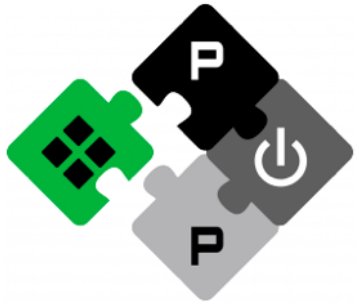


PULP PLATFORM

Open Source Hardware, the way it should be!

# *Parallel Ultra-Low-Power Computing with RISC-V*

## Optimizing CNNs on battery-powered devices


**Giuseppe Tagliavini**

giuseppe.tagliavini@unibo.it


ETH zürich


 http://pulp-platform.org  @pulp_platform  https://www.youtube.com/pulp_platform

# Agenda

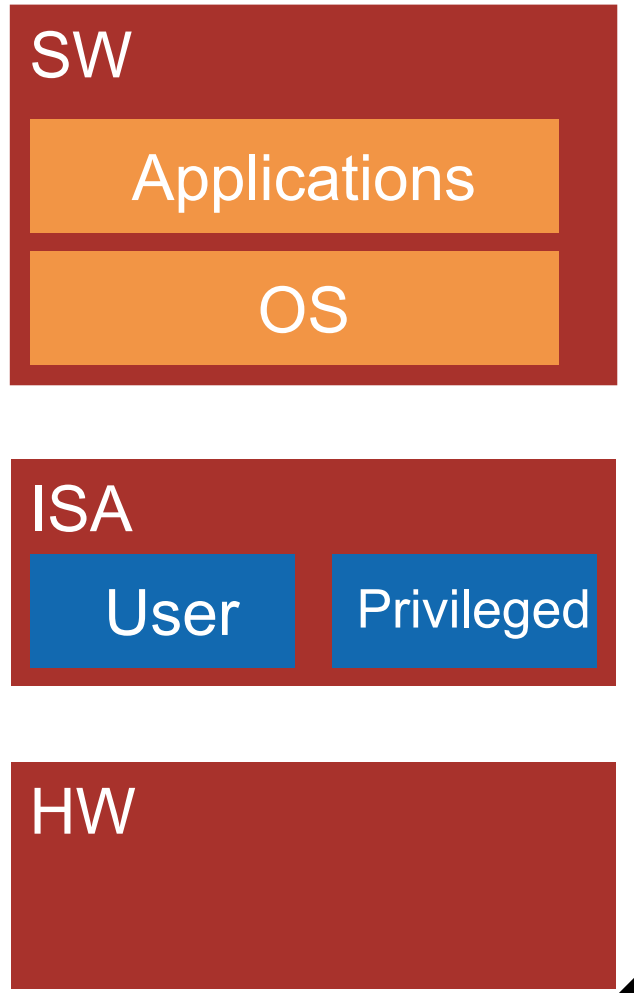
- **Introduction to RISC-V**
- RISC-V design with custom extensions
- Parallel ultra-low-power (PULP) architectures
- Enabling CNNs on PULP platforms

ETH zürich



# RISC-V Instruction Set Architecture

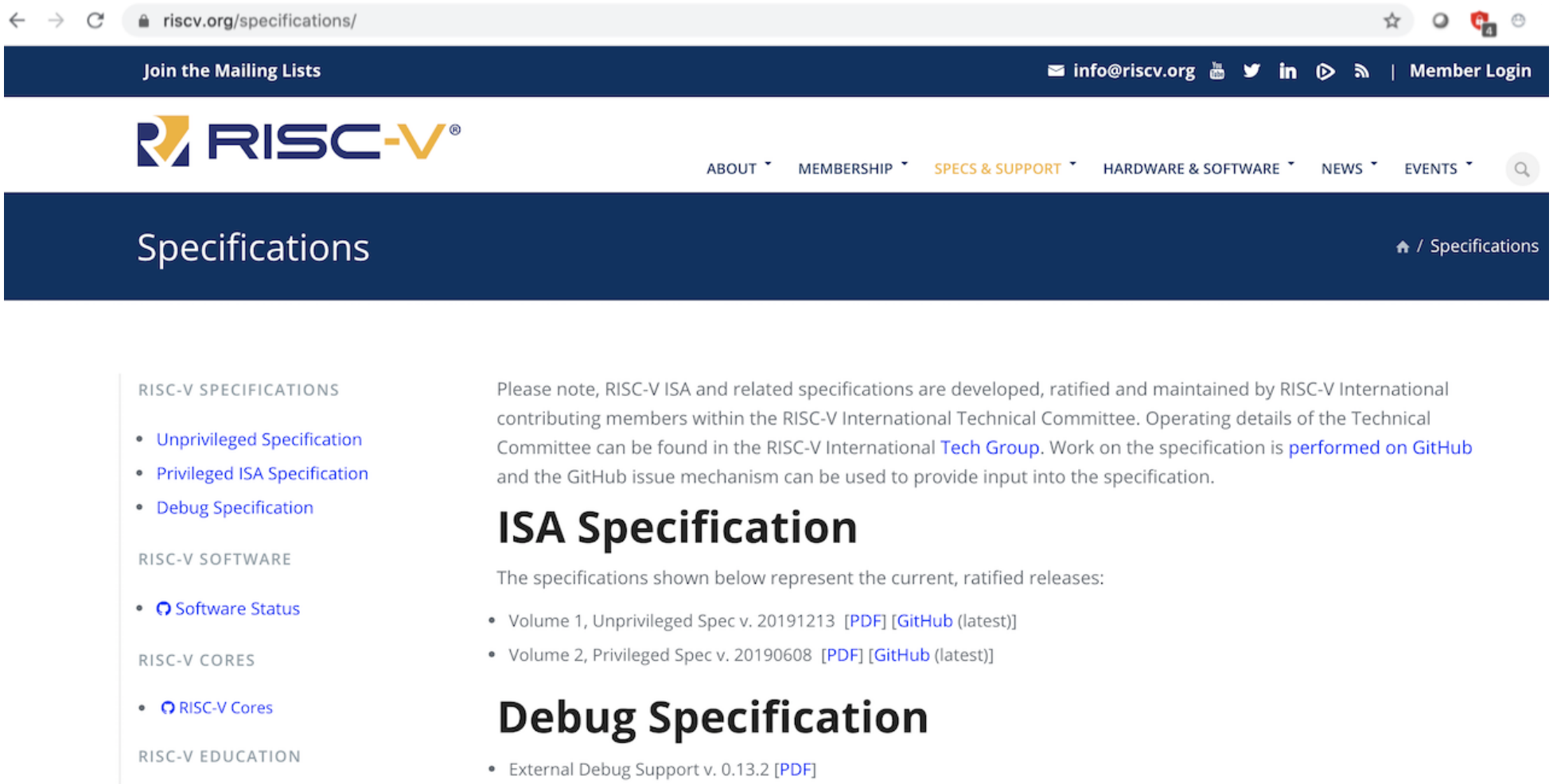
- **Started by UC-Berkeley in 2010**
- **Contract between SW and HW**
  - Partitioned into *User* and *Privileged* specifications
  - External *Debug* interface
- **Standard governed by a foundation**
  - Necessary for the continuity
- **Specification for 32, 64, or 128 bit ISA**
  - ***No implementation, just the ISA***
  - Different implementations (both *open and close source*)



# RISC-V Foundation: 700+ companies


ETH zürich

# RISC-V maintains a set of documents



# RISC-V Ecosystem

- Binutils – upstream
- GCC – upstream
- LLVM – upstream
- Simulator:
  - ”Spike” - reference
  - QEMU, Gem5
- OpenOCD
- OS
  - Linux, sel4, freeRTOS, zephyr
- Runtimes
  - Jikes, Ocaml, Go

*See https://riscv.org/exchange/software/ for an updated list*

# RISC-V ISA is divided into extensions

| | |
|---|---|
| I | Integer instructions (frozen) |
| E | Reduced number of registers |
| M | Multiplication and Division (frozen) |
| A | Atomic instructions (frozen) |
| F | Single-Precision Floating-Point (frozen) |
| D | Double-Precision Floating-Point (frozen) |
| C | Compressed Instructions (frozen) |
| X | Non Standard Extensions |

- **Kept very simple and extendable**
  - Wide range of applications from IoT to HPC
- **RV + word-width + extensions**
  - RV32IMC: 32bit, integer, multiplication, compressed
- **User specification:**
  - Separated into extensions, only I is mandatory
- **Privileged Specification (WIP):**
  - Governs OS functionality: Exceptions, Interrupts
  - Virtual Addressing
  - Privilege Levels



# RISC-V Architectural State

- **There are 32 registers, each 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 floating point operation 32 additional FP registers**
- **Additional Control Status Registers (CSRs)**
  - Encoding for up to 4'096 registers are reserved (not all are used)





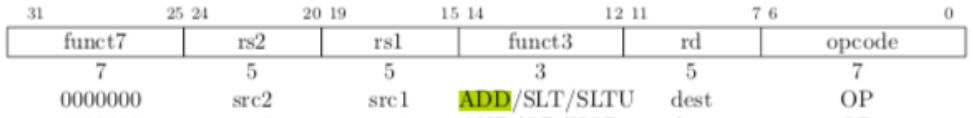
# The FREEDOM in RISC-V is implementation

- **You can access all ISAs 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.




## ARM

**C2.9 ADD**

Add without Carry.

**Syntax**

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

# Agenda

- Introduction to RISC-V
- **RISC-V design with custom extensions**
- Parallel ultra-low-power (PULP) architectures
- Enabling CNNs on PULP platforms

ETH zürich



# RISC-V cores developed at UNIBO+ETHZ

| 32 bit | | | 64 bit |
|---|---|---|---|
| **Low Cost Core** | **DSP Enhanced Core** | **Streaming Compute Core** | **Linux capable Core** |
| ▪ **Zero-riscy**<br>▪ RV32-IMC<br>▪ **Micro-riscy**<br>▪ RV32-IC | ▪ **RI5CY**<br>▪ RV32-IMCFEX<br>▪ SIMD<br>▪ HW loops<br>▪ Bit manipulation<br>▪ Fixed point | ▪ **Snitch**<br>▪ RV32-IMCFDX | ▪ **Ariane**<br>▪ RV64-IC(MA)<br>▪ Full privileged specification |

**Ibex by LowRISC**

**CV32E40P by OpenHW**

**CV6A by OpenHW**

ETH zürich

lowRISC

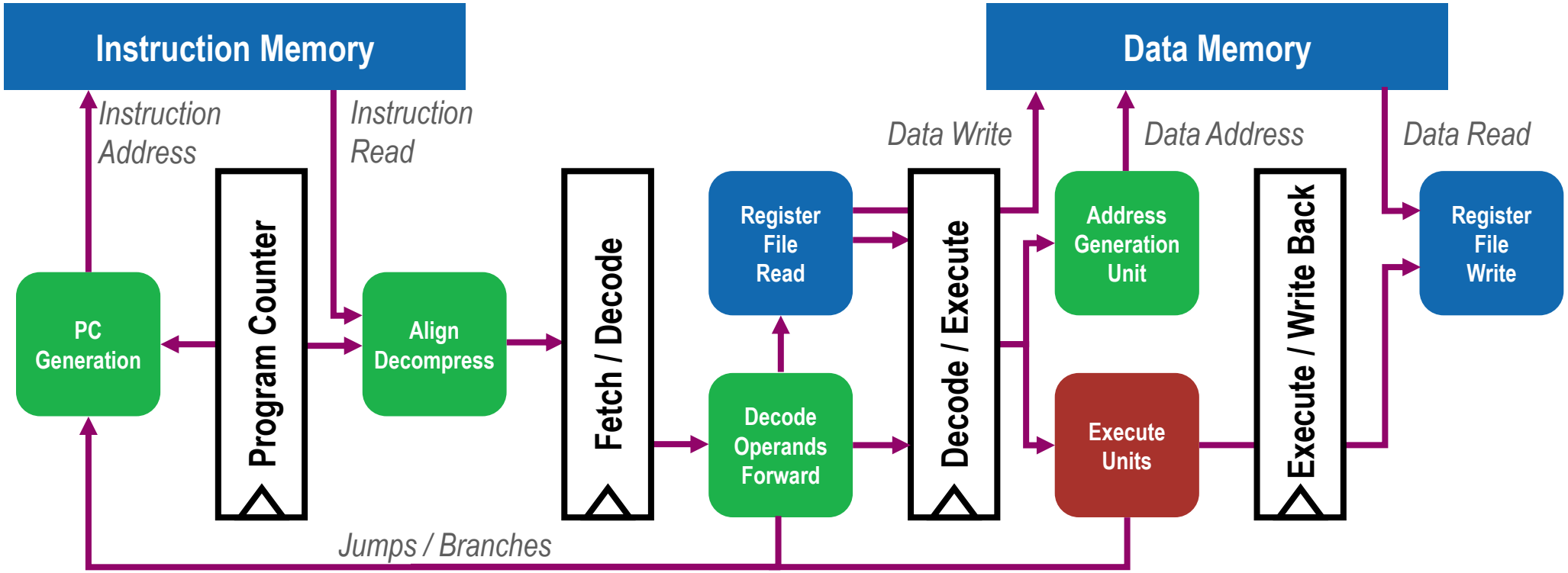
OPENHW GROUP — PROVEN PROCESSOR IP

OPENHW GROUP — PROVEN PROCESSOR IP

# RI5CY / CV32E40P our main 32bit RISC-V core

- **Ibex is suitable for simple applications**
  - Control applications, book-keeping
- **For our research we need more capable cores**
  - Mainly used in clusters for signal processing / machine learning applications
- **Tuned for energy efficiency**
  - Not necessarily low power
- **Make use of custom extensions**
  - The Xpulp extensions enhance the capabilities
  - Several Xpulp extensions in discussions for ratification

# Simplified pipeline for RI5CY / CV32E40P




Website: **https://github.com/openhwgroup/cv32e40p**

# 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 regularly uses these instructions**
  - Great tool for exploring
  - The goal is to help ratify these extensions as standards through working groups

ETH zürich



# Our extensions to RI5CY

- **ALU instructions**
  - Bit manipulation (count, set, clear, leading bit detection)
  - Fused operations: (add/sub & shift)
  - Immediate branch instructions
- **Multiply and accumulate (32x32 bit and 16x16 bit)**
- **SIMD instructions (32 bit = 2x16 bit or 4x8 bit)**
  - add, min/max, dotproduct, shuffle, pack (copy), vector comparison

For 8-bit values the following can be executed in a single cycle (`pv.dotup.b`)

$$Z = D_1 \times K_1 + D_2 \times K_2 + D_3 \times K_3 + D_4 \times K_4$$

- **Post–incrementing load/store instructions**
- **Hardware loops**

# Post-incrementing load / store

- **Automatic address update**
  - Update base register with computed address after the memory access

  ⇒Save instructions to update address register

  - Post-increment:
    - Base address serves as memory address
- **Offset can be stored in:**
  - Register
  - Immediate

**Original RISCV**

```
addi x4, x0, 64
Lstart :
lb x2, 0(x10)
lb x3, 0(x12)
addi x10, x10, 1
addi x12, x12, 1
…..
bne x2,x3, Lstart
```

**Auto-incr load/store**

```
addi x4, x0, 64
Lstart :
lb x2, 0(x10!)
lb x3, 0(x12!)
…..
bne x2,x3, Lstart
```

⇒ save 2 additional instructions to update the read addresses of the operands!

