Set Advantages

- **Compact**
  - One instruction (short) encodes N operations
- **Expressive, highly powerful semantic, tells the hardware that these N operations:**
  - are **INDEPENDENT**
  - Use the **same Functional Units**
  - Access **Disjoint registers**
  - Access registers **with the same pattern as previous instructions**
  - Access either
    - A contiguous block of memory (unit-strided load/store)
    - A known memory pattern (strided load/store)
- **Highly scalable**
  - Can run the same object code on more parallel pipelines (or lanes)

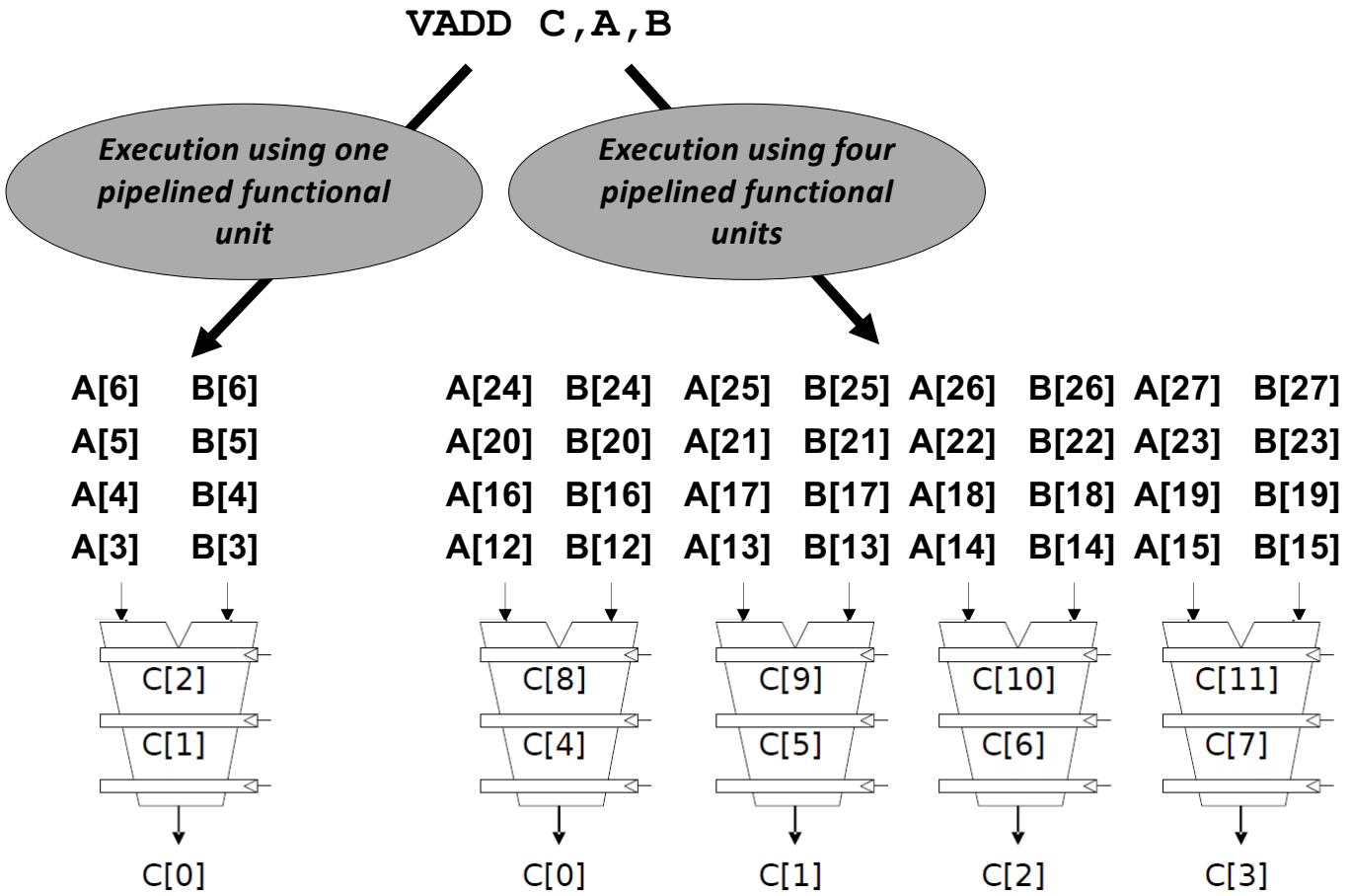
Vector Arithmetic Execution

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

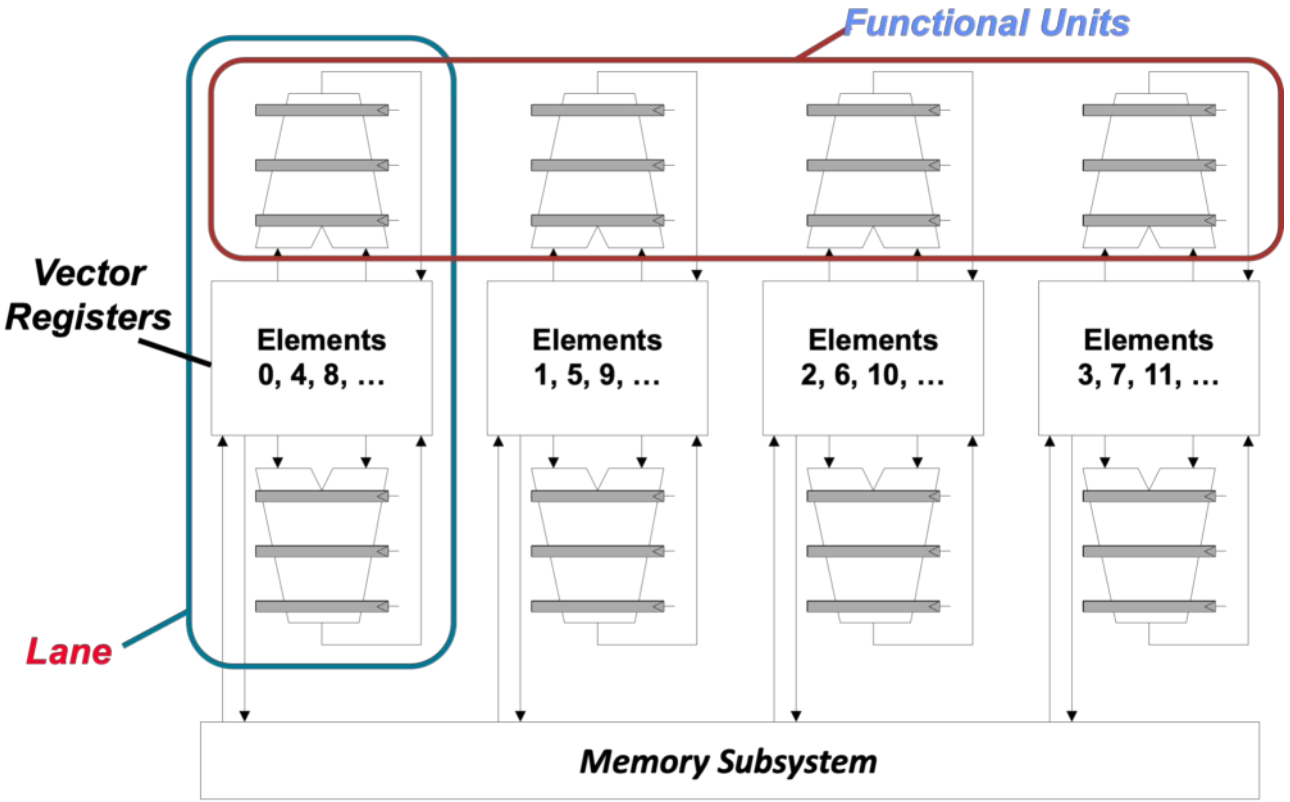



```
V3 <- v1 * v2
```

Vector Instruction Execution



Vector Unit Structure



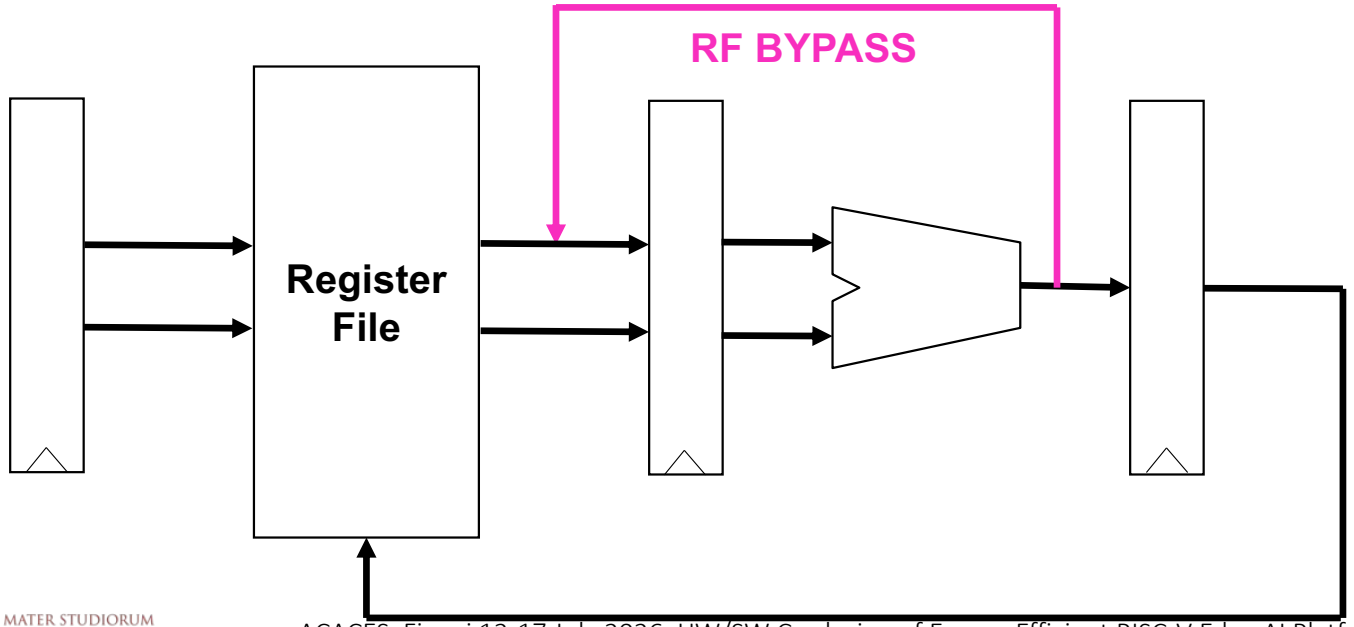
- Each lane sees one portion of the VRF and one set of FUs
- Each lane computes in parallel independently from others (no communication)

A Refresh: Operand Forwarding

- **Operand forwarding (register bypassing)**

```
ADD r1,r2,r2
MUL r4,r1,r3
```

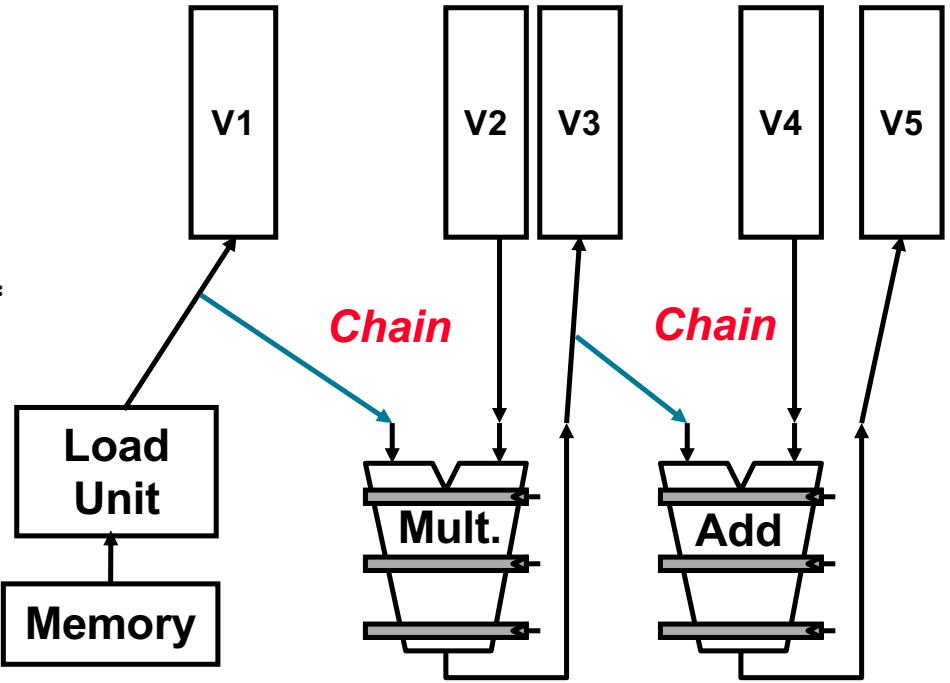
RAW dependency on **r1**!
We should wait until LD has written back before reading operands for MUL