# Hardware loops

- **Hardware loop setup with:**
  - 3 separate instructions

    *lp.start, lp.end, lp.count, lp.counti*

    ⇒No restriction on start/end address

  - Fast setup instructions

    *lp.setup, lp.setupi*

    ⇒Start address= PC + 4

    ⇒End address= start address + offset

    ⇒Counter from immediate/register

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

**Original RISC-V**

```
//initialize counter
mv   x4, 100
// init accumulator
mv   x5, 0
Lstart:
  //decrement counter
  addi  x4, x4, -1
  //load elements from mem
  lw x8, 0(x9)
  lw x10, 0(x11)
  //update memory pointers
  add x9, x9, 4
  add x11, x11, 4
  //mac
  mul x8, x8, x10
  add x5, x5, x8
bne  x4, x0, Lstart
```

**HW Loop Ext**

```
// init accumulator
mv   x5, 0
//set number iterations, start and end of the loop
lp.setupi 100, Lend
  //load elements from mem
  lw x8, 0(x9)
  lw x10, 0(x11)
  //update memory pointers
  add x9, x9, 4
  add x11, x11, 4
  //mac
  mul x8, x8, x10
Lend:  add x5, x5, x8
```

No counter and branch overhead!

# RI5CY ISA extensions improve performance

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

### Baseline

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb     x2, 0(x10)
  lb     x3, 0(x11)
  addi   x10,x10, 1
  addi   x11,x11, 1
  add    x2, x3, x2
  sb     x2, 0(x12)
  addi   x4, x4, -1
  addi   x12,x12, 1
bne      x4, x5, Lstart
```

11 cycles/output

### Auto-incr load/store

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb     x2, 0(x10!)
  lb     x3, 0(x11!)
  addi   x4, x4, -1
  add    x2, x3, x2
  sb     x2, 0(x12!)
bne      x4, x5, Lstart
```

8 cycles/output

### HW Loop

```
lp.setupi 100, Lend
  lb     x2, 0(x10!)
  lb     x3, 0(x11!)
  add    x2, x3, x2
Lend:  sb x2, 0(x12!)
```

5 cycles/output

### Packed-SIMD

```
lp.setupi 25, Lend
  lw     x2, 0(x10!)
  lw     x3, 0(x11!)
  pv.add.b x2, x3, x2
Lend: sw x2, 0(x12!)
```

1,25 cycles/output

# Agenda

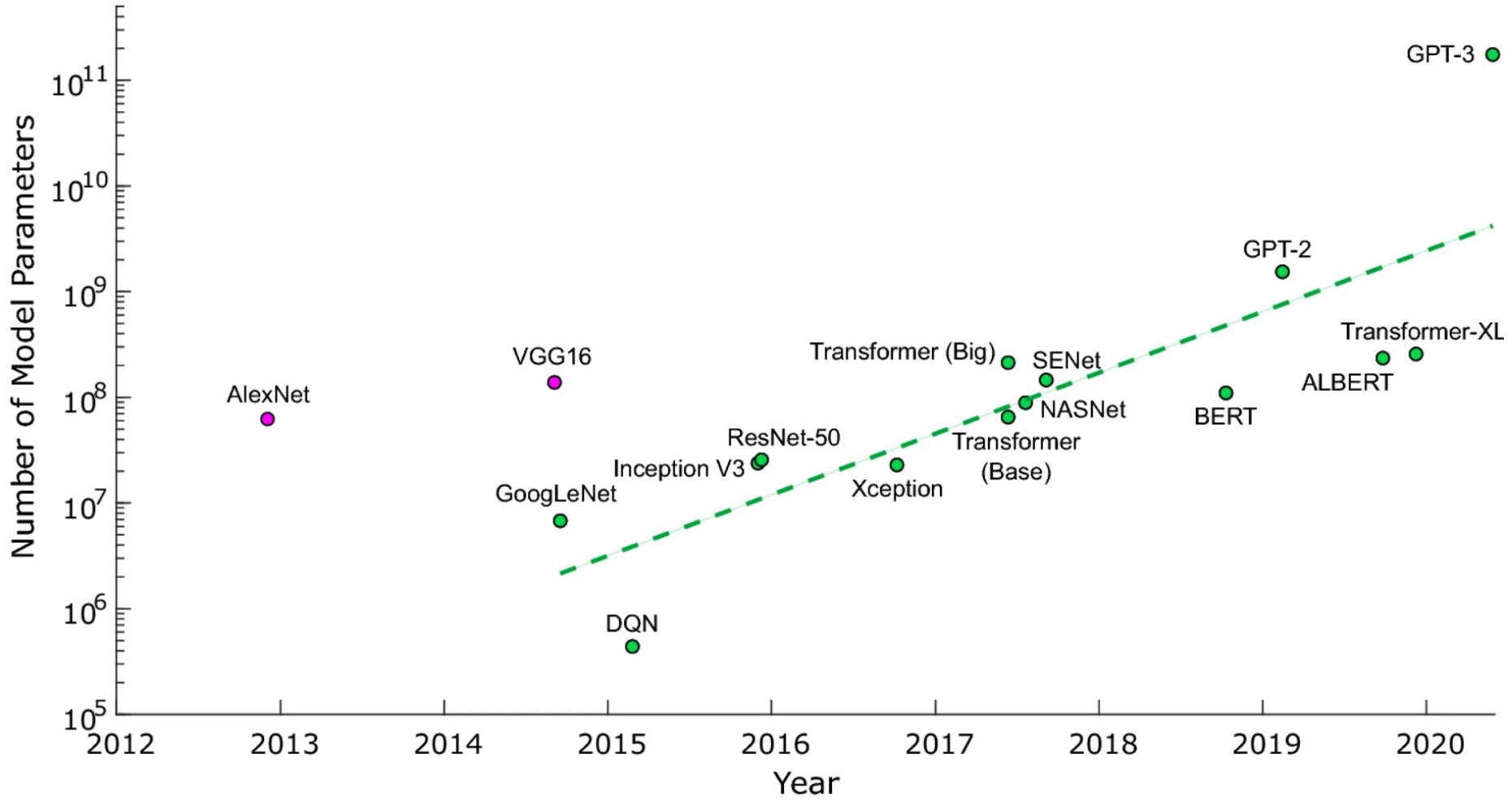
- Introduction to RISC-V
- RISC-V design with custom extensions
- **Parallel ultra-low-power (PULP) architectures**
- Enabling CNNs on PULP platforms

ETH zürich



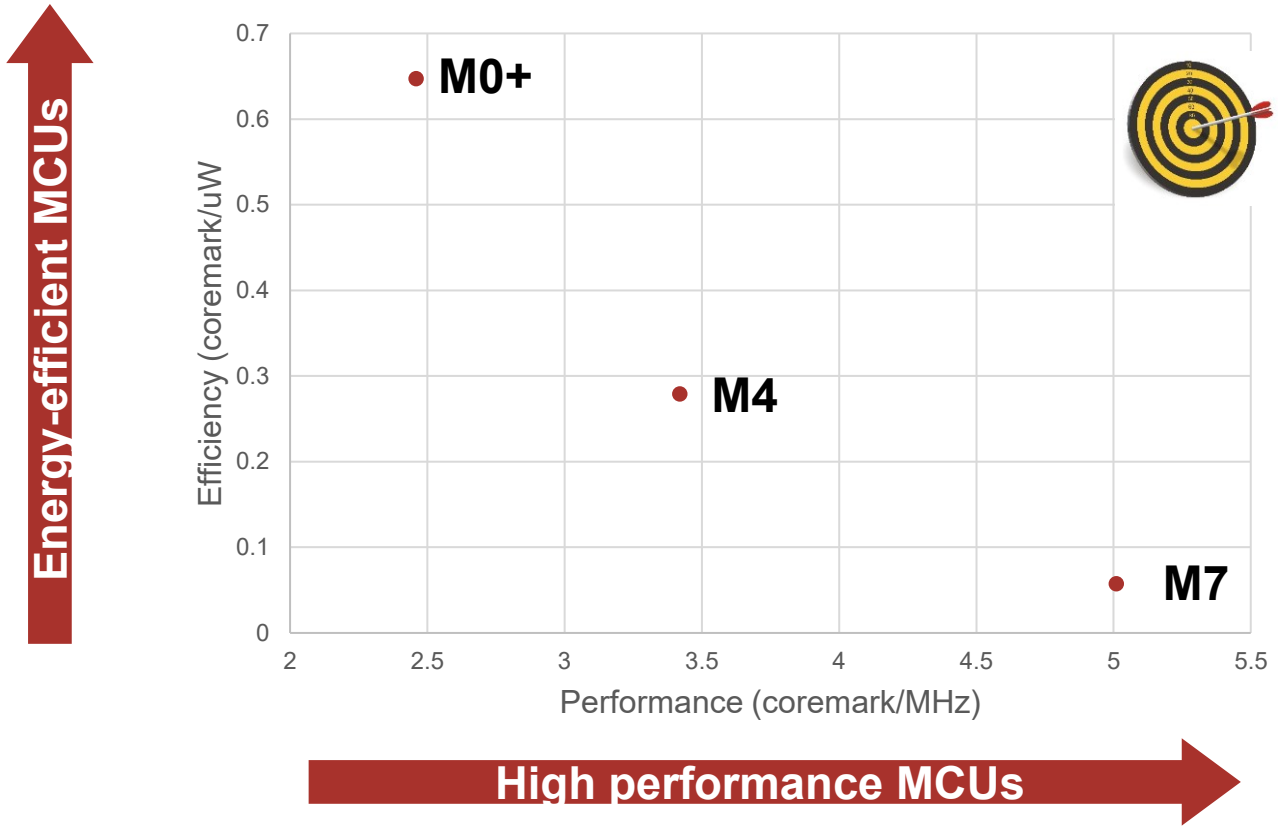
# Workloads are Growing

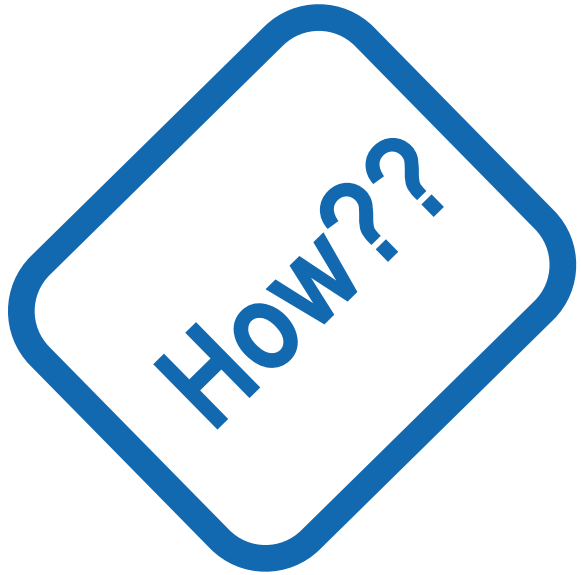
- Exponential growth in time (e.g. ML)