Vector Chaining

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

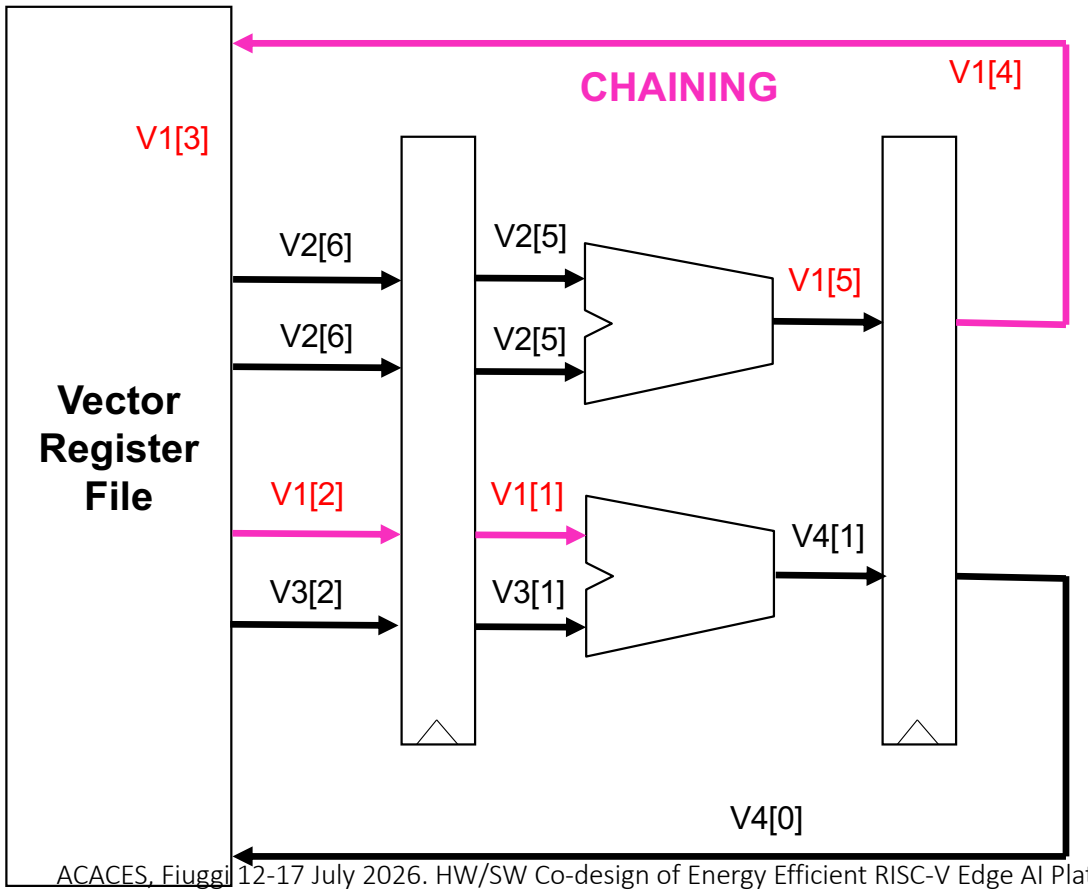
```
VLE  v1
VMUL v3,v1,v2
VADD v5, v3, v4
```



Vector Flexible Chaining

- **We can use VRF as chaining node**

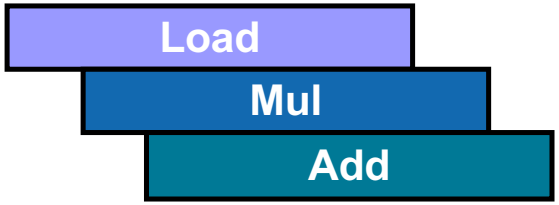
```
VADD v1,v2,v2
VMUL v4,v1,v3
```



Vector Chaining

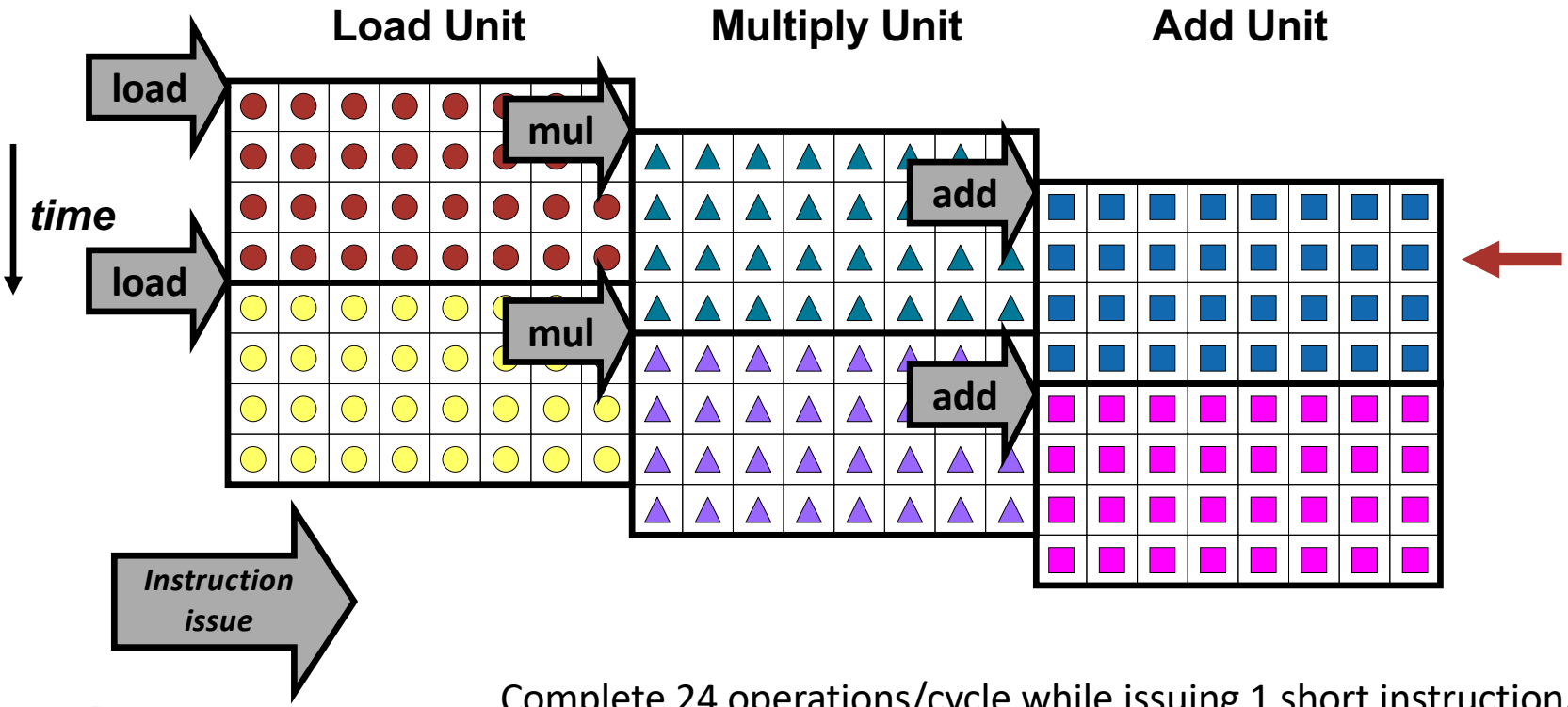
- **Without chaining, must wait for last element of result to be written before starting dependent instruction**

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



Vector instruction parallelism

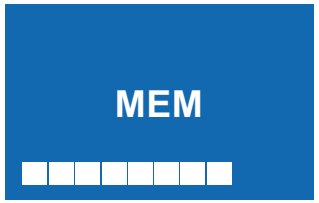
- **Can overlap execution of multiple vector instructions**
  - Example machine has 32 elements per vector register and 8 lanes



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

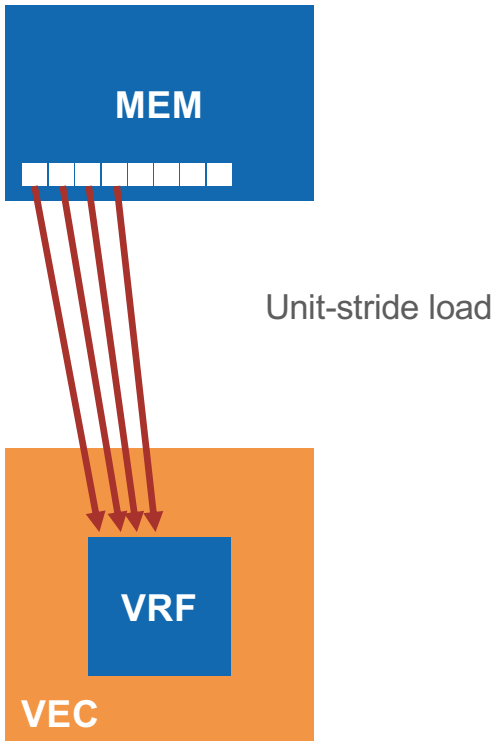
A brief overview of vector execution

- **The vector elements to process are in memory**
- **Our vector architecture contains a Vector Register File (VRF)**
- **We load vector elements from memory to the VRF**
  - The **VRF buffers** our **vector** close to the processing units
  - The **VRF** is the **bottom** of the **memory hierarchy**!
  - The **VRF increases data reuse** by exploiting data locality
- **And if our vector registers are too small to host a full vector?**
- **We only load and process the first chunk of the vector!**
- **Then, when we are over with it, we continue with the second chunk**
- **Until we process the whole vector in memory!**



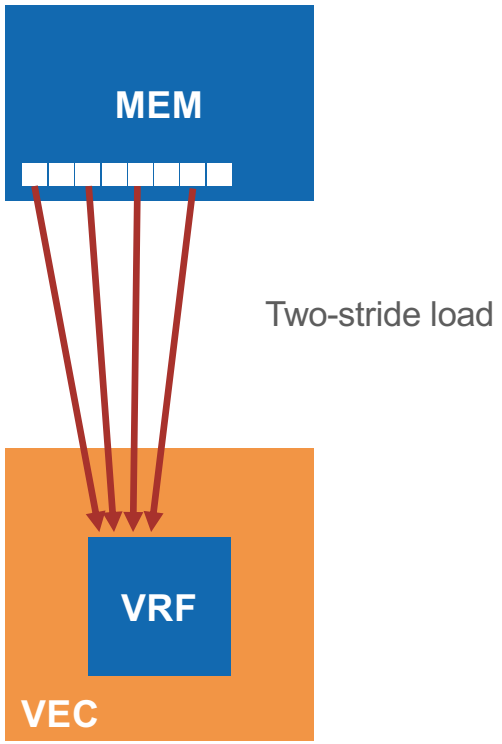
Vector memory operations

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



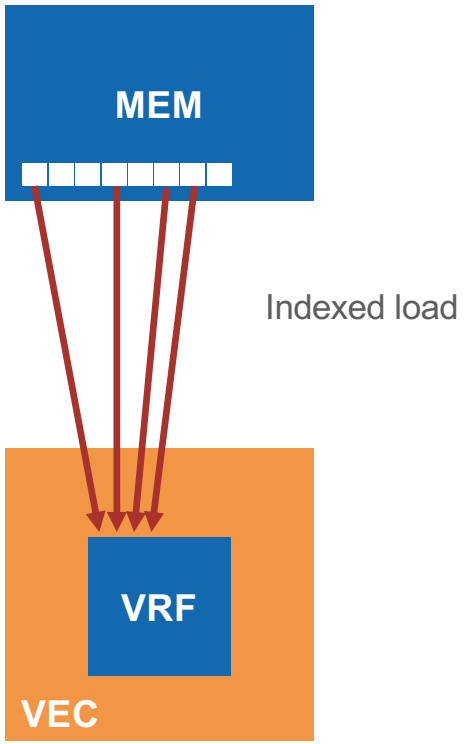
Vector memory operations

- **Load/store operations move groups of data between registers and memory**
- **Three types of addressing**
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth



Vector memory operations

- **Load/store operations move groups of data between registers and memory**
- **Three types of addressing**
  - **Unit stride**
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - **Non-unit (constant) stride**
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - **Indexed (gather-scatter)**
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize



Vector Scatter-Gather

- **Want to vectorize loops with indirect accesses:**

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

Vector Scatter-Gather

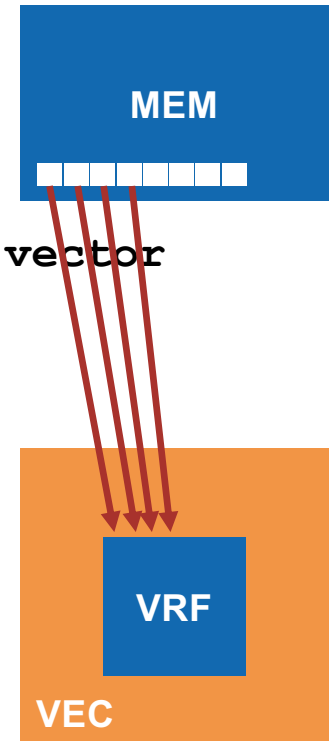
- **Want to vectorize loops with indirect accesses:**

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

```
vle32.v  v4, (a1)          # Load indices in D vector
```

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

Unit-stride load



Vector Scatter-Gather

- **Want to vectorize loops with indirect accesses:**

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

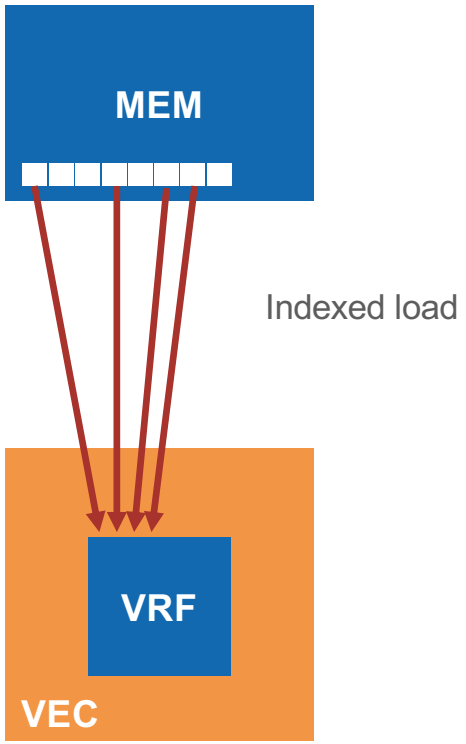
- **Indexed load instruction (Gather) [RVV]**

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

Indexed load

C memory address

D indices



Vector Scatter-Gather

- **Want to vectorize loops with indirect accesses:**

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

- **Indexed load instruction (Gather) [RVV]**

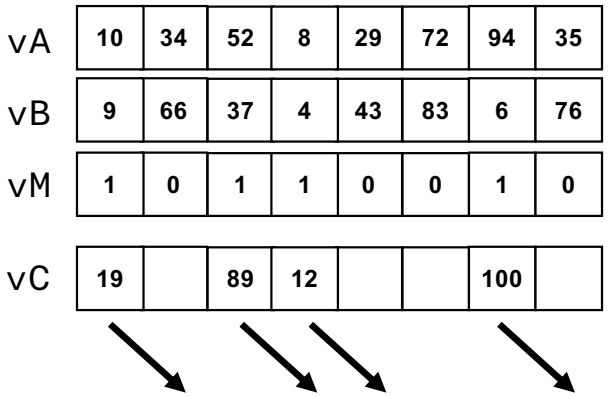
```
vle32.v  v4, (a1)          # Load indices in D vector
vsll.vi v4, v4, 2          # index to byte addressing (conversion)
vluxei32.v v3, (a2), v4 # v3[i] = C[index[i]]
vle32.v v2, (a3)           # v2 = B[i]
vadd.vv v1, v2, v3         # v1 = B[i] + C[index[i]]
vse32.v v1, (a4)           # store A[i]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

Vector Conditional Execution

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

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

- **Solution: Add vector *mask* (or *flag*) registers**
  - vector version of predicate registers, 1 bit per element
- **…and *maskable* vector instructions**
  - vector operation becomes NOP at elements where mask bit is clear



- **Code example (RVV):**

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

Vector Reductions

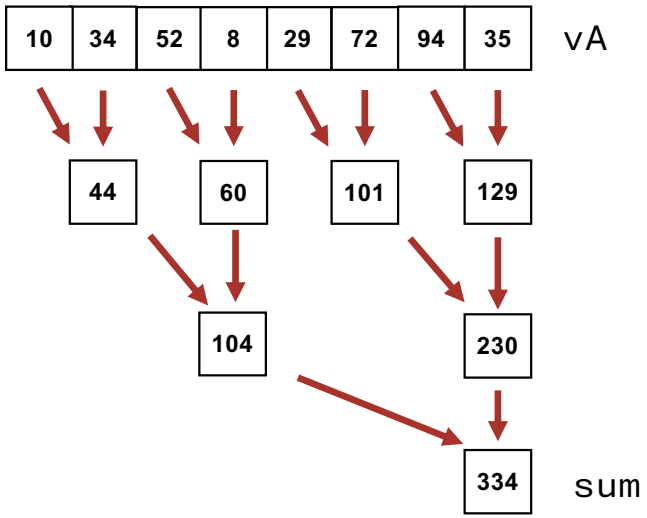
- **Problem: Loop-carried dependence on reduction variables**

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

- **Solution: vector reduction**

```
VLE      vA, rA
VREDSUM sum, vA
```

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

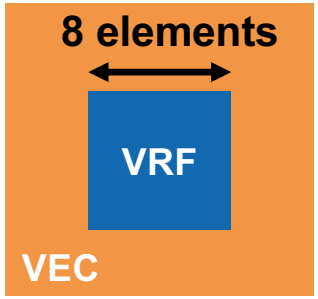


How to know the vector register length

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

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

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

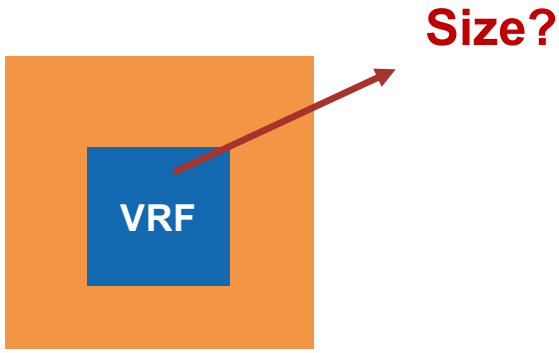
(vl will be set to 8)



A vector length problem

- **Problem:**
  - You want to add two 143-element vectors
  - You don't know your vector architecture and its VRF size
  - You will probably not be able to process the whole vector in one iteration

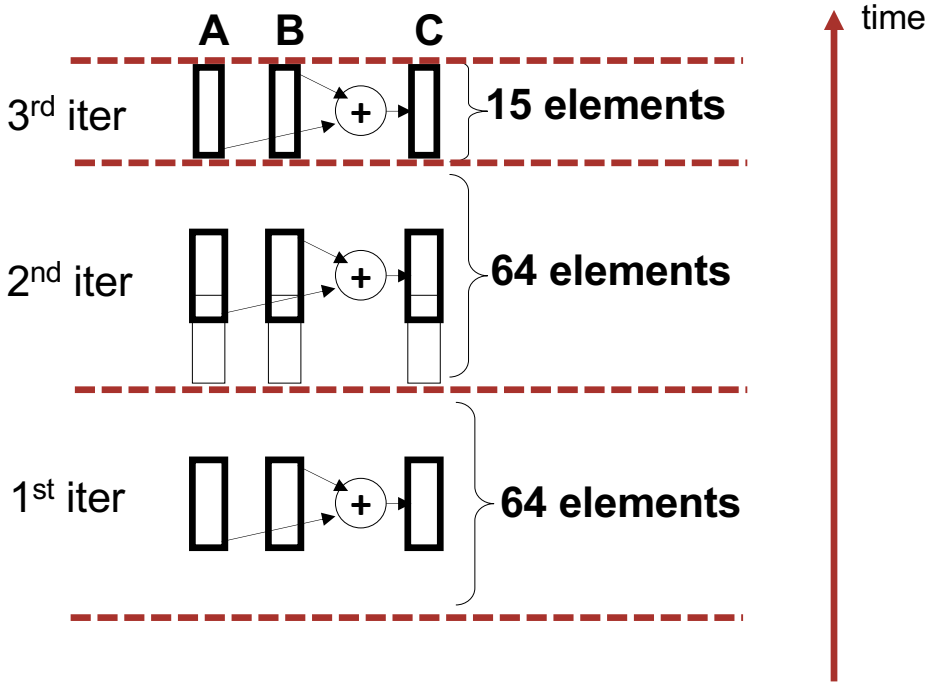
```
// We have a vector with 143 elements
avl = 143;