# Energy efficiency is the Challenge

ARM Cortex-M MCUs: M0+, M4, M7 (40LP, typ, 1.1V)*



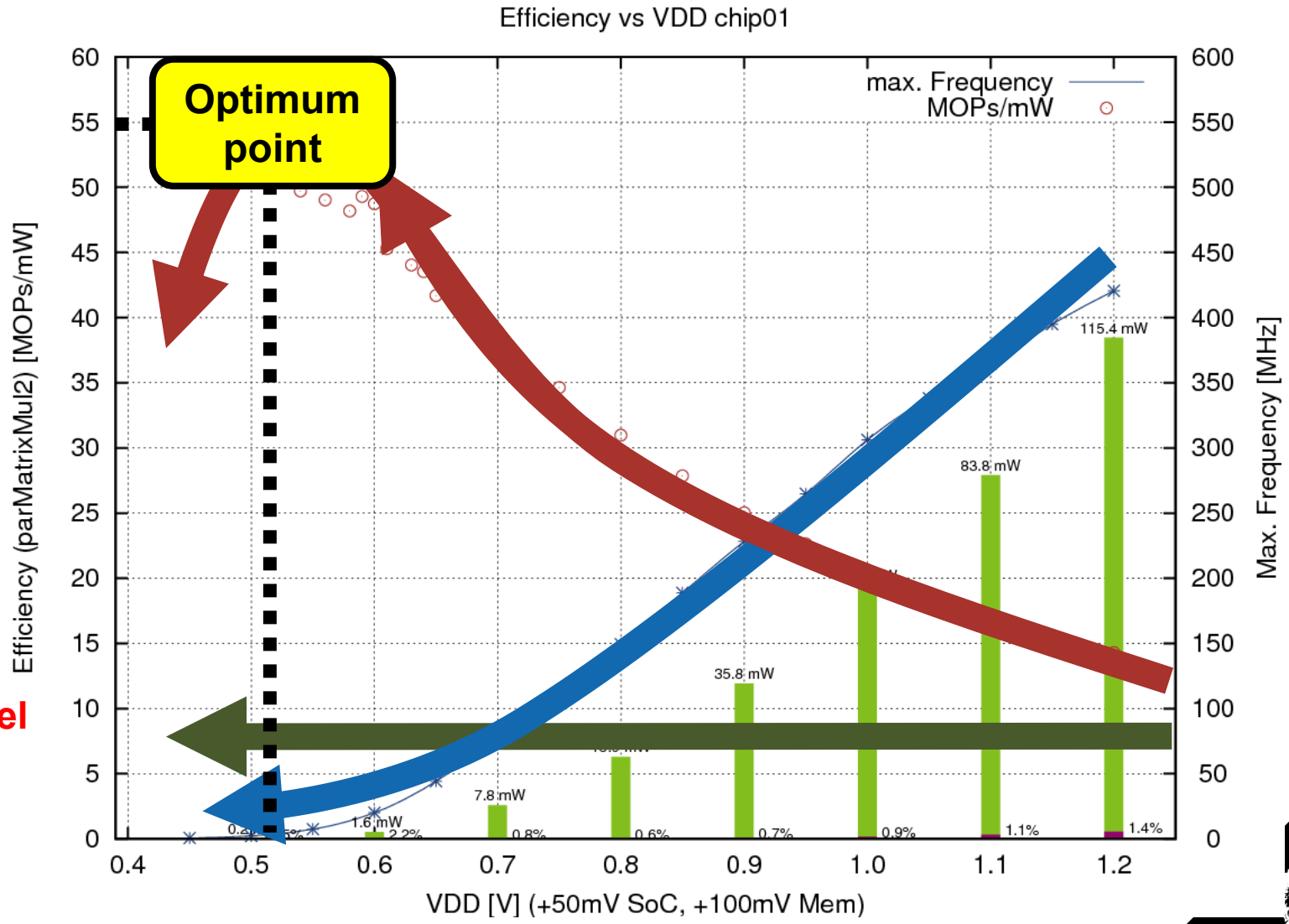




*data from ARMs web

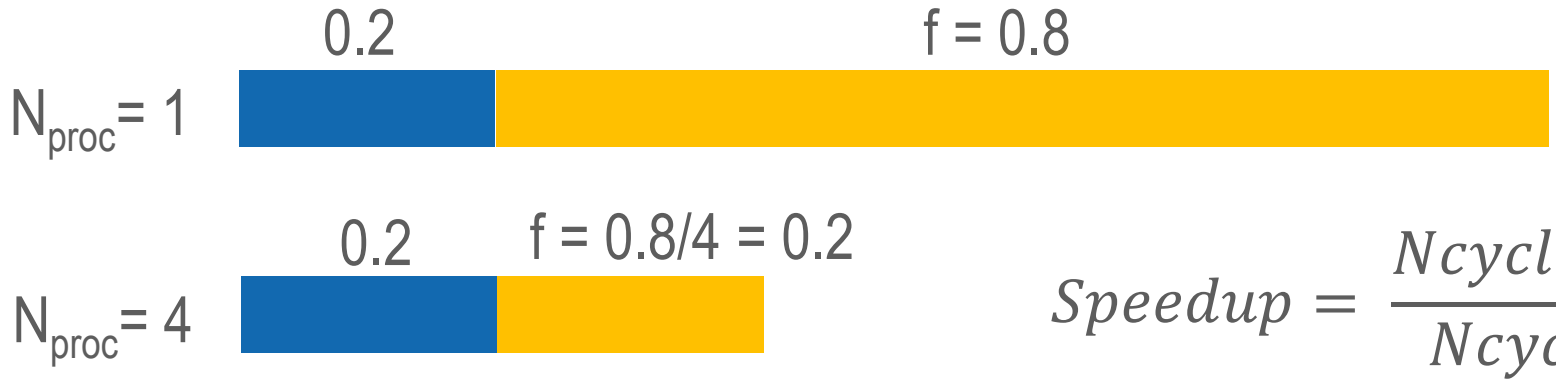
# ML & Parallel, Near-threshold: a Marriage Made in Heaven

- As VDD decreases, operating speed decreases
- However efficiency increases→ more work done per Joule
- Until leakage effects start to dominate
- **Put more units in parallel to get performance up and keep them busy with a parallel workload**

**ML is massively parallel and scales well (P/S ↑ with NN size)**



# Amdahl's Law




$$Speedup = \frac{NcyclesSequential}{NcyclesParallel}$$

f – fraction that can run in parallel

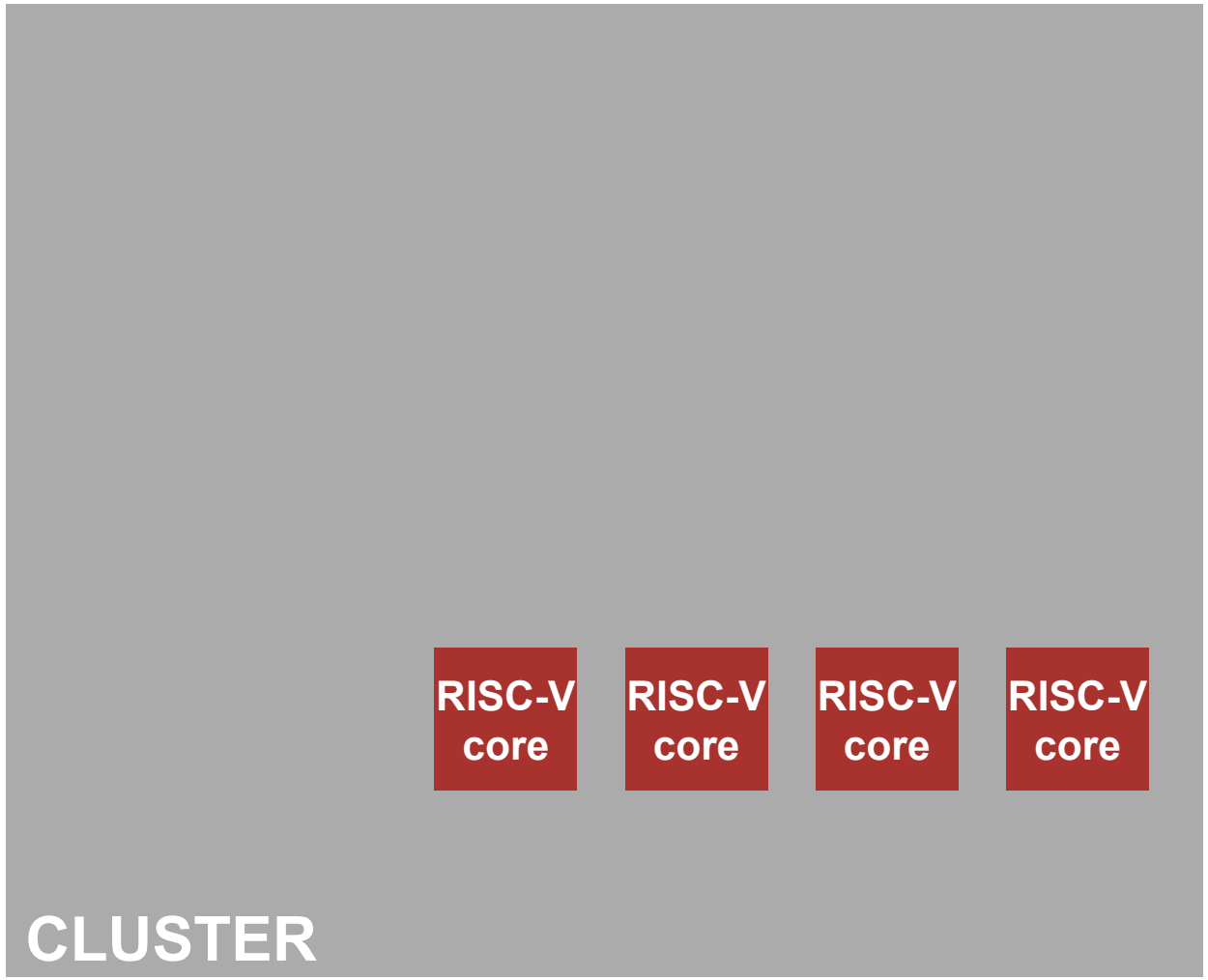
1-f – fraction that must run serially

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

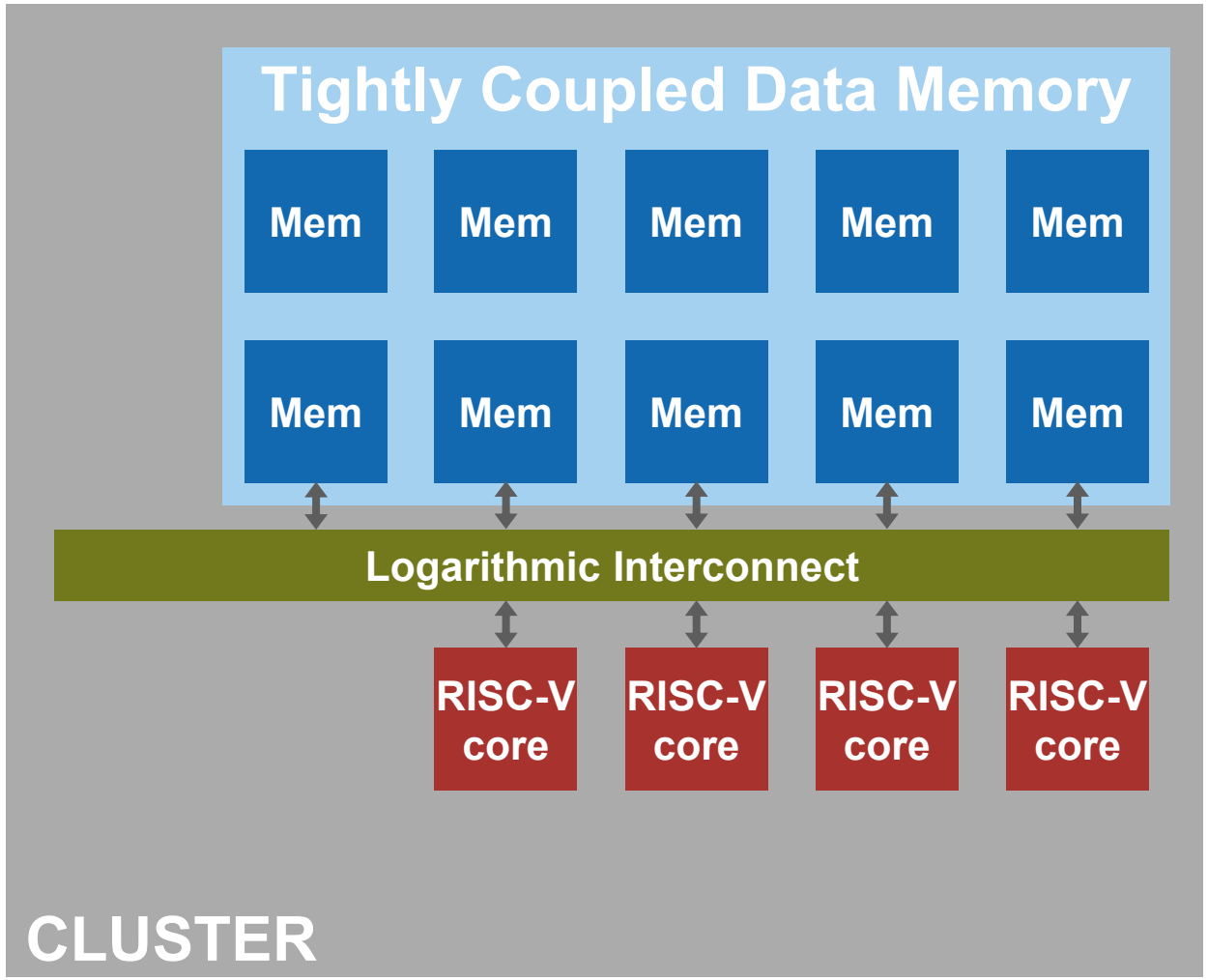
$$\lim_{n\to\infty} \frac{1}{1-f+\frac{f}{n}} = \frac{1}{1-f}$$