// We can query the architecture
vl = vsetvl(0)   // vl == 0
vl = vsetvl(8)   // vl == 8
vl = vsetvl(35)  // vl == 35
vl = vsetvl(63)  // vl == 63
vl = vsetvl(64)  // vl == 64
vl = vsetvl(65)  // vl == 64
vl = vsetvl(143) // vl == 64
// We can process up to 64 elements with a vector instruction!
```



Vector stripmining

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

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



Vector code is portable

- **Vector code is agnostic to the vector length…**
- **ONLY IF YOUR CODE SUPPORTS STRIPMINING!**
- **Proper vector code can work on different architectures with different VRF sizes**

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

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

Matrix Multiplication

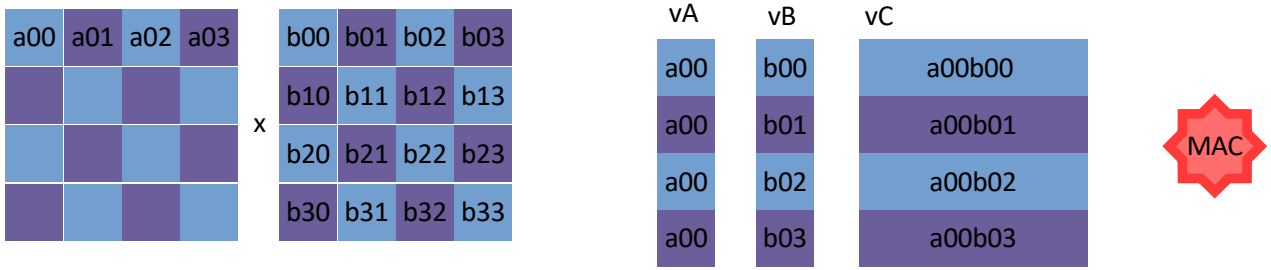
- **Consider the following:**

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
    for (j=1; j<n; j++) {
        sum = 0;
        for (t=1; t<k; t++) {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```

- **Do you need to do a bunch of reductions? NO!**

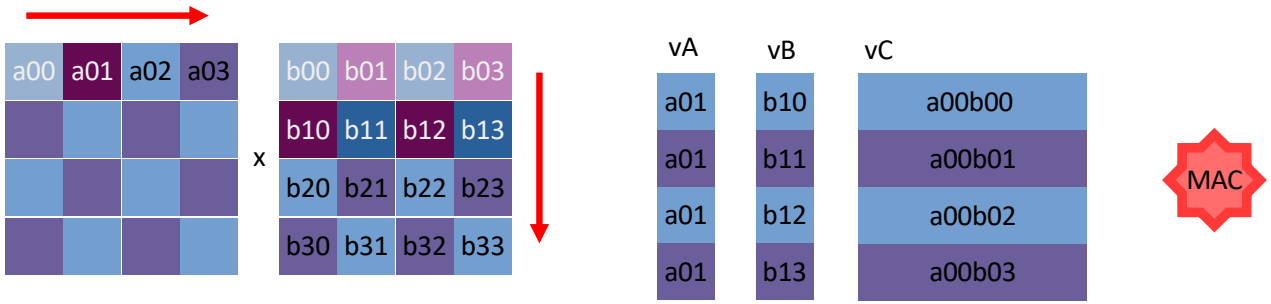
Matrix multiplication by “scalar splatting”

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



Matrix multiplication by “scalar splatting”

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

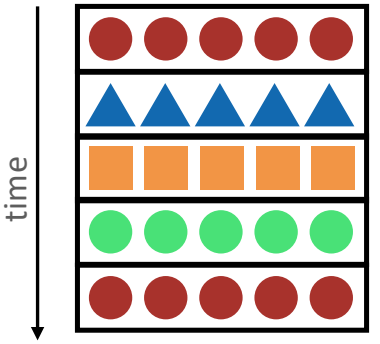


Matrix Multiplication by “Scalar Splatting”

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

Vector Processors: fighting back the von Neumann Bottleneck

- **One of the most efficient approaches to tackle the VNB**
- **Exploiting the DLP of modern workloads by amortizing the instruction fetch and decode energy cost over many cycles of computation**

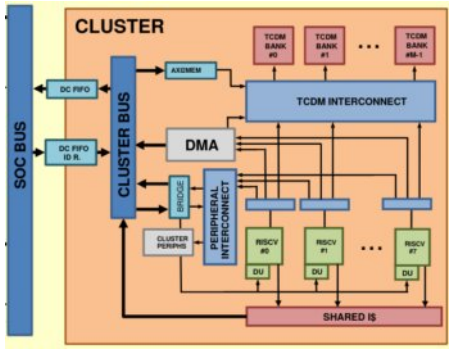


- **Since their inception, vector processors are tied with supercomputers**
  - Cray-1, A64FX, Hwacha, **Ara,** Vitruvius…
    - Large vector units tied with application-class processors
    - Support for tricks that exploit **ILP** on top of **DLP** (OoO execution, renaming, speculation) which lead to *efficiency loss*

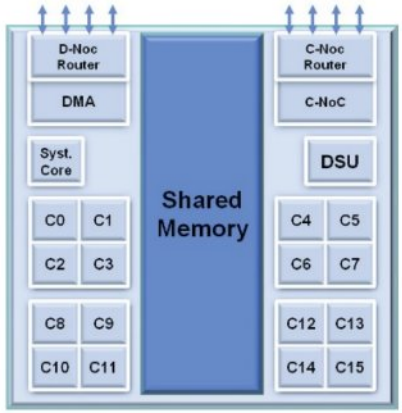
- **What about a tiny embedded vector machine that focus on the DLP?**
  - Arm Helium/MVE (M-Profile Vector Extension) on the Cortex-M cores
  - RISC-V's Zve32x and Zve64x subsets of the Vector Extension (RVV)

The shared-L1 cluster: a ubiquitous building block

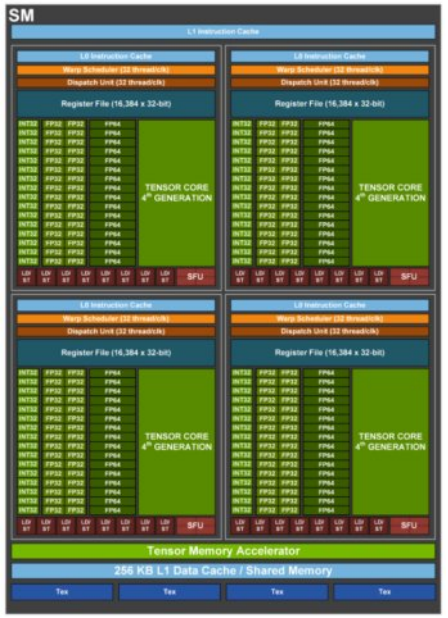
- Cluster of PEs sharing tightly-coupled L1 SPM through a low-latency interconnect
  - Common architectural pattern!



Embedded domain:
GreenWaves' GAP8



Many-core SoCs:
Kalray's MPPA-256

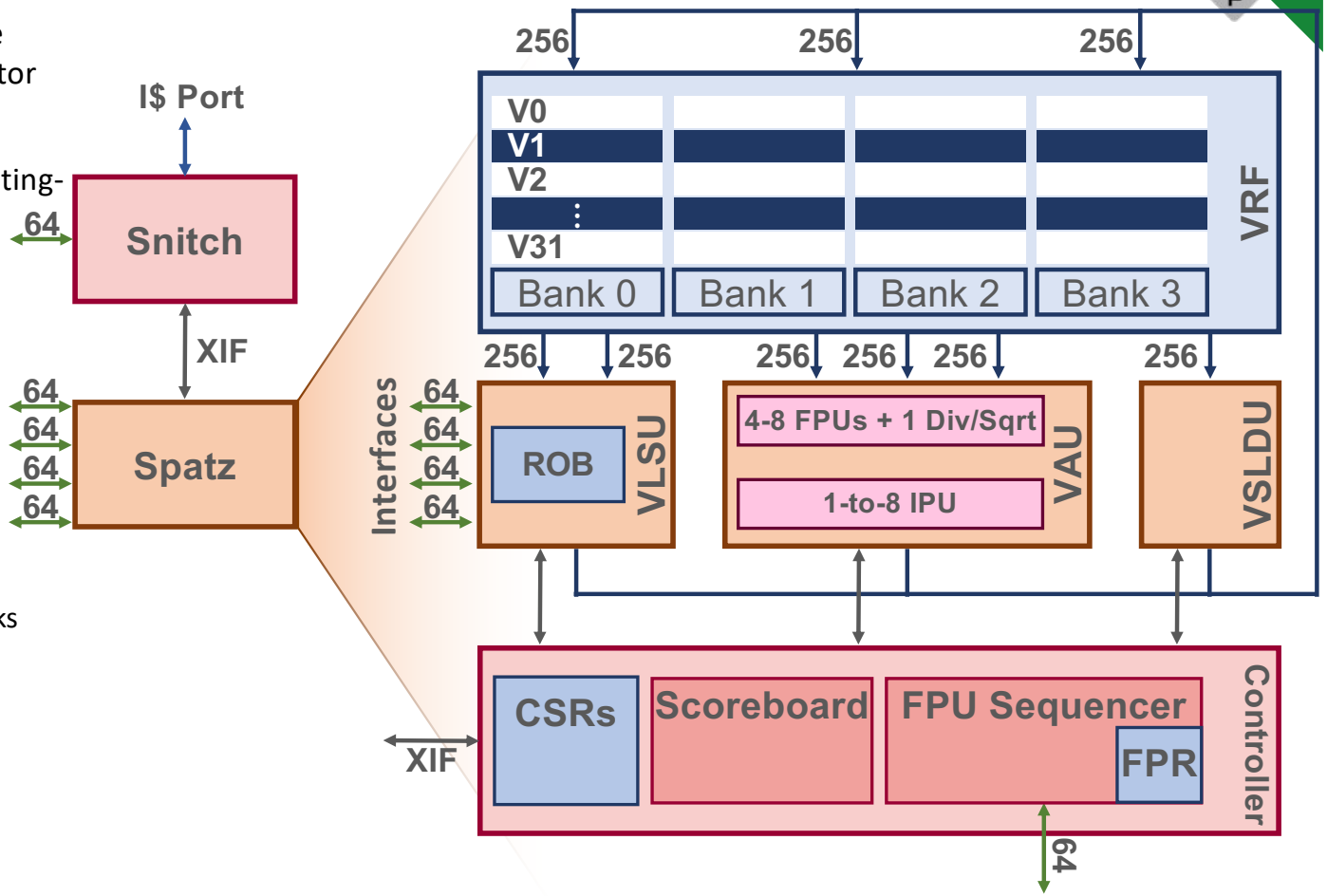


GPUs:
NVIDIA's H100 SMs

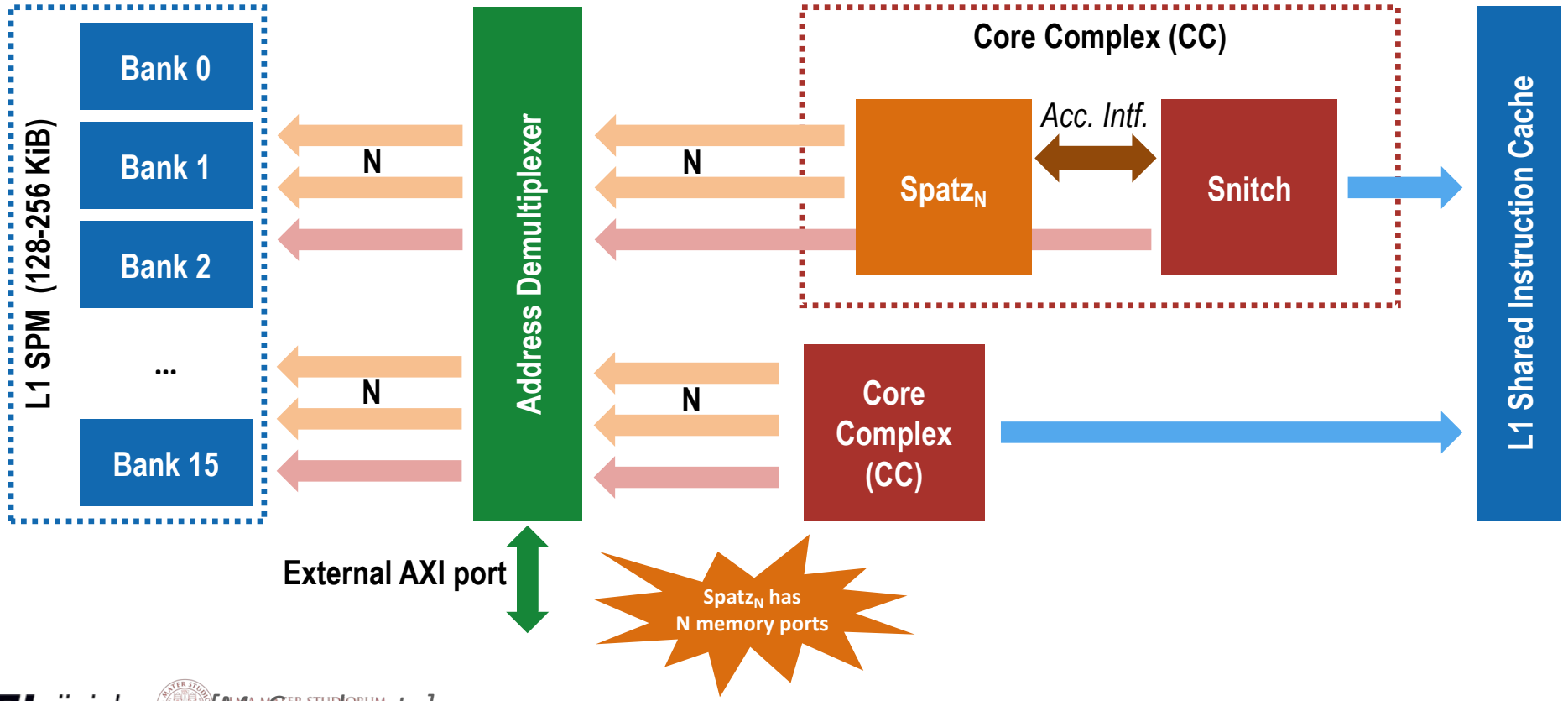
- The efficiency of this building block is **key** for building efficient large-scale systems

Spatz: Embedded Multi-precision RISC-V Vector Processor

- A compact 64-bit floating-point-capable vector processor based on RISC-V's Vector Extension.
- Supports 8, 16, 32, 64-bit integer & floating-point operations
- **Instruction Flow**
  - **Snitch**: pre-decodes vector instructions, dispatches to vector unit
  - **Spatz:** decodes instructions, execute, and acknowledges completion to Snitch
- **Vector Register File (VRF)**
  - VLEN = 512
  - Multi-banked, multi-ported 4× 3R1W banks
- **Functional Units**
  - **VLSU** – Vector Load/Store Unit
  - **VAU** – Vector Arithmetic Unit
  - **VSLDU** – Vector Slide Unit

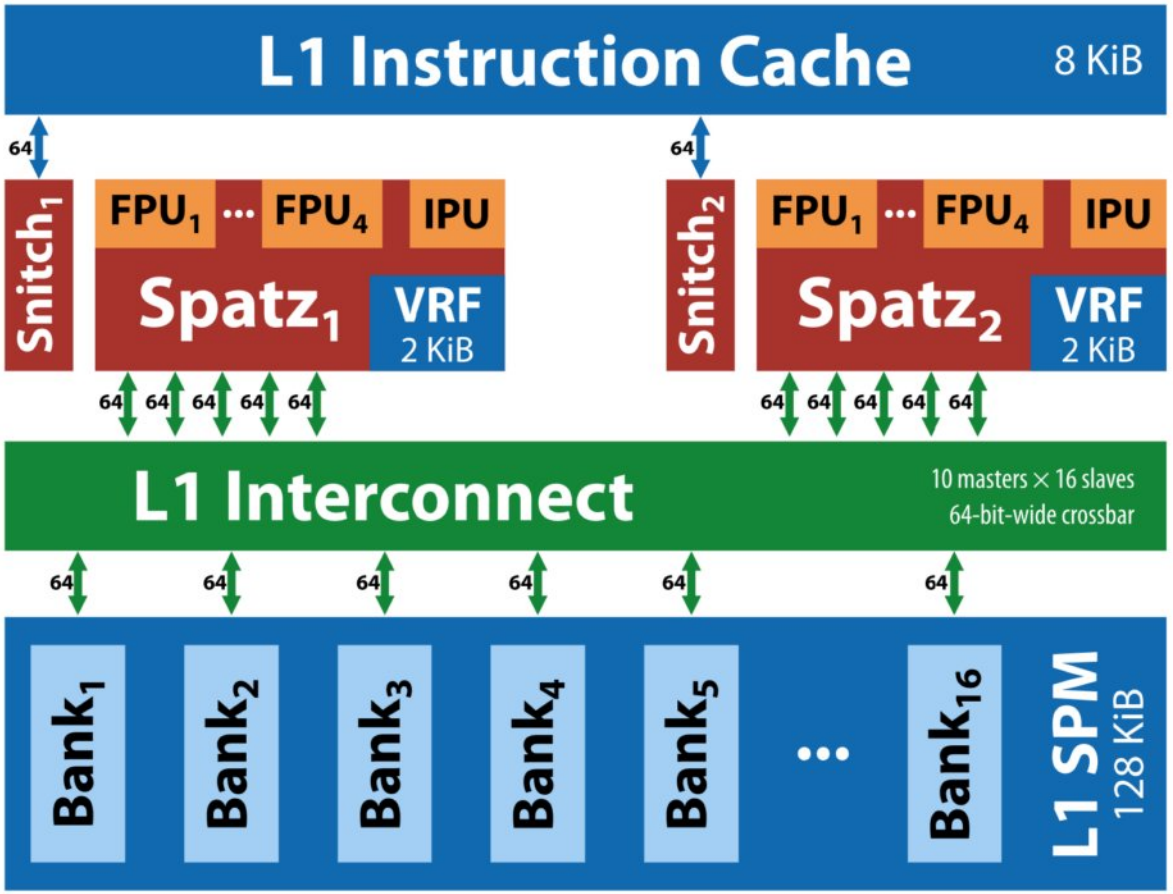


Spatz integration within a shared-L1 cluster

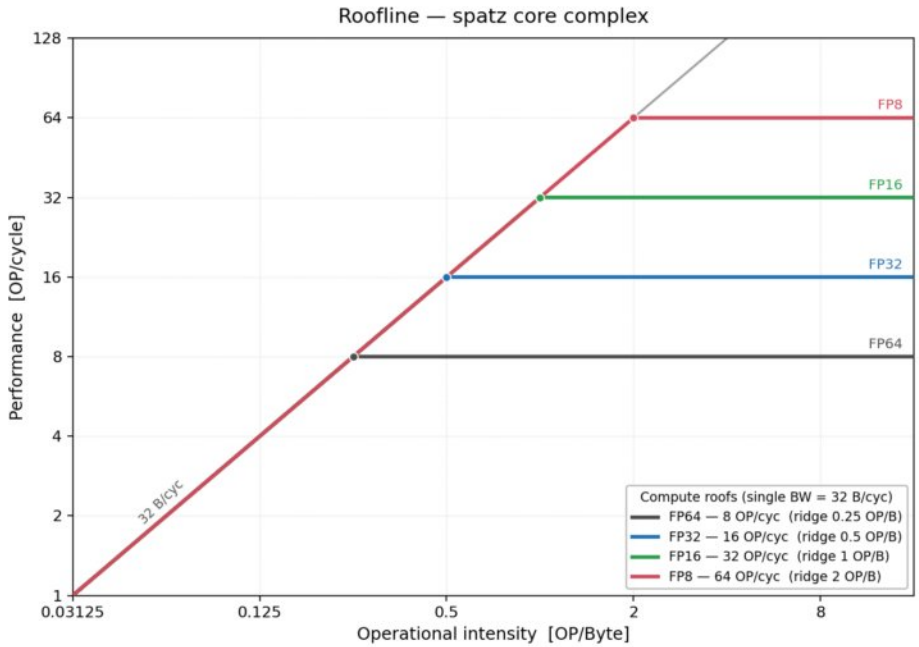


The Spatz Compute Cluster

- Two Vector core complexes
- Potentially, each CC can handle a different “thread”



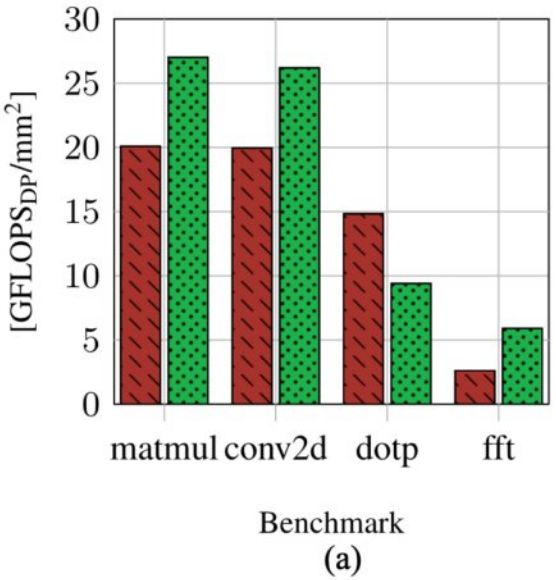
The Spatz Roofline and Benchmarks

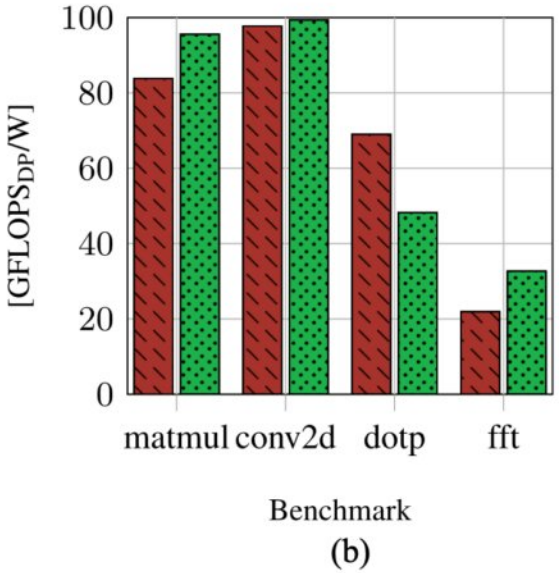


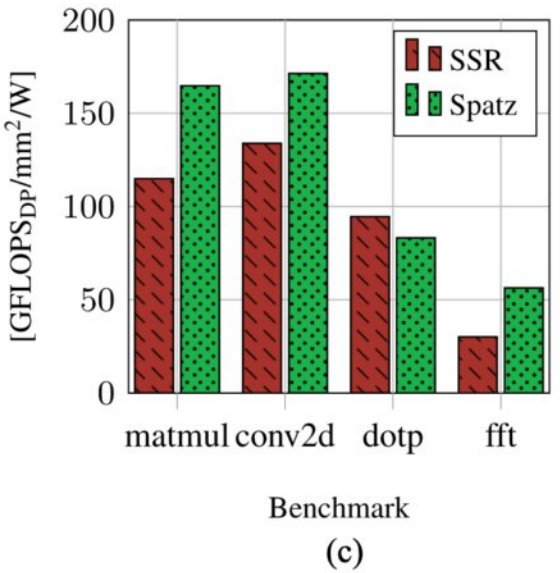
Benchmark	n	Perf. [FLOP/cycle]	Util. [%]
matmul	16	11.57	72.3
	32	15.00	93.8
	64	15.67	97.9
wid-matmul$_{16}$	64	57.53	89.9
	128	61.52	96.1
wid-matmul$_{8}$	64	112.9	88.2
	128	121.8	95.2
conv2d	32	14.91	93.2
	64	15.20	95.0
dotp	256	1.67	10.4
	4096	5.45	34.0
fft	128	3.43	34.2
	256	4.01	40.1

`github.com/pulp-platform/spatz`

Comparison against Snitch-based Cluster



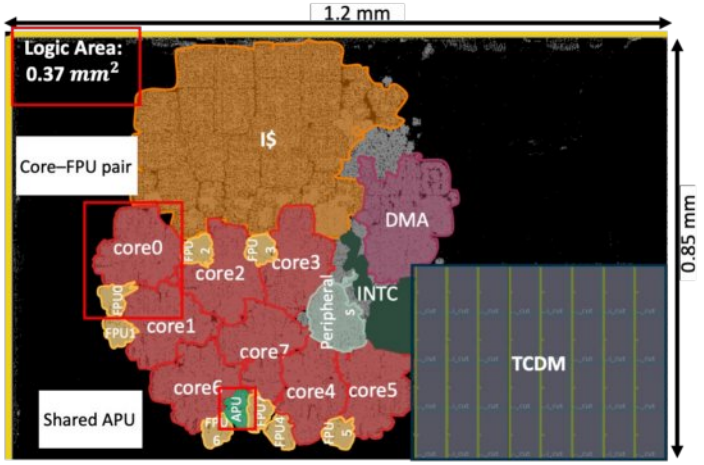




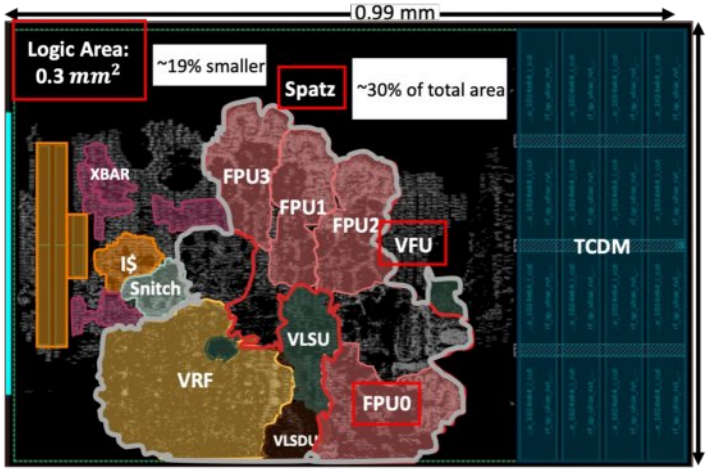
- Same number of functional units
- Same cumulative BW towards the L1: 64B/cyc

PULP vs Spatz: Power, Performance, Area

PULP Cluster



Spatz Cluster



- Synth and P&R in GF12 tech, SS, 0.72V, 125C;
- Max Operating Frequency @0.72V: **600 MHz (PULP) – 1 GHz (Spatz)**
- Area: **0.37mm2 (PULP) – 0.3 mm2 (Spatz)**
- Power @max freq, @0.72V: **68 mW (PULP) – 113 mW (Spatz)**
- Idle Power: **13.8 mW (PULP) - 22.4 mW (Spatz)**

Kiamarzi, A., et al.. Threads or Vectors? Evaluating SPMD and Vector Accelerators for Resource Constrained RISC-V Architectures. In Proceedings of the 23rd ACM International Conference on Computing Frontiers (pp. 71-79).

Threads or Vectors for Edge Workloads?

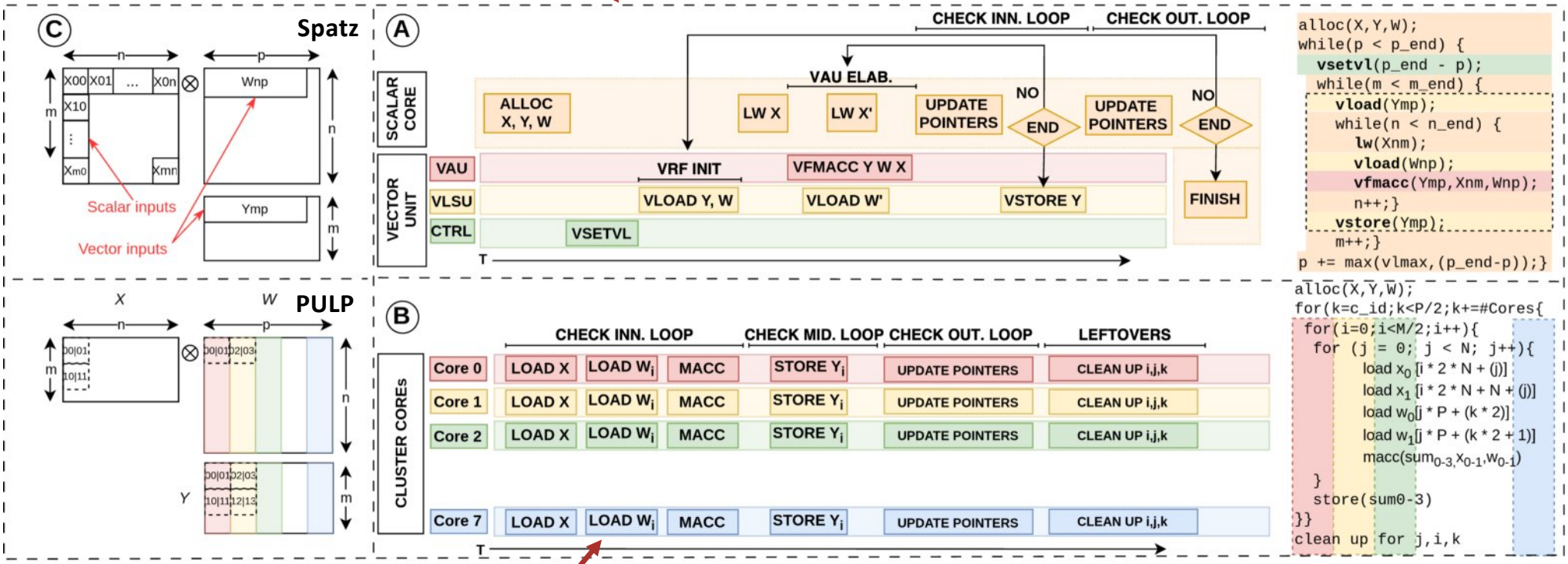
GEMM Execution Flow: A PULP/Spatz Perspective

Mitigate the Von Neumann bottleneck

Data tile organization

Temporal execution flow

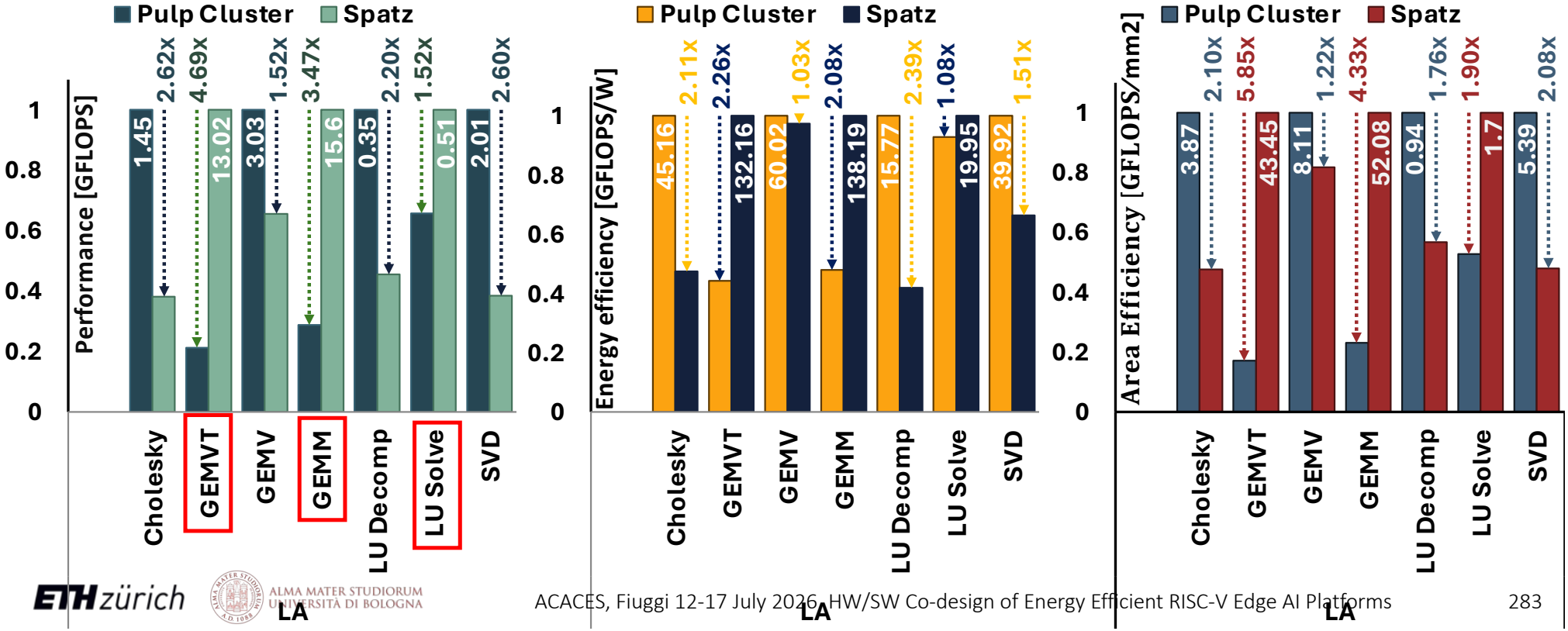
Pseudo code snippets



Von Neumann bottleneck

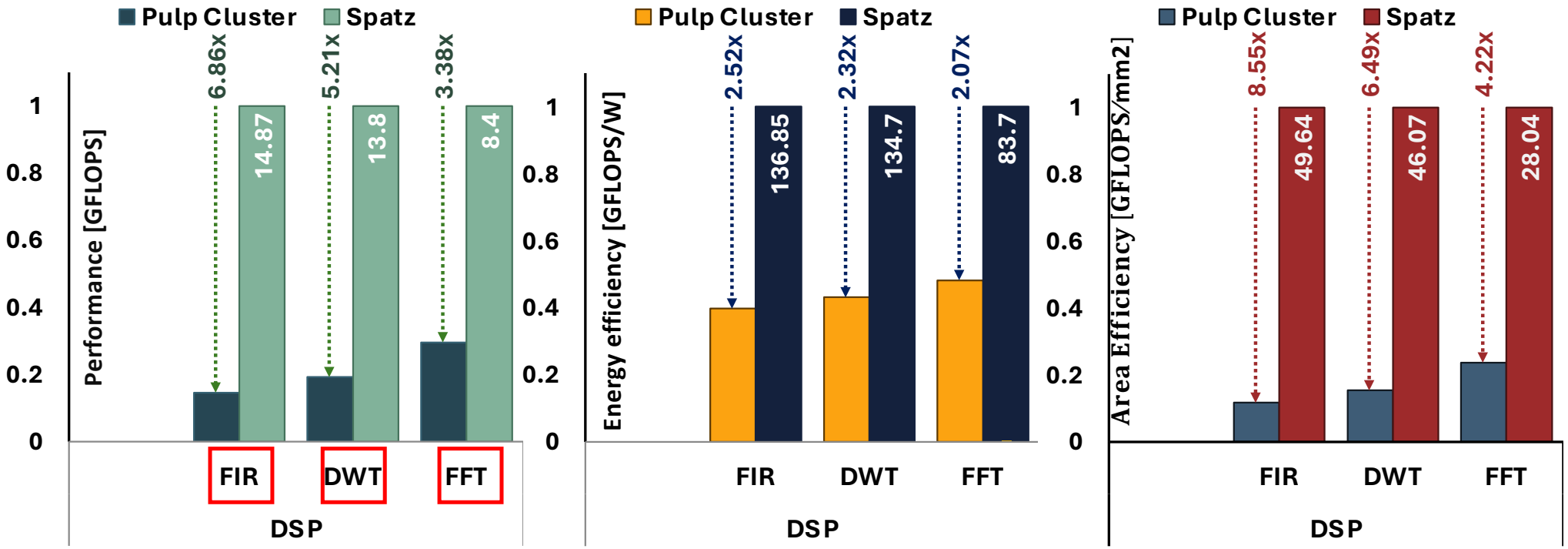
Post-layout PPA comparison (LA)

- Irregular memory access (e.g. Cholesky) and reduction-heavy computation (e.g. SVD) kernels perform better on the PULP Cluster.
- Compute-intensive kernels with regular memory access perform better on Spatz.
- GEMV and GEMVT are both compute-intensive and reduction-heavy, but different memory access layouts make each one favor a different platform.



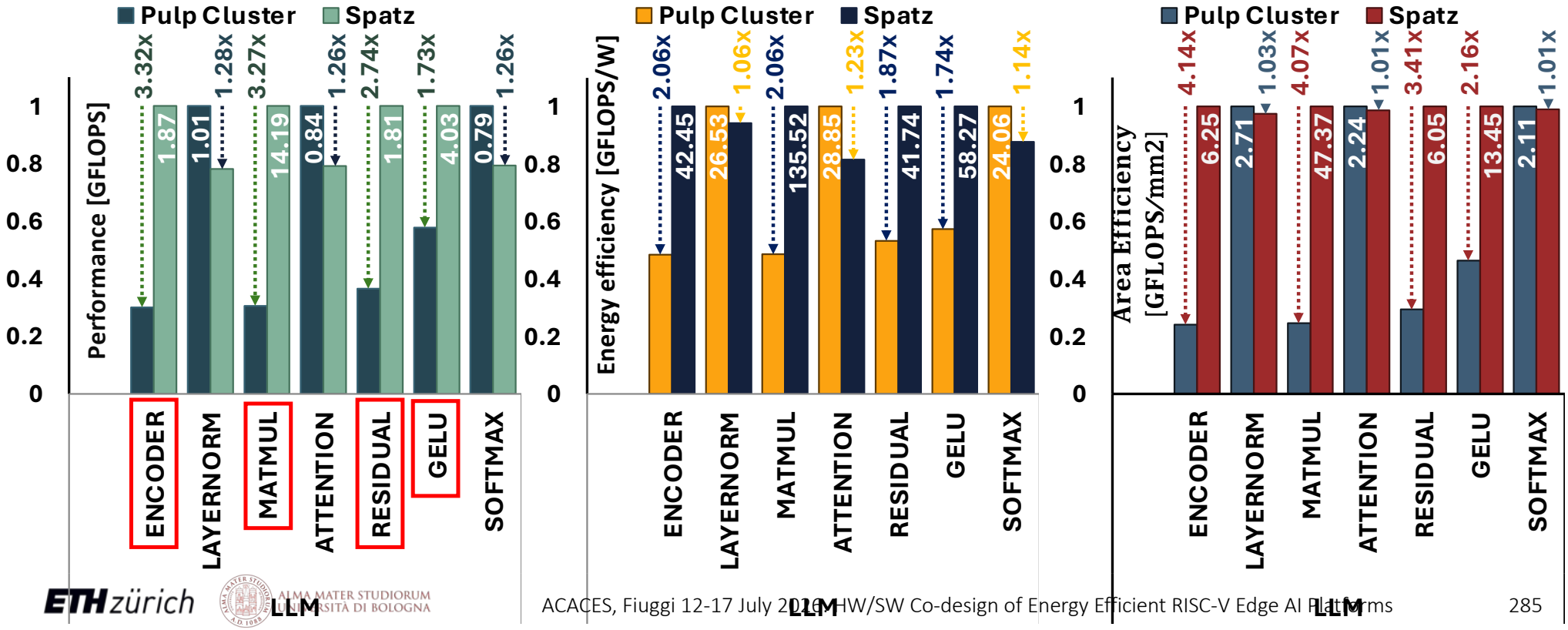
Post-layout PPA comparison (DSP)

- Irregular memory access latency can be hidden when followed by large computation and high data reuse, as in FFT, making vector execution still competitive in such cases.



Post-layout PPA comparison (LLM)

- Since memory access and computation are overlapped, memory-intensive kernels such as Encoder and Residual perform better on Spatz.
- Kernels with dependent nested loop patterns perform better on the PULP Cluster, as the number of operations reduces at each iteration leaving the FPU underutilized on Spatz.

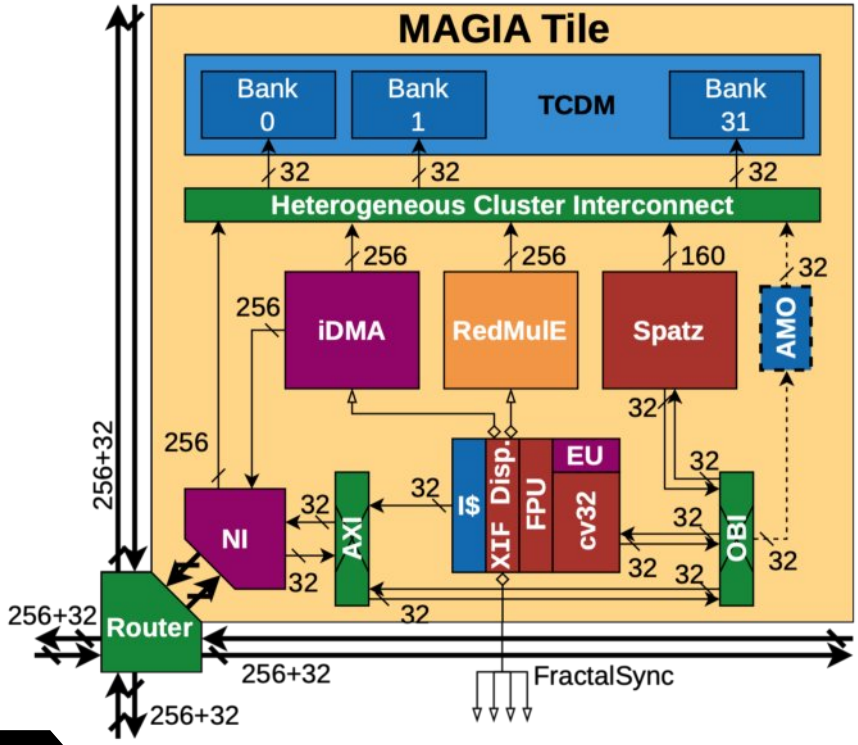


Threads or Vectors? It Depends..

- **Spatz (Vector SIMD Processor)**
  - Best for regular, compute intensive kernels with regular/predictable memory access
  - Very high utilization on kernels with high OP/B
  - On these kernels, significant improvements over PULP cluster
- **PULP (Multi-core SPMD)**
  - Best for better control-heavy and irregular workloads
  - Handles irregular memory access, reductions, and dependent nested loop well
  - On these workloads, significant improvements over Spatz
- **So..?**
  - Reconfigurable thread/vector compute clusters?
  - Boosting Vector Paradigm with RISC-V Matrix Extensions? IME, VME, AME..?
  - Heterogeneous composition of scaled-up multi-tile systems?

MAGIA: An Open-Research Platform For Scaled Systems

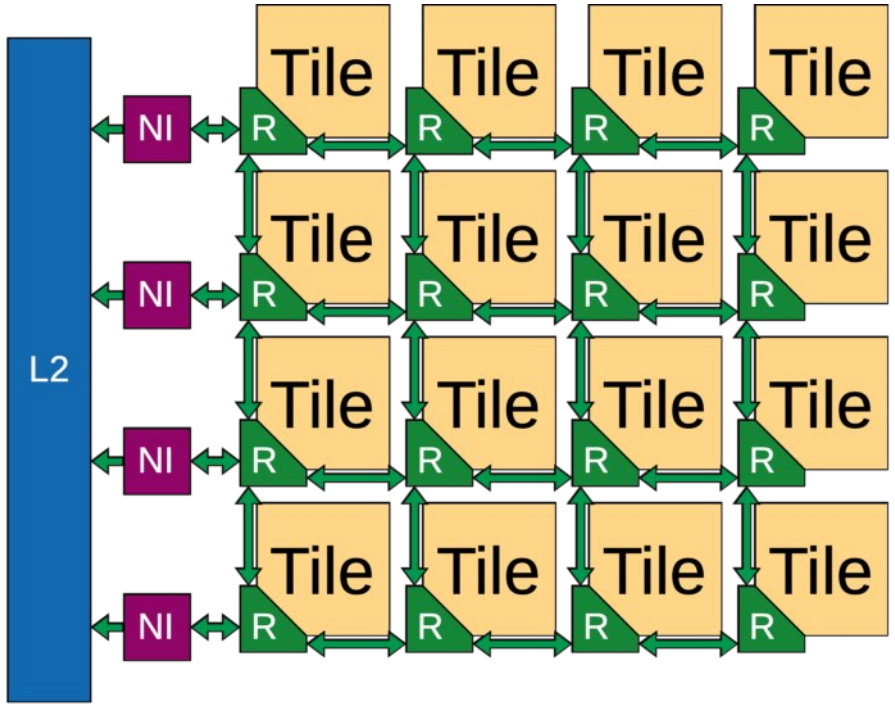
- **Each tile is highly configurable and may feature:**
  - Lightweight **RV32IMAFC RISC-V** control core with 16 KiB of I$, FPU, EU and CV-X-IF - **cv32e40x**
  - Tile-local L1 **SPM** (up to 1 MiB) - TCDM
  - Systolic **GEMM accelerator** - **RedMulE**
  - 4 transprecision 32-bit FPU **VPU** - **Spatz**
  - **256-bit AXI4** DMA - **iDMA**
  - Local and NoC **interconnect** - **AMO**, **NI**, **Router**, **HCI**
  - More units are being added as configurable choices



github.com/pulp-platform/MAGIA

MAGIA: System-level View

- **MAGIA is organized as a 2D mesh of (Heterogeneous) tiles:**
  - Square meshes from **2×2** to **32×32**
  - Global NoC interconnect - **FlooNoC**
  - **Shared L2** memory
- **A dedicated SDK facilitates programming**
  - Contains common computational kernels (e.g. *GEMM*, *FFT*)
- **C++/Python models for high-level architectural exploration - GVSoC**



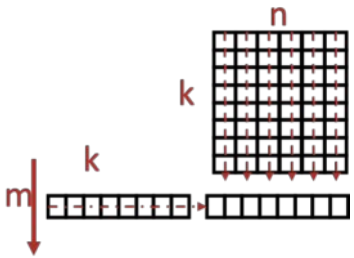
`github.com/pulp-platform/magia-sdk`

A New Frontier in RISC-V: Matrix Extensions

- **Why Matrix Extensions?**
- **Driven by AI:** GEMM is one of the most common operations in AI algorithms
- Programmers are implementing AI algorithms in many domains
  - SW portability is paramount today
- Matrix multiplication is a bottleneck in many AI algorithms (…and in other types of workloads as well)
  - Optimal Matrix execution on processors required to boost performance
    - …from IoT to HPC
- **Other ISAs already implement them**
  - IBM Matrix-Multiply Assist (2020)
  - Intel Advanced Matrix Extension (2020)
  - Arm Scalable Matrix Extension (2021)

Matrix: A Wide Design Space

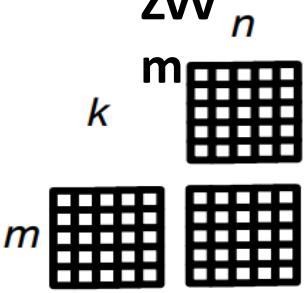
Dot-product



- No new state
- Just one additional dot-p insn

IME

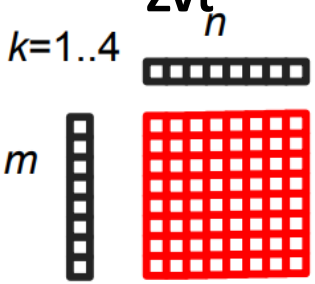
Zvv



- No new state
- Integrated within vector architecture
- VRF interpreted as 2D tiles

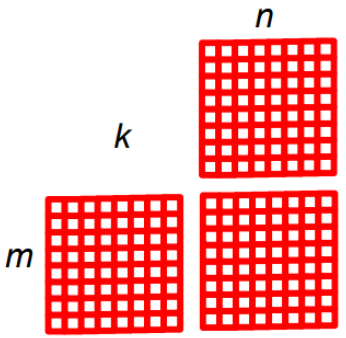
VME

Zvt



- New state for accumulators
- Within RVV
- Outer-product oriented Matrix engine

AME



- Completely separate co-processor
- Dedicated state for input matrix regs and accumulators
- Can work without RVV

The Archi Difference is where the data lives

	Operands A, B	Accumulator C	Abstraction
BDP	RVV regs	RVV regs	vector-matrix primitives
IME	RVV regs (tiled)	RVV regs (tiled)	tile GEMM
VME	RVV regs	Acc regs	tile GEMM
AME	Matrix regs	Acc regs	tile GEMM

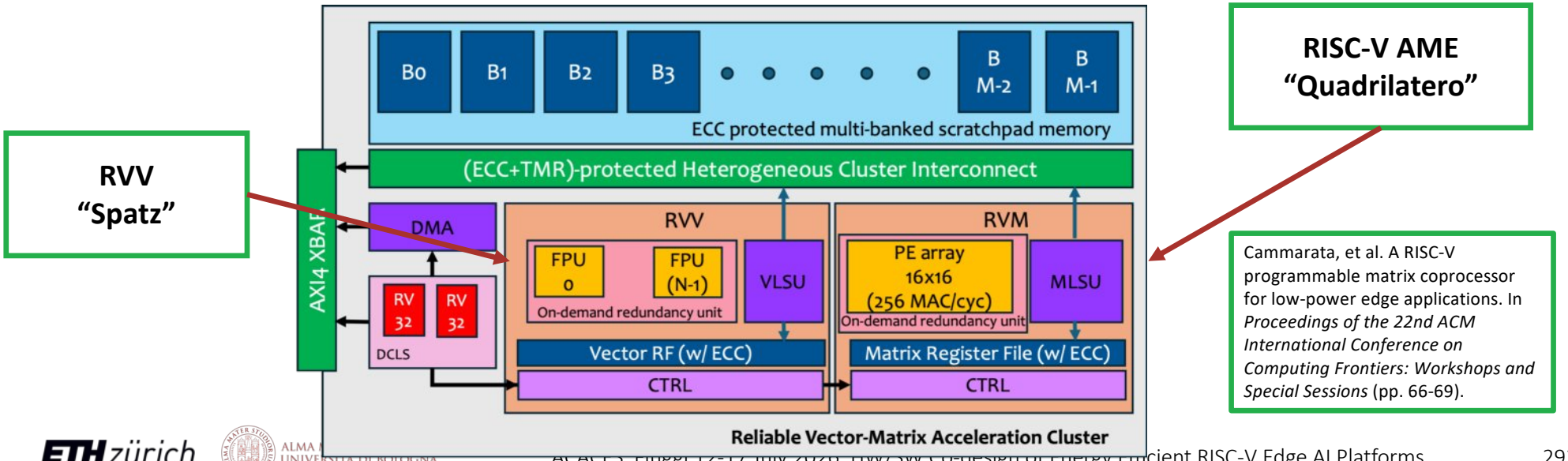
- This affects how you shape you matmul kernel code

	New state	Micro-archi hints
BDP	No	simplicity, minmal uarchi change
IME	just minimal control	richer abstraction than BDP