**For modern workloads (e.g. ML) f→1 with problem size (GOOD!!)**

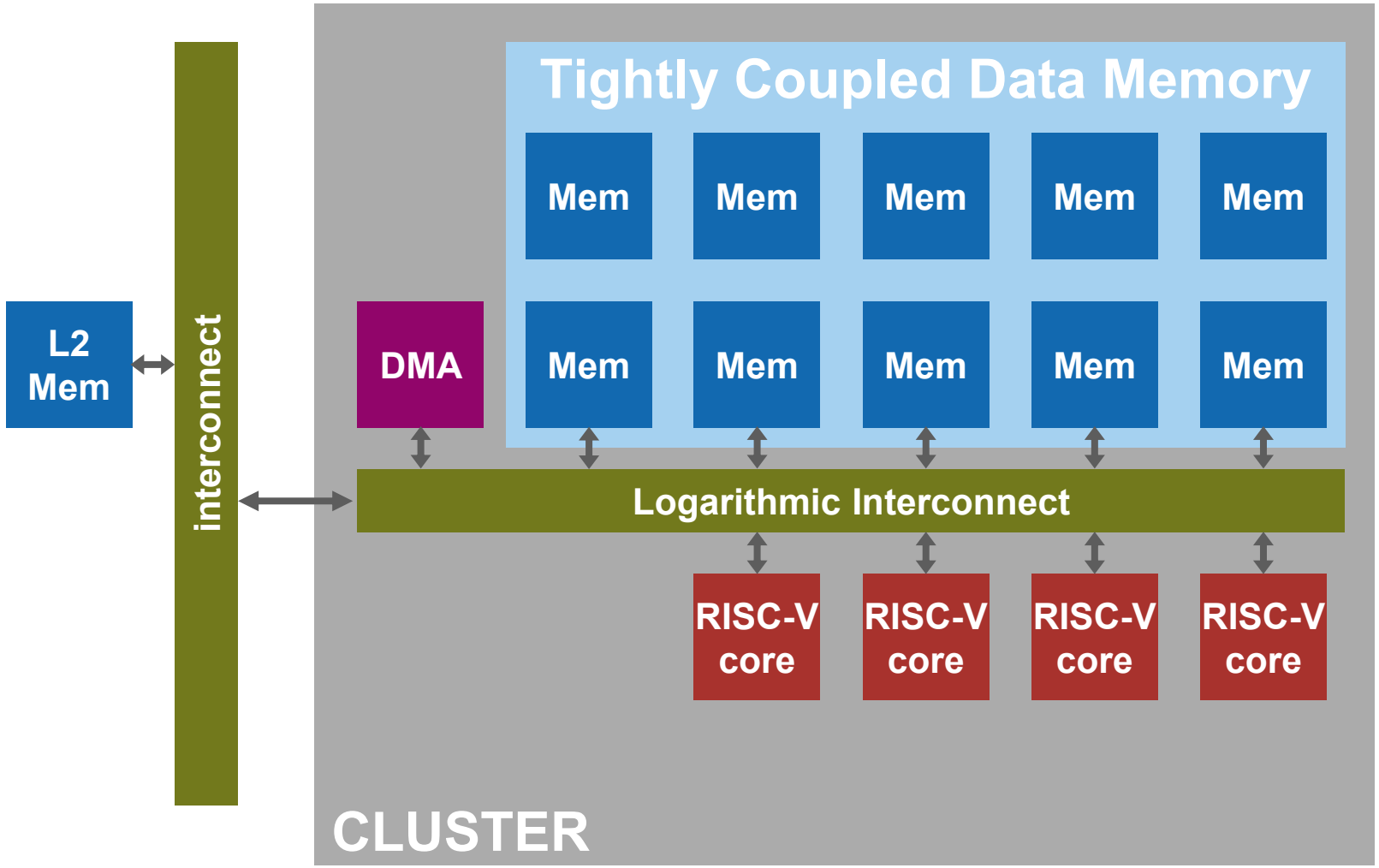
# Multiple RI5CY Cores (1-16)



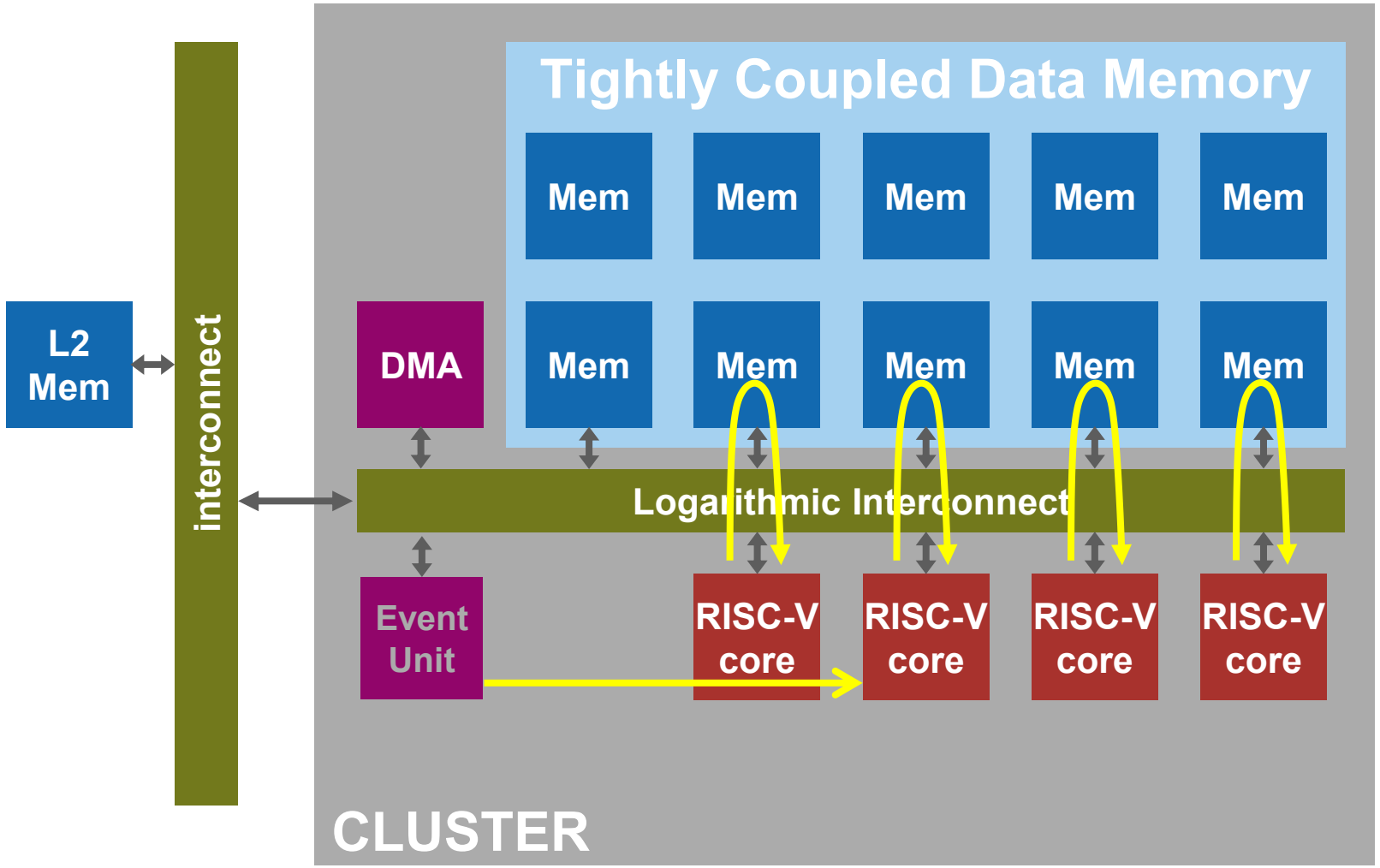
# Low-Latency Shared TCDM (L1 memory, NOT a cache)