VME	dedicated accumulators	aggressive performance boost (outerp-product oriented..)
AME	new matrix state (input + accum.)	a distinct subsystem – not necessarily coupled with RVV

- Not mutually exclusive

Not Only On-earth computation..

- **RISC-V is really appealing for the space landscape**
  - AI-enabled autonomous satellites, active 5G/gNB on satellites, autonomous space robotics..
- **Performance-Efficiency-Flexibility (..and reliability/fault-tolerance!)**
  - Space is a harsh environments (especially at >LEO orbits): thermal and radiation effects
  - We are building a reliable/fault-tolerant RV Matrix-Vector Compute tile for Edge AI in Space
    - …and will demonstrate this in a SoC in an advanced tech node



Vector-Matrix Co-operation on Flash-Attention Layers

- **Mix of element-wise vectorial operations and pure GEMM operations**
- **Nice collaboration between Vector-Matrix**
  - Using the shared L1 memory buffer
  - …but we are also thinking about more tightly-coupled approached → more efficient
- **Idea**
  - MatMul on AME + Sum tree on RVV
  - Softmax/exp/non-linearities on RVV (w/ custo extensions)
  - Final projection on AME + RVV for sum
- **Lowering through MLIR**
  - Simulation framework on GVSOC
  - MLIR + LLVM for lowering linalg to Vector-Matrix

Algorithm 1: FlashAttention-2 and 3 forward pass

Require: Matrices $Q, K, V \in \mathbb{R}^{LEN \times d}$

Divide Q into $T_r = \lceil \frac{LEN}{B_r} \rceil$ blocks of size $B_r \times d$ each, and divide K and V into $T_c = \lceil \frac{LEN}{B_c} \rceil$ blocks of size $B_c \times d$ each;

for $1 \le i \le T_r$ **do**

  Initialize $old\_m, old\_l = (-\infty), (0) \in \mathbb{R}^{B_r}$;

  Initialize $old\_O = (0) \in \mathbb{R}^{B_r \times d}$;

  for $1 \le j \le T_c$ **do**

    $S = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$;

    $local\_m = rowmax(S) \in \mathbb{R}^{B_r}$;

    $new\_m = max(local\_m, old\_m) \in \mathbb{R}^{B_r}$;

    $a = old\_m - new\_m \in \mathbb{R}^{B_r}$;

    $b = exp(\frac{a}{\sqrt{d}}) = exp2(\frac{\log_2 e}{\sqrt{d}} a) \in \mathbb{R}^{B_r}$;

    $N = S - new\_m \in \mathbb{R}^{B_r \times B_c}$;

    $P = exp(\frac{N}{\sqrt{d}}) = exp2(\frac{\log_2 e}{\sqrt{d}} N) \in \mathbb{R}^{B_r \times B_c}$;

    $local\_l = rowsum(P) \in \mathbb{R}^{B_r}$;

    $new\_l = old\_l \times b + local\_l \in \mathbb{R}^{B_r}$;

    $local\_O = PV_j \in \mathbb{R}^{B_r \times d}$;

    $new\_O = diag(b) old\_O + local\_O \in \mathbb{R}^{B_r \times d}$;

    $old\_m = new\_m$;

    $old\_l = new\_l$;

    $old\_O = new\_O$;

  end for

  $O_i = diag(old\_l)^{-1} old\_O \in \mathbb{R}^{B_r \times d}$;

end for

Conclusion

- **RISC-V a key enabler of domain-specialized Edge AI/DSP processors**
- **Threads/Vectors/Streams or Heterogeneity?**
  - No one-fit-all solution, it really depends on area/power budget, application/workload constraints
- **Some observed points**
  - MIMD/VLEM cluster
    - High Energy efficiency for low-cost IoT solutions, but scales badly to higher compute requirements
  - Streaming processors based on RISC-V ISA
    - High utilization, scales well in clusters with high BW towards L1/other tiles, higher power than others
  - Heterogeneous RISC-V cluster
    - Better performance with accelerators, less flexible, move the design challenges towards the interconnect, BW..
  - RISC-V Embedded Vector Processors
    - Highest peak energy efficiency, scalable with RISC-V IME/VME extensions, sensitive to workload characteristics
- **What's Next?**

Hope This Course Was Helpful for You

- **Covered the basiscs of RISC-V, advanced architectures with different paradigms**
- **All the platforms/examples seen are based on Open-source Hardware**
  - You can try them out, deepen your study with hands-on sessions
    - Every platform has readme sections and in some cases video-tutorials
  - Use them for your research!
- **Useful links for further material**
  - https://pulp-platform.org/index.html : news, on-going projects, PULP trainings, slides of conference papers, tutorials, workshops, etc
  - PULP Platform @ Linkedin: news, events, new papers, etc
  - https://www.youtube.com/@PULPPlatform : video tutorials, hands-on sessions, demonstrators, etc
  - https://github.com/pulp-platform: here is where we put all our work (SW, HW, compilers, etc)

Thank you!

Credits

- **The slides presented in this course are inspired by the following sources:**

- Conference, tutorial, and workshop slides by Dr. Angelo Garofalo
- Conference, tutorial, keynote materials by the members of the PULP research group
- Undergraduate and graduate courses taught at the University of Bologna by prof. Francesco Conti and Dr. Angelo Garofalo
- Courses taught at ETH Zurich by prof. Luca Benini

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

References

Part 1: RISC-V ISA and Edge RISC-V Processors

- RISC-V ISA: https://docs.riscv.org/reference/isa/v20260120/index.html
- Ratified extensions: https://riscv.org/specifications/ratified/
- Hennessy, Patterson, Computer Architecture: A Quantitative Approach, fifth edition.
- Schiavone, P. D., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., & Benini, L. (2017, September). Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS) (pp. 1-8). IEEE.
- Ibex processor: https://lowrisc.org/ibex/
- Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., ... & Benini, L. (2017). Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE transactions on very large scale integration (VLSI) systems, 25(10), 2700-2713.
- Cv32e4 processor: https://docs.openhwgroup.org/projects/core-v-verif/en/latest/intro.html

References

Part 2: ISA Extensions for Domain-specialized RISC-V processors (DSP/AI)

- Garofalo, A., Rusci, M., Conti, F., Rossi, D., & Benini, L. (2019). PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, *378*(2164), 20190155.
- Bruschi, N., Garofalo, A., Conti, F., Tagliavini, G., & Rossi, D. (2020, May). Enabling mixed-precision quantized neural networks in extreme-edge devices. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (pp. 217-220).
- Garofalo, A., Tagliavini, G., Conti, F., Benini, L., & Rossi, D. (2021). XpulpNN: Enabling energy efficient and flexible inference of quantized neural networks on RISC-V based IoT end nodes. IEEE Transactions on Emerging Topics in Computing, 9(3), 1489-1505.
- Ottavi, G., Garofalo, A., Tagliavini, G., Conti, F., Benini, L., & Rossi, D. (2020, July). A mixed-precision RISC-V processor for extreme-edge DNN inference. In 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (pp. 512-517). IEEE.
- Examples of RISC-V ISA extensions in different domains which are not covered during the lectures (e.g., security, audio, communication) can be easily found through a quick research on “google scholar”.

References

Part 3: PULP Cluster and VLEM modes

- Rossi, D., Pullini, A., Loi, I., Gautschi, M., Gürkaynak, F. K., Teman, A., ... & Benini, L. (2017). Energy-efficient near-threshold parallel computing: The PULPv2 cluster. Ieee Micro, 37(5), 20-31.
- Garofalo, A., & Benini, L. (2025). Leveraging RISC-V for HW/SW Co-Design of Flexible and Efficient TinyML SoCs. IEEE Design & Test.
- Rossi, D., Loi, I., Haugou, G., & Benini, L. (2014, May). Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters. In Proceedings of the 11th ACM Conference on Computing Frontiers (pp. 1-10).
- Ottavi, G., Garofalo, A., Tagliavini, G., Conti, F., Di Mauro, A., Benini, L., & Rossi, D. (2023). Dustin: A 16-cores parallel ultra-low-power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode. IEEE Transactions on Circuits and Systems I: Regular Papers, 70(6), 2450-2463.
- Burrello, A., Garofalo, A., Bruschi, N., Tagliavini, G., Rossi, D., & Conti, F. (2021). Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. IEEE Transactions on Computers, 70(8), 1253-1268.
- Glaser, F., Tagliavini, G., Rossi, D., Haugou, G., Huang, Q., & Benini, L. (2020). Energy-efficient hardware-accelerated synchronization for shared-L1-memory multiprocessor clusters. IEEE Transactions on Parallel and Distributed Systems, 32(3), 633-648.

References

Part 4: Heterogeneous RISC-V Clusters

- Garofalo, A., Tortorella, Y., Perotti, M., Valente, L., Nadalini, A., Benini, L., ... & Conti, F. (2022). DARKSIDE: A heterogeneous RISC-V compute cluster for extreme-edge on-chip DNN inference and training. *IEEE Open Journal of the Solid-State Circuits Society*, *2*, 231-243.
- Conti, F., Paulin, G., Garofalo, A., Rossi, D., Di Mauro, A., Rutishauser, G., ... & Benini, L. (2023). Marsellus: A heterogeneous RISC-V AI-IoT end-node SoC with 2–8 b DNN acceleration and 30%-boost adaptive body biasing. *IEEE Journal of Solid-State Circuits*, *59*(1), 128-142.
- Silvano, C., Ielmini, D., Ferrandi, F., Fiorin, L., Curzel, S., Benini, L., ... & Perri, S. (2025). A survey on deep learning hardware accelerators for heterogeneous hpc platforms. ACM Computing Surveys, 57(11), 1-39.
- Islamoglu, G., Scherer, M., Paulin, G., Fischer, T., Jung, V. J., Garofalo, A., & Benini, L. (2023, August). Ita: An energy-efficient attention and softmax accelerator for quantized transformers. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)* (pp. 1-6). IEEE.
- Belano, A., Tortorella, Y., Garofalo, A., Benini, L., Rossi, D., & Conti, F. (2025). A flexible template for edge generative ai with high-accuracy accelerated softmax & gelu. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*.
- Le Gallo, M., Khaddam-Aljameh, R., Stanisavljevic, M., Vasilopoulos, A., Kersting, B., Dazzi, M., ... & Sebastian, A. (2023). A 64-core mixed-signal in-memory compute chip based on phase-change memory for deep neural network inference. *Nature Electronics*, *6*(9), 680-693.