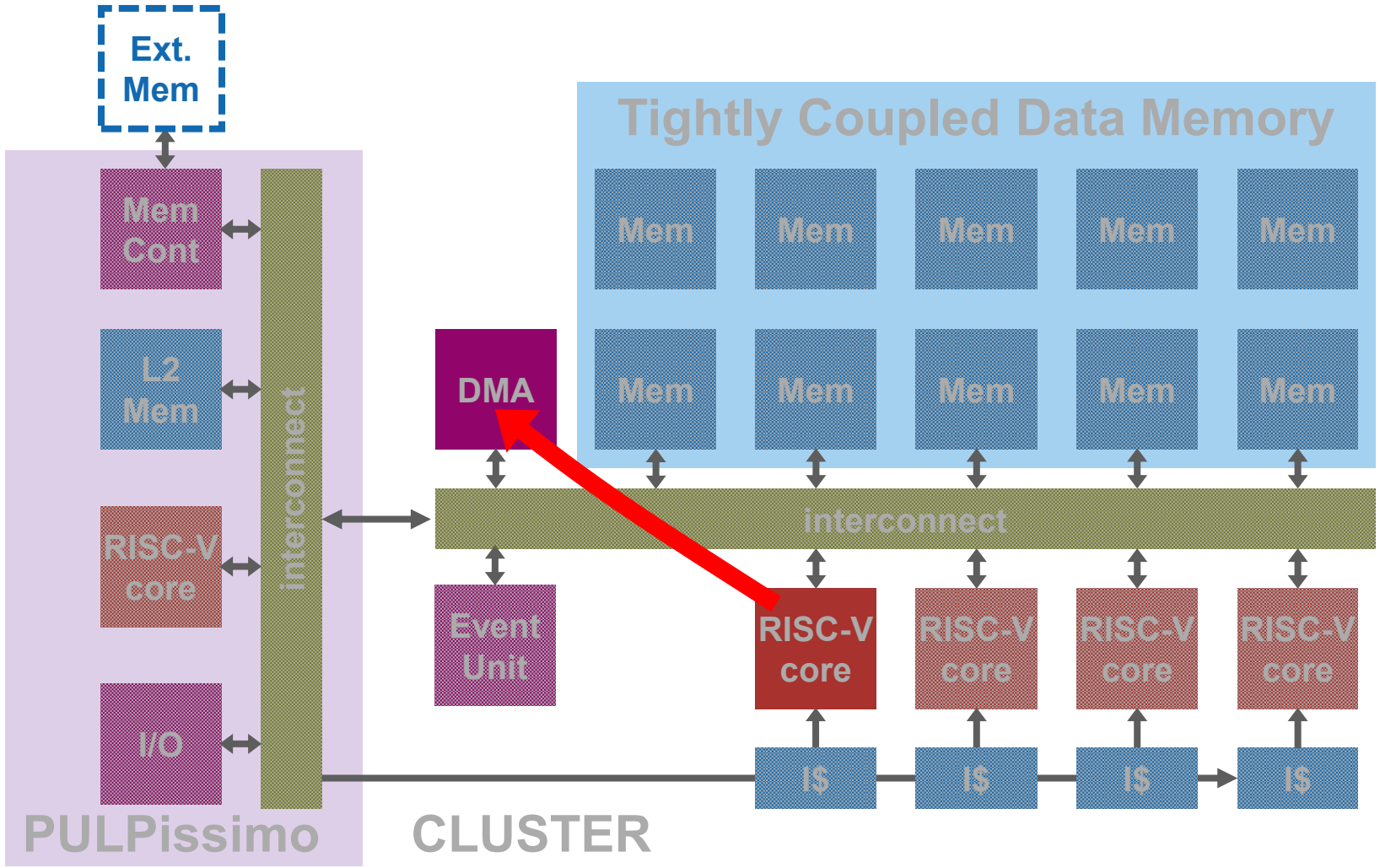
# DMA for data transfers from/to L2



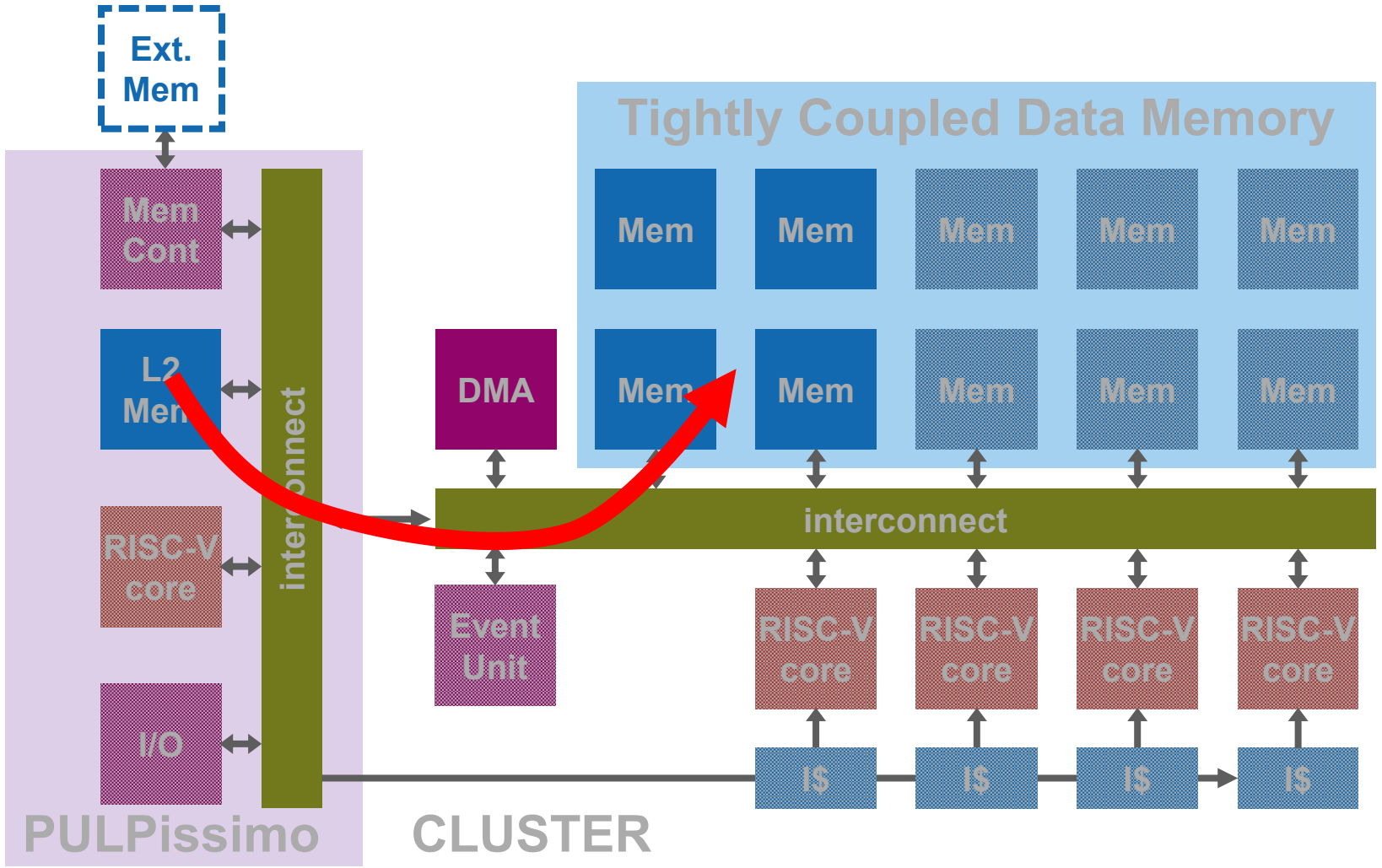
# Fast synchronization and atomics



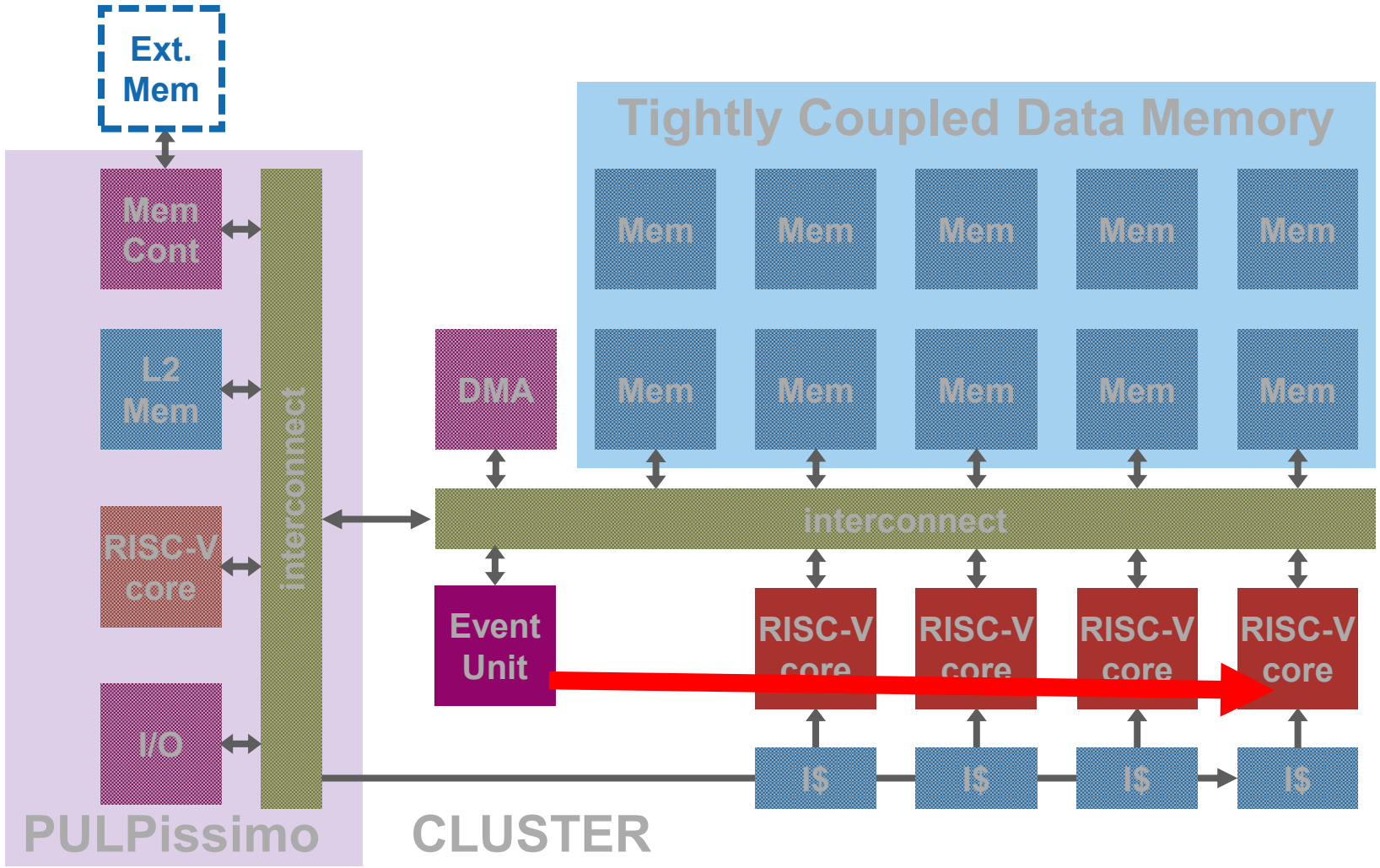
# How do we work: Initiate a DMA transfer



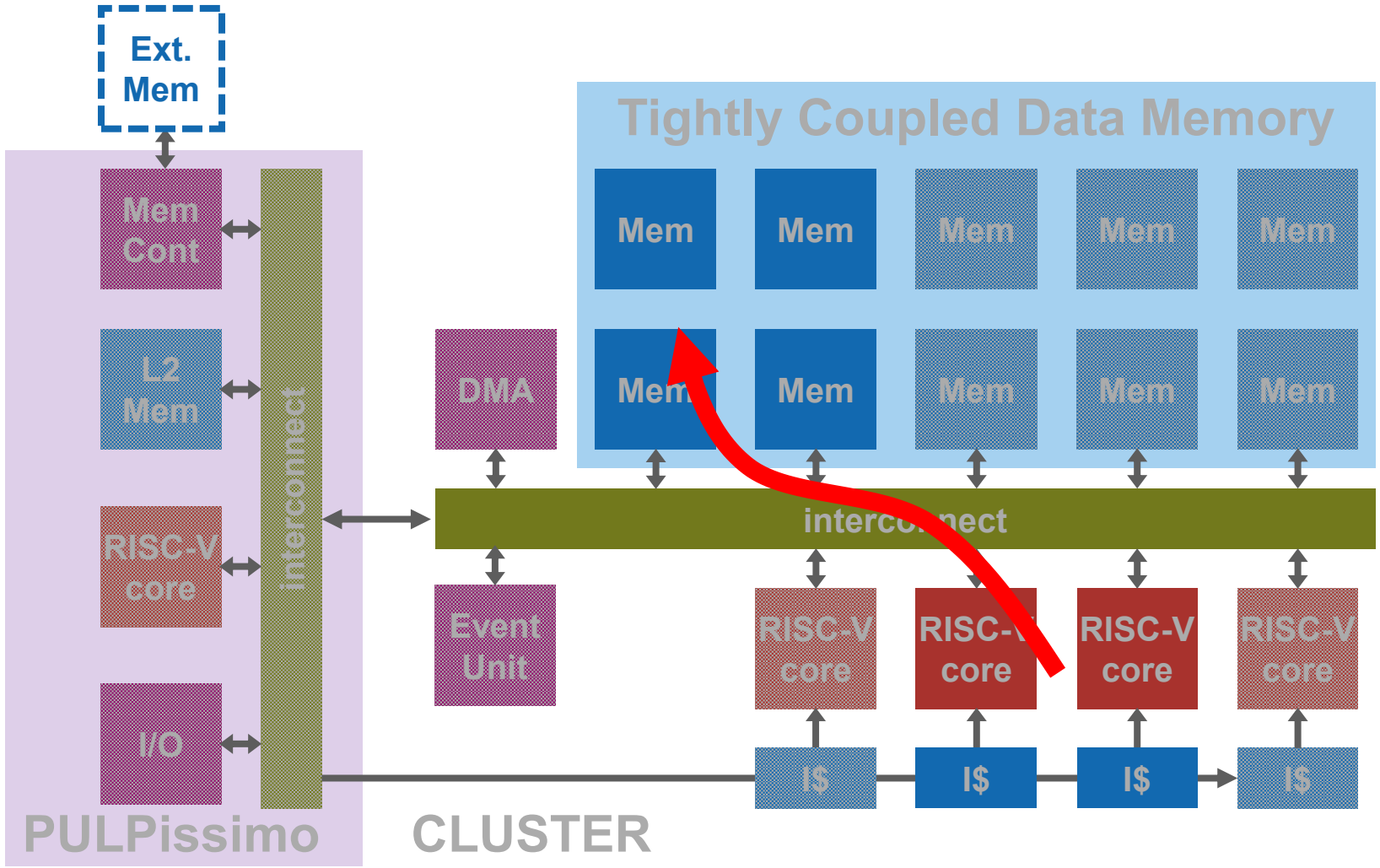
# Data copied from L2 into TCDM



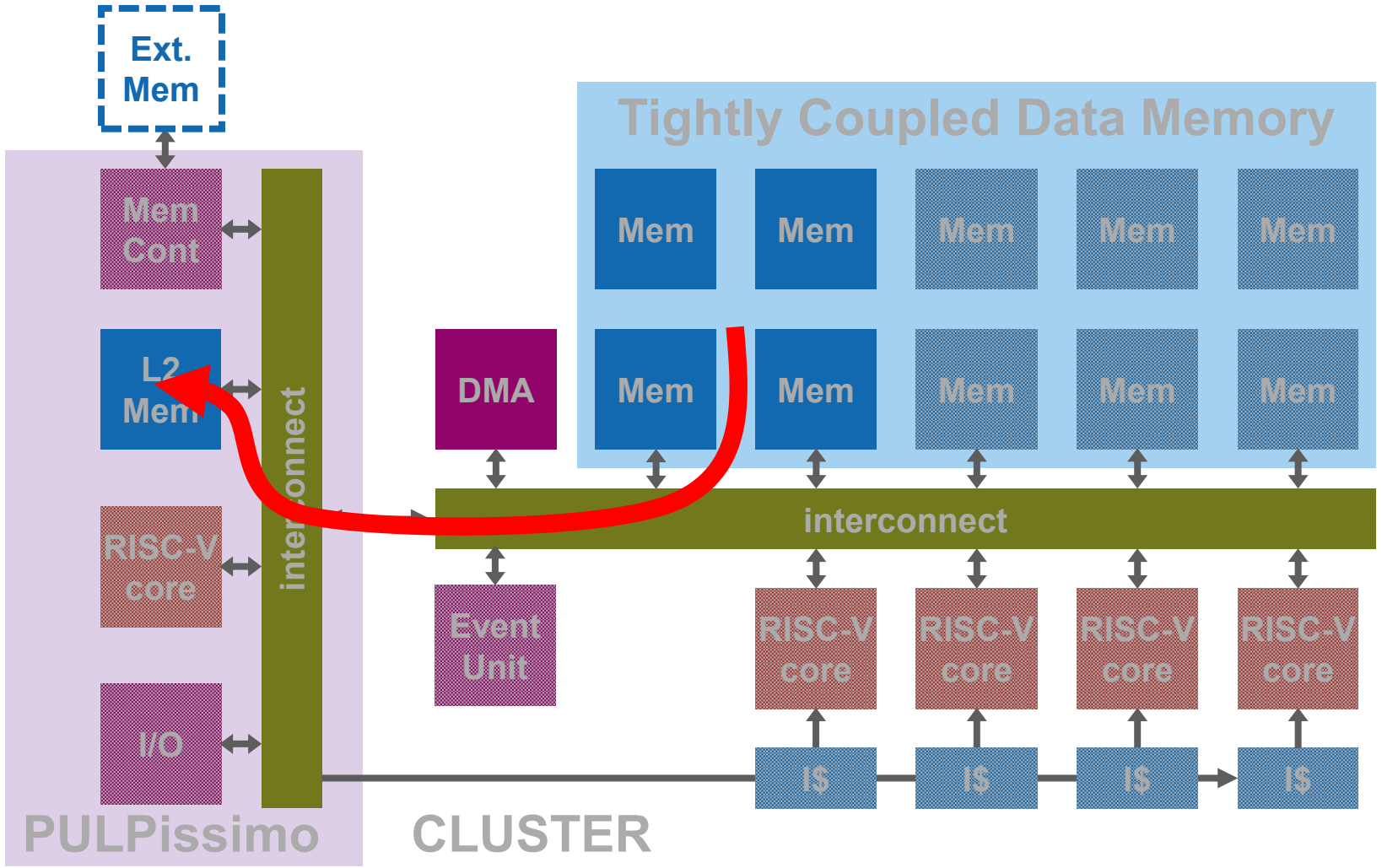
# Once data is transferred, event unit notifies cores



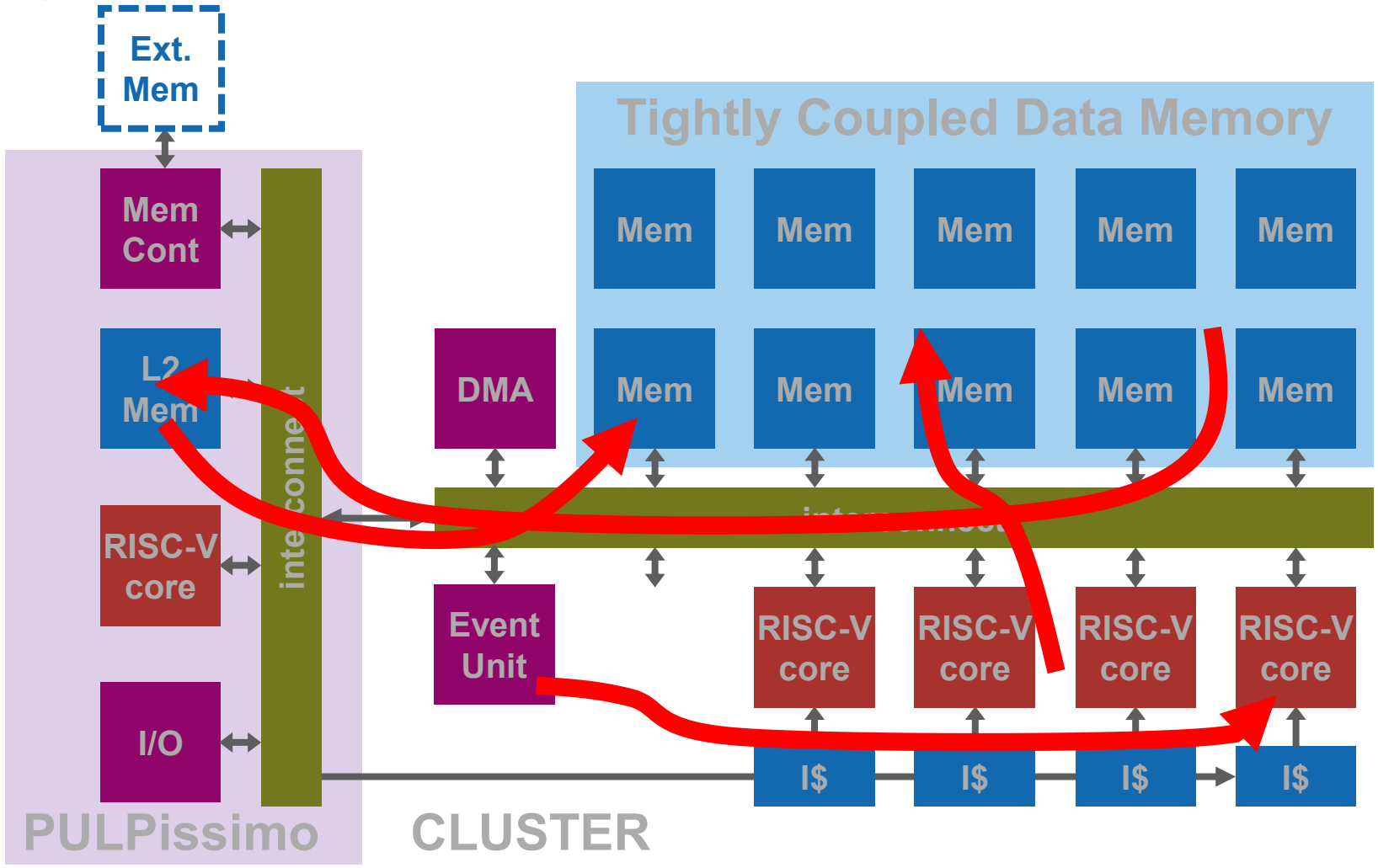
# Cores can work on the data transferred



# Once our work is done, DMA copies data back

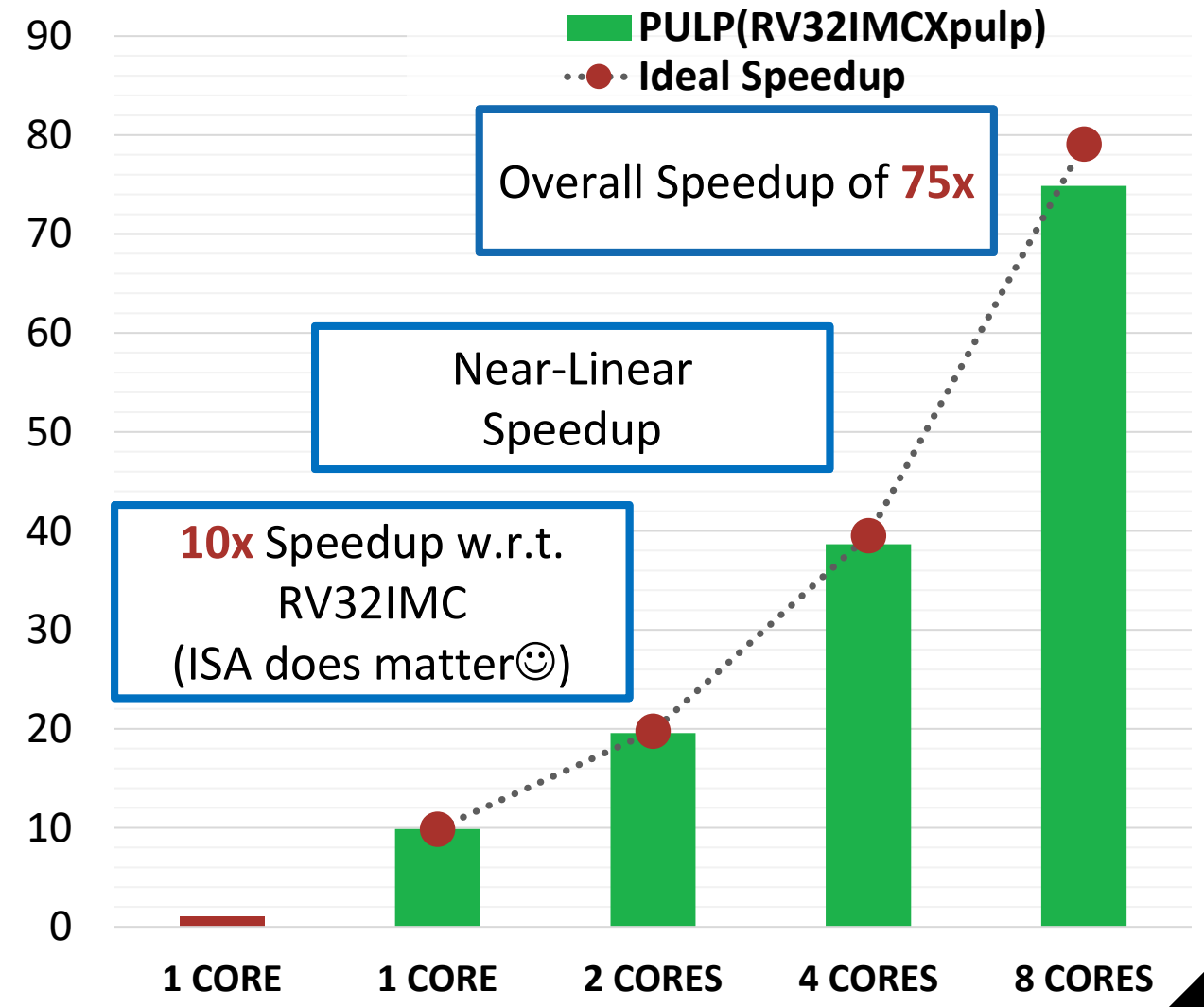


# During normal operation all of these occur concurrently



# Results: RV32IMCXpulp vs RV32IMC

- **8-bit convolution**
  - Open source DNN library
- **10x through xPULP**
  - Extensions bring real speedup
- **Near-linear speedup**
  - Scales well for regular workloads
- **75x overall gain**



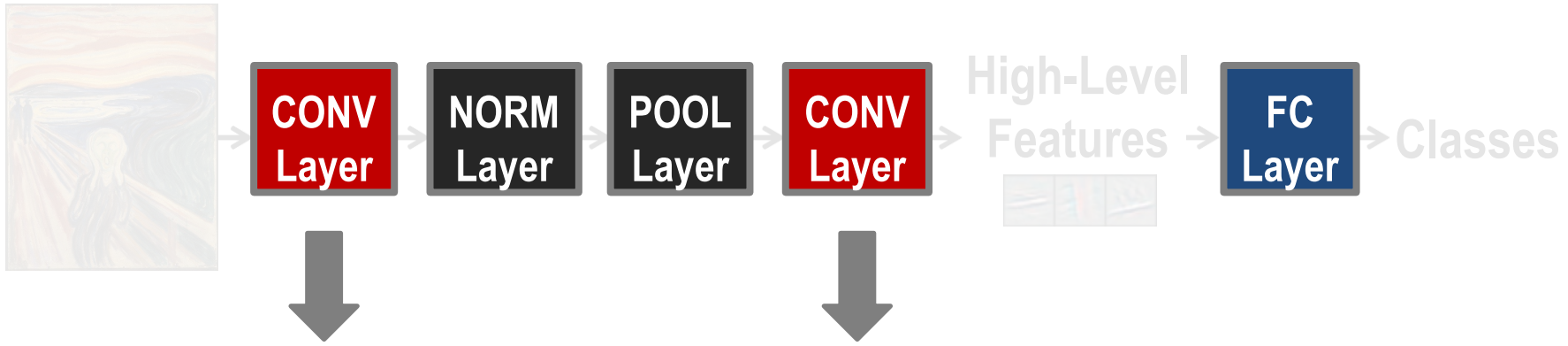

PULP-related chips 2013 - 2021
37+ SoCs
WOLF
ROSETTA
FULMINE
XAVIER
ATOMARIO
BAIKONUR
БАЙКОНУР
ETH zürich
2001: A SYSTEM ON CHIP
PLINK

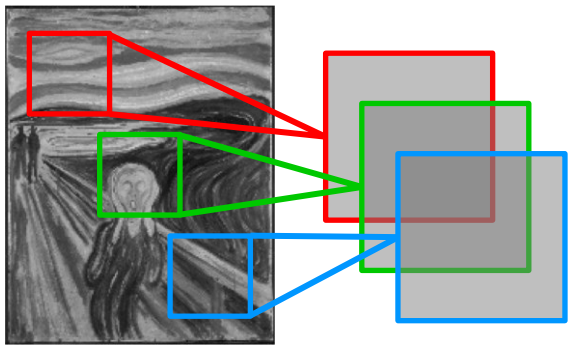
# Agenda

- Introduction to RISC-V
- RISC-V design with custom extensions
- Parallel ultra-low-power (PULP) architectures
- **Enabling CNNs on PULP platforms**