- Garofalo, A., Ottavi, G., Conti, F., Karunaratne, G., Boybat, I., Benini, L., & Rossi, D. (2022). A heterogeneous in-memory computing cluster for flexible end-to-end inference of real-world deep neural networks. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 12(2), 422-435.
- Bruschi, N., Tagliavini, G., Garofalo, A., Conti, F., Boybat, I., Benini, L., & Rossi, D. (2023, April). End-to-end DNN Inference on a massively parallel analog in memory computing architecture. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1-6). IEEE.

References - 2

- Conti, F., Garofalo, A., Rossi, D., Tagliavini, G., & Benini, L. (2025). Open Source Heterogeneous SoCs for Artificial Intelligence: The PULP Platform experience. IEEE Solid-State Circuits Magazine, 17(2), 49-60.
- Cammarata, D., Garofalo, A., & Benini, L. (2026). O-POPE: High-Frequency Pipelined Outer Product based GEMM acceleration with minimal buffering overhead. arXiv preprint arXiv:2606.02333.
- Sinigaglia, M., Garofalo, A., Cereda, E., Tedeschi, R., Ciani, M., Tesfai, H. T., ... & Rossi, D. (2026). A Heterogeneous System-on-Chip with an 83 GFLOp/s, 1.2 TFLOp/s/W Parallel Programmable Accelerator for Real-time On-device Learning in Autonomous Nano-UAVs. IEEE Open Journal of the Solid-State Circuits Society.
- Ferro, E., Ottaviano, A., Lammie, C., İslamoğlu, G., Garofalo, A., Conti, F., ... & Boybat, I. (2026). EAGLE: A Flexible Heterogeneous Analog Compute-In-Memory Architecture With RISC-V Programmable Multi-Core Accelerators. IEEE Transactions on Circuits and Systems I: Regular Papers.
- Verhelst, M., Benini, L., & Verma, N. (2025). How to keep pushing ML accelerator performance? Know your rooflines!. IEEE Journal of Solid-State Circuits.

References

Part 4: Streaming Processors

- Verhelst, M., Benini, L., & Verma, N. (2025). How to keep pushing ML accelerator performance? Know your rooflines!. IEEE Journal of Solid-State Circuits.
- Zaruba, F., Schuiki, F., Hoefler, T., & Benini, L. (2020). Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. IEEE Transactions on Computers, 70(11), 1845-1860.
- Scheffler, P., Zaruba, F., Schuiki, F., Hoefler, T., & Benini, L. (2023). Sparse stream semantic registers: A lightweight ISA extension accelerating general sparse linear algebra. IEEE Transactions on Parallel and Distributed Systems, 34(12), 3147-3161.
- Scheffler, P., Benz, T., Potocnik, V., Fischer, T., Colagrande, L., Wistoff, N., ... & Benini, L. (2025). Occamy: A 432-core dual-chiplet dual-hbm2e 768-dp-gflop/s risc-v system for 8-to-64-bit dense and sparse computing in 12-nm finfet. *IEEE Journal of Solid-State Circuits*, *60*(4), 1324-1338.
- İslamoğlu, G., Bertaccini, L., Prasad, A. S., Conti, F., Garofalo, A., & Benini, L. (2025, July). Mxdotp: A risc-v isa extension for enabling microscaling (mx) floating-point dot products. In 2025 IEEE 36th International Conference on Application-specific Systems, Architectures and Processors (ASAP) (pp. 81-84). IEEE.
- Wang, R., Islamoglu, G., Belano, A., Potocnik, V., Conti, F., Garofalo, A., & Benini, L. (2025). VEXP: A Low-Cost RISC-V ISA Extension for Accelerated Softmax Computation in Transformers. *arXiv preprint arXiv:2504.11227*.

References

Part5: RISC-V Vector Processor

- Hennessy, Patterson, Computer Architecture: A Quantitative Approach, fifth edition, chapter on Vector Processors
- https://docs.riscv.org/reference/isa/extensions/vector/_attachments/riscv-v-spec.pdf
- Perotti, M., Cavalcante, M., Andri, R., Cavigelli, L., & Benini, L. (2024). Ara2: Exploring single-and multi-core vector processing with an efficient RVV 1.0 compliant open-source processor. *IEEE Transactions on Computers*, *73*(7), 1822-1836.
- Perotti, M., Riedel, S., Cavalcante, M., & Benini, L. (2025). Spatz: Clustering compact RISC-V-based vector units to maximize computing efficiency. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 44(7), 2488-2502.
- Wipfli, M., İslamoğlu, G., Purayil, N. K., Garofalo, A., & Benini, L. (2026, April). Vmxdotp: A RISC-V Vector ISA Extension for Efficient Microscaling (MX) Format Acceleration. In 2026 Design, Automation & Test in Europe Conference (DATE) (pp. 1-7). IEEE.
- Cammarata, D., Perotti, M., Bertuletti, M., Garofalo, A., Schiavone, P. D., Atienza, D., & Benini, L. (2025, May). Quadrilatero: A RISC-V programmable matrix coprocessor for low-power edge applications. In *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions* (pp. 66-69).
- Kiamarzi, A., Colmi, S., Tortorella, Y., Garofalo, A., Rossi, D., & Tagliavini, G. (2026, May). Threads or Vectors? Evaluating SPMD and Vector Accelerators for Resource Constrained RISC-V Architectures. In Proceedings of the 23rd ACM International Conference on Computing Frontiers (pp. 71-79).

- **N.B.** Interested people can ask for more references/textbooks by contacting the lecturer