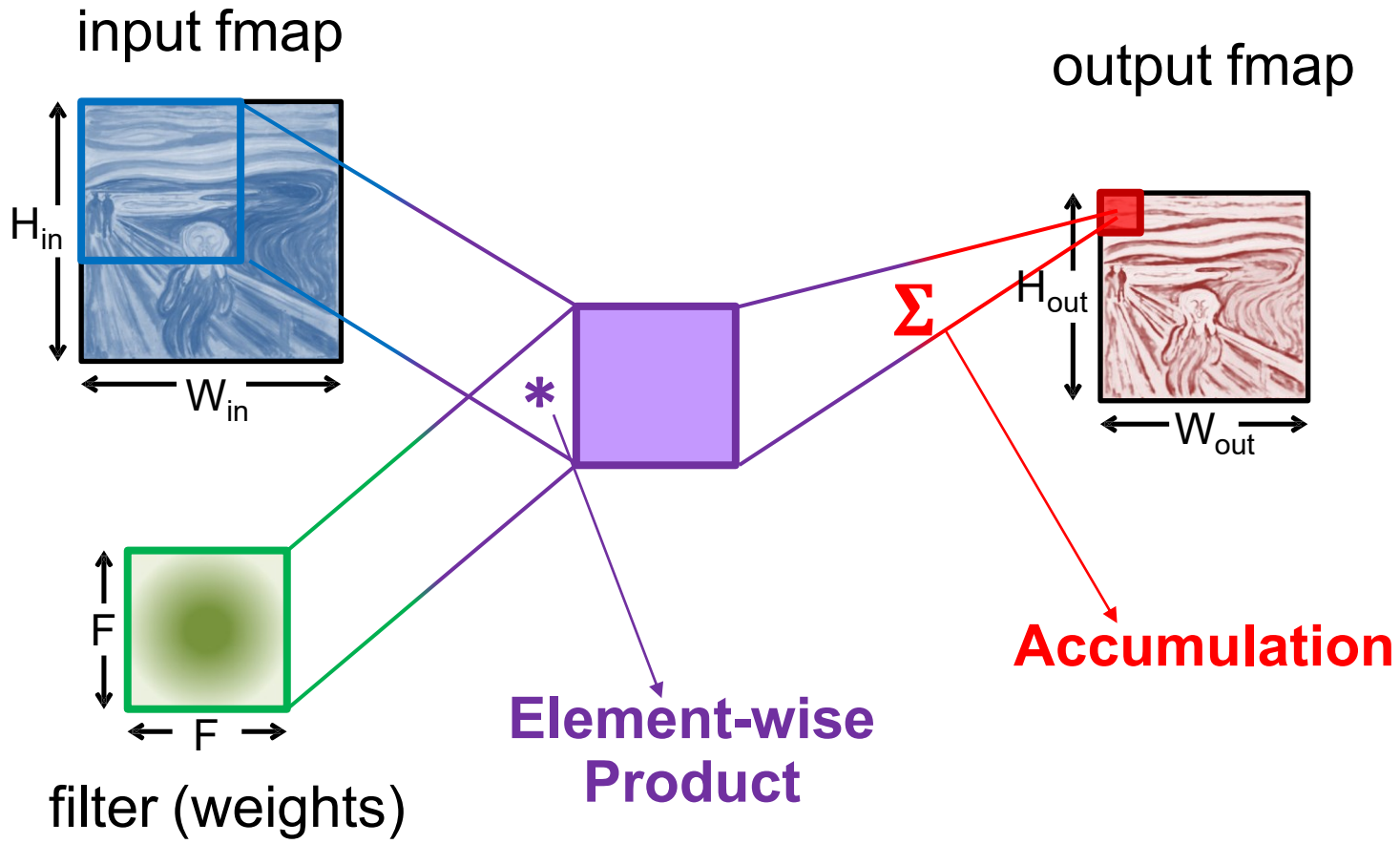
# Deep Convolutional Neural Networks



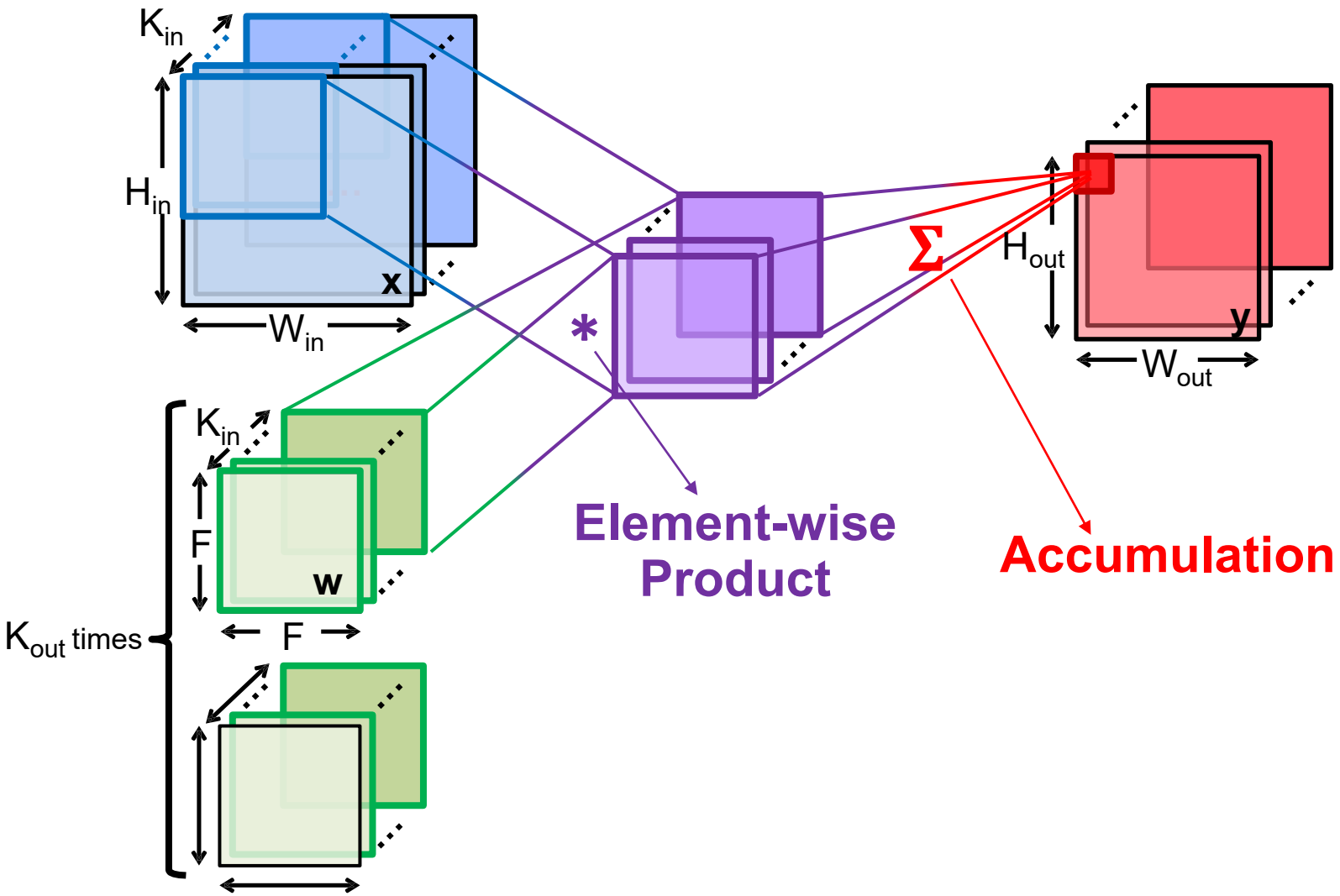



**Convolutions** account for more than 90% of overall computation, dominating **runtime** and **energy consumption** – and are also the "model" computation of DNNs

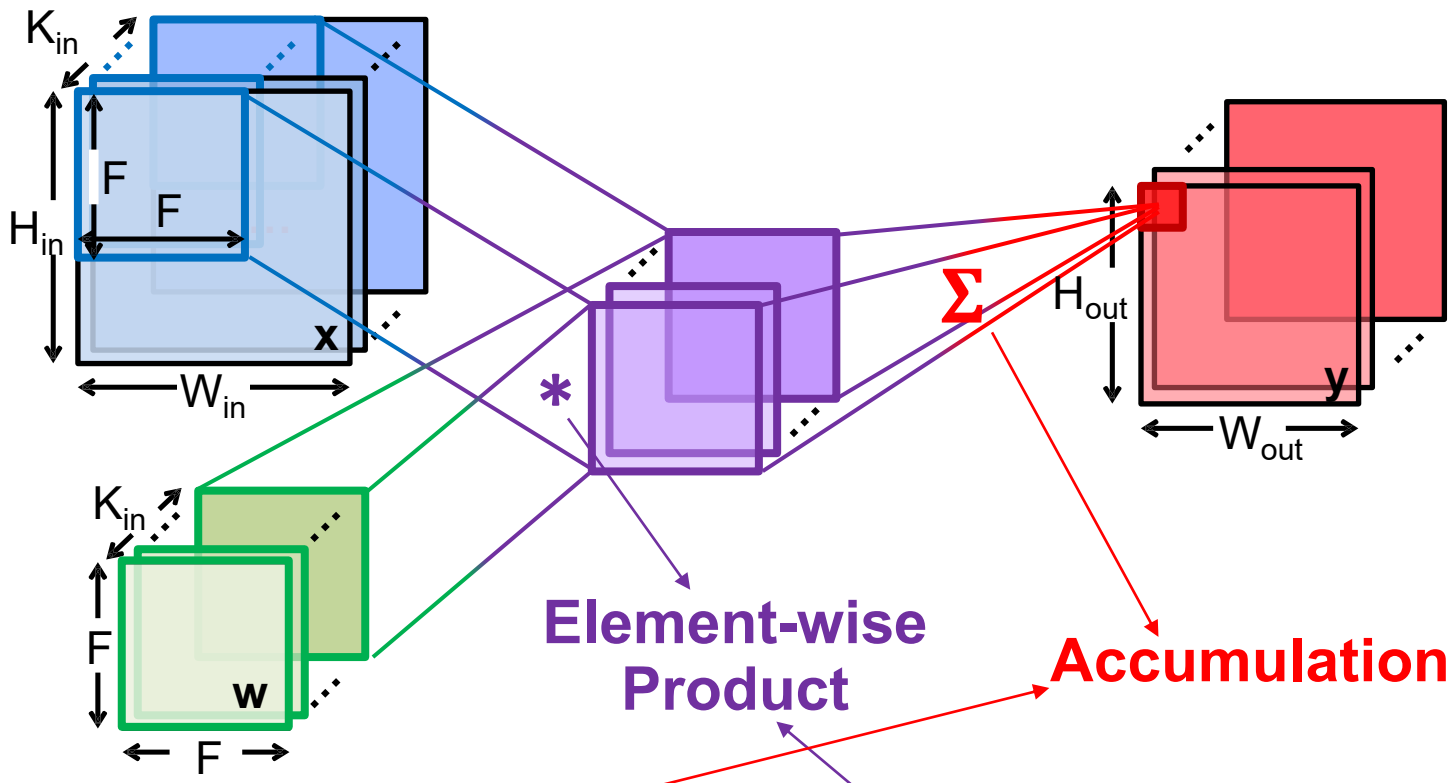
# Convolution (CONV) Layer



# Convolution (CONV) Layer

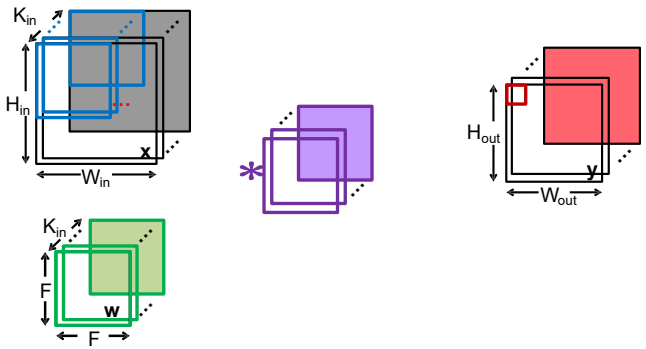


# Convolution (CONV) Layer




$$y[m,i,j] = b[m] + \Sigma(w[m,0:K_{in},0:F,0:F] * x[0:K_{in},i:i+F,j:j+F])$$

# From Specification to Dataflow




The specification of a certain operator (e.g., Conv layer) does not have any specific ordering for the operations to be performed:

$$y[m,i,j] = \mathit{CONV}(w, x[0{:}K_{in}, i{:}i+u, j{:}j+v])$$

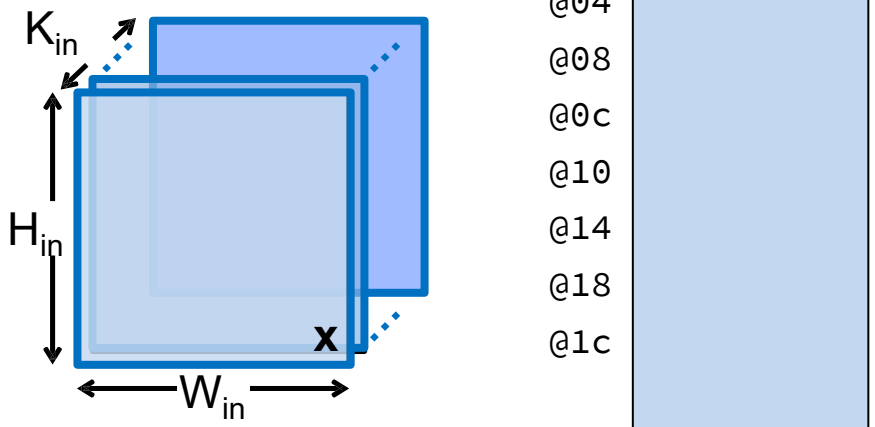
A ***dataflow*** is an *ordering choice*. It is specified by means of a ***loop nest***: for a Conv layer, 6 nested for-loops

```
for m in range(0, K_out):          # output channels
  for n in range(0, K_in):         # input channels
    for i in range(0, H_out):      # spatial (height)
      for j in range(0, W_out):    # spatial (width)
        psum = b[m]
        for ui in range(0, F):     # filter (height)
          for uj in range(0, F):   # filter (width)
            psum += w[m,n,ui,uj] * x[n,i+ui,j+uj]
        y[m,i,j] = psum            # often with act()
```

# From Tensor to Memory Layout

Memory is **flat**: N-dimensional data (i.e., tensors) have to be *flattened* into a certain data layout

- many possible ways to lay out data in the flat memory



# PULP-NN: a library of primitives for NN on PULP

- **Full reference: https://arxiv.org/abs/1908.11263**

- **Work on L1 memory** → data exchange with L2 is managed at higher level (DORY, discussed later)
- **Exploit parallelism + vectorization** capabilities of RI5CY cores
- Transform all **linear operators into a GEMM (Generalized Matrix Multiplication)** form
- Target data layout: Height/Width/Channel (**HWC**)

# 2D conv as a GEMV

*Input* **x**

| | | | | | |
|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

| | | |
|---|---|---|
| 00 | 01 | 02 |
| 10 | 11 | 12 |
| 20 | 21 | 22 |

*Weight* **w**

***flatten*** *to 1D vector*

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

*Linearized* **w**

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

*Input* **x**

| | | | | | |
|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

***Im2Col*** *transformation*

*Im2col* **x**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |

*Linearized* **w**

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

*Input* **x**

| | | | | | |
|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

***Im2Col*** *transformation*

*Im2col* **x**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |

*Linearized* **w**

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

*Input* **x**

| | | | | | |
|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

***Im2Col*** *transformation*

*Im2col* **x**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |

*Linearized* **w**

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

***Im2Col** transformation*

*Input* **x**

| | | | | | |
|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

*Im2col* **x**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

*Linearized* **w**

A lot of data duplication!

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

*Im2col* **x**

•

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

*Linearized* **w**

=

| |
|---|
| 00 |
| 01 |
| 02 |
| 03 |
| 10 |
| 11 |
| 12 |
| 13 |
| 20 |
| 21 |
| 22 |
| 23 |
| 30 |
| 31 |
| 32 |
| 33 |

*Linearized* **y**

| | | | |
|---|---|---|---|
| 00 | 01 | 02 | 03 |
| 10 | 11 | 12 | 13 |
| 20 | 21 | 22 | 23 |
| 30 | 31 | 32 | 33 |

*Output* **y**

```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# 2D conv as a GEMV

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Core 0 | 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| Core 0 | 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| Core 1 | 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| Core 1 | 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| Core 2 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| Core 2 | 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| Core 3 | 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| Core 3 | 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| Core 4 | 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| Core 4 | 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| Core 5 | 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| Core 5 | 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| Core 6 | 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| Core 6 | 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| Core 7 | 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| Core 7 | 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

*Im2col* **x**

•

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

*Linearized* **w**

=

| |
|---|
| 00 |
| 01 |
| 02 |
| 03 |
| 10 |
| 11 |
| 12 |
| 13 |
| 20 |
| 21 |
| 22 |
| 23 |
| 30 |
| 31 |
| 32 |
| 33 |

*Linearized* **y**

| | | | |
|---|---|---|---|
| 00 | 01 | 02 | 03 |
| 10 | 11 | 12 | 13 |
| 20 | 21 | 22 | 23 |
| 30 | 31 | 32 | 33 |

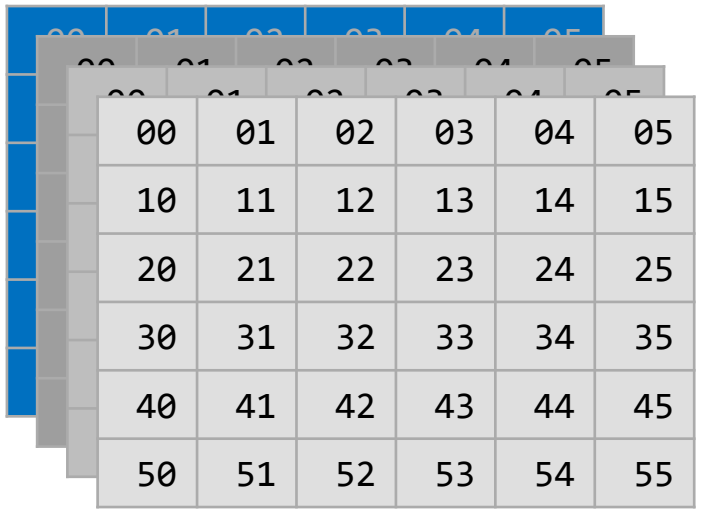
*Output* **y**

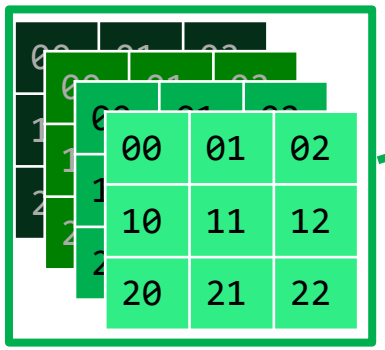
```
y[i,j] = Σ(w[0:F,0:F] * x[i:i+F,j:j+F])
```

# CONV Layer as a GEMM

Considering $K_{out}$=1 for simplicity

*Input* **x**




*Weight* **w**

***flatten** to 1D vector*

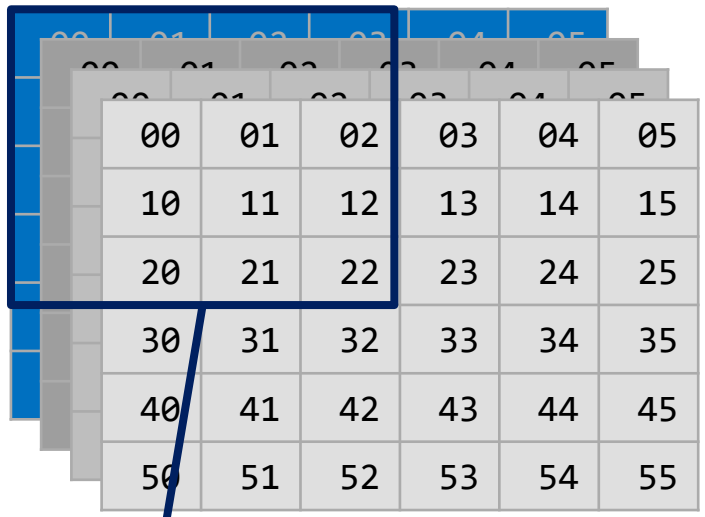
*Linearized* **w**


$$y[m,i,j] = \Sigma(w[m,0{:}K_{in},0{:}F,0{:}F] * x[0{:}K_{in},i{:}i{+}F,j{:}j{+}F])$$

# CONV Layer as a GEMM

Considering $K_{out}$=1 for simplicity

*Input* **x**




***Im2Col** transformation*

| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 | … | 21 | 22 | 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

*Im2col* **x**

*Linearized* **w**

| 00 |
|---|
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |
| … |
| 21 |
| 22 |
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

$$y[m,i,j] = \Sigma(w[m,0{:}K_{in},0{:}F,0{:}F] * x[0{:}K_{in},i{:}i{+}F,j{:}j{+}F])$$





# CONV Layer as a GEMM

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

…

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 22 | 23 | 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 23 | 24 | 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 24 | 25 | 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 31 | 32 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 32 | 33 | 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 33 | 34 | 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 34 | 35 | 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 41 | 42 | 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 42 | 43 | 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 43 | 44 | 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 44 | 45 | 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 51 | 52 | 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 52 | 53 | 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 53 | 54 | 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 54 | 55 | 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

*Linearized* **w**

| |
|---|
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |
| … |
| 21 |
| 22 |
| 00 |
| 01 |
| 02 |
| 10 |
| 11 |
| 12 |
| 20 |
| 21 |
| 22 |

*Im2col* **x**

Considering $K_{out}$=1 for simplicity

$$y[m,i,j] = \Sigma(w[m,0:K_{in},0:F,0:F] * x[0:K_{in},i:i+F,j:j+F])$$

# CONV Layer as a GEMM

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

…

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 |
| 22 | 23 | 01 | 02 | 03 | 11 | 12 | 13 | 21 | 22 | 23 |
| 23 | 24 | 02 | 03 | 04 | 12 | 13 | 14 | 22 | 23 | 24 |
| 24 | 25 | 03 | 04 | 05 | 13 | 14 | 15 | 23 | 24 | 25 |
| 31 | 32 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
| 32 | 33 | 11 | 12 | 13 | 21 | 22 | 23 | 31 | 32 | 33 |
| 33 | 34 | 12 | 13 | 14 | 22 | 23 | 24 | 32 | 33 | 34 |
| 34 | 35 | 13 | 14 | 15 | 23 | 24 | 25 | 33 | 34 | 35 |
| 41 | 42 | 20 | 21 | 22 | 30 | 31 | 32 | 40 | 41 | 42 |
| 42 | 43 | 21 | 22 | 23 | 31 | 32 | 33 | 41 | 42 | 43 |
| 43 | 44 | 22 | 23 | 24 | 32 | 33 | 34 | 42 | 43 | 44 |
| 44 | 45 | 23 | 24 | 25 | 33 | 34 | 35 | 43 | 44 | 45 |
| 51 | 52 | 30 | 31 | 32 | 40 | 41 | 42 | 50 | 51 | 52 |
| 52 | 53 | 31 | 32 | 33 | 41 | 42 | 43 | 51 | 52 | 53 |
| 53 | 54 | 32 | 33 | 34 | 42 | 43 | 44 | 52 | 53 | 54 |
| 54 | 55 | 33 | 34 | 35 | 43 | 44 | 45 | 53 | 54 | 55 |

*Im2col* **x**

With $K_{out}>1$

| | | | | |
|---|---|---|---|---|
| 00 | 00 | 00 | | 00 |
| 01 | 01 | 01 | | 01 |
| 02 | 02 | 02 | | 02 |
| 10 | 10 | 10 | | 10 |
| 11 | 11 | 11 | | 11 |
| 12 | 12 | 12 | | 12 |
| 20 | 20 | 20 | | 20 |
| 21 | 21 | 21 | | 21 |
| 22 | 22 | 22 | | 22 |

…

| | | | | |
|---|---|---|---|---|
| 21 | 21 | 21 | | 21 |
| 22 | 22 | 22 | | 22 |
| 00 | 00 | 00 | | 00 |
| 01 | 01 | 01 | | 01 |
| 02 | 02 | 02 | | 02 |
| 10 | 10 | 10 | | 10 |
| 11 | 11 | 11 | | 11 |
| 12 | 12 | 12 | | 12 |
| 20 | 20 | 20 | | 20 |
| 21 | 21 | 21 | | 21 |
| 22 | 22 | 22 | | 22 |

*Matrix* **w**

$$y[m,i,j] = \Sigma(w[m,0:K_{in},0:F,0:F] * x[0:K_{in},i:i+F,j:j+F])$$

# PULP-NN: optimized computational back-end

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

```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      psum = 0                                          # ignore bias
      for ui in range(0, F):
        for uj in range(0, F):
          for n in range(0, K_in):
            psum += w[m,ui,uj,n] * x[i+ui,j+uj,n]
      y[i,j,m] = act(psum)
```

# PULP-NN: optimized computational back-end

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

```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      psum = 0                                          # ignore bias
      for ui in range(0, F):
        for uj in range(0, F):
          for n in range(0, K_in):
            psum += w[m,ui,uj,n] * x[i+ui,j+uj,n]
      y[i,j,m] = act(psum)
```

# PULP-NN: optimized computational back-end

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

```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      for ui in range(0, F):
        for uj in range(0, F):
          for n in range(0, K_in):
            im2col[ui,uj,n] = x[i+ui,j+uj,n]
      psum = 0                                          # ignore bias
      for ui in range(0, F):
        for uj in range(0, F):
          for n in range(0, K_in):
            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, CoHWCi for weights

```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      for idx in range(0, F*F*K_in):
        for uj in range(0, F):
          for n in range(0, K_in):
            im2col[ui*F*K_in + uj*K_in + n] = x[i+ui,j+uj,n]
      psum = 0                                          # ignore bias
      for idx in range(0, F*F*K_in):
        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, CoHWCi for weights

```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      for idx in range(0, F*F*K_in):
        for uj in range(0, F):
          for n in range(0, K_in):
            im2col[ui*F*K_in + uj*K_in + n] = x[i+ui,j+uj,n]
      psum = 0                                          # ignore bias
      for idx in range(0, F*F*K_in):
        psum += w[m,idx] * im2col[idx]
      y[i,j,m] = act(psum)
```

3. Highlighted kernel is a ***Matrix Mul!***

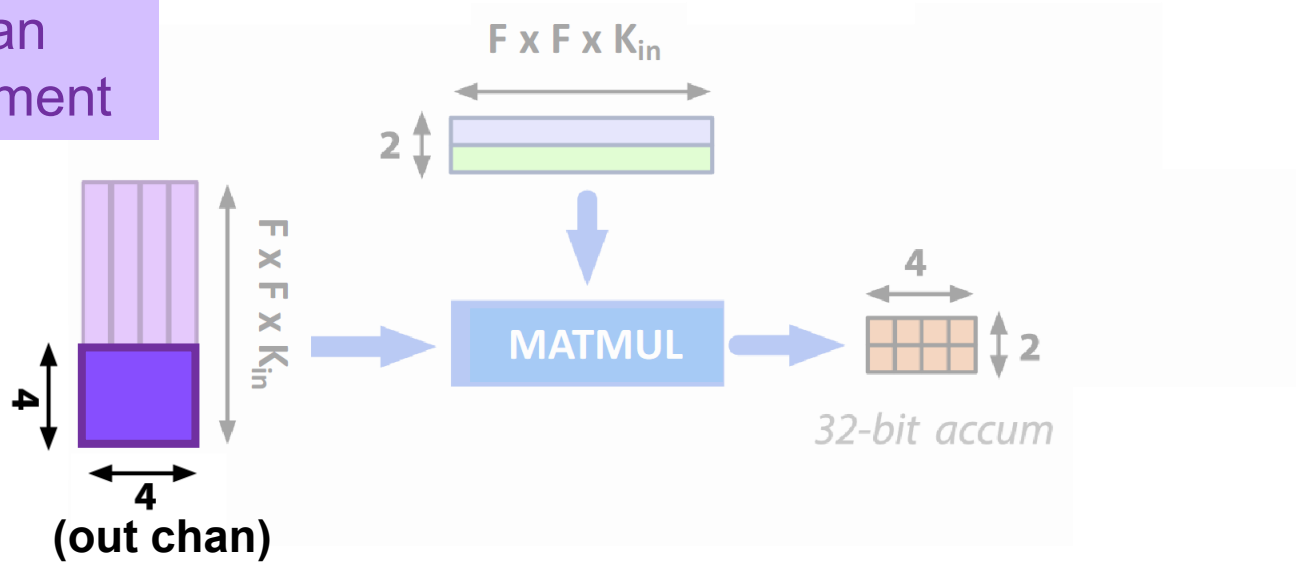
# PULP-NN: optimized computational back-end

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




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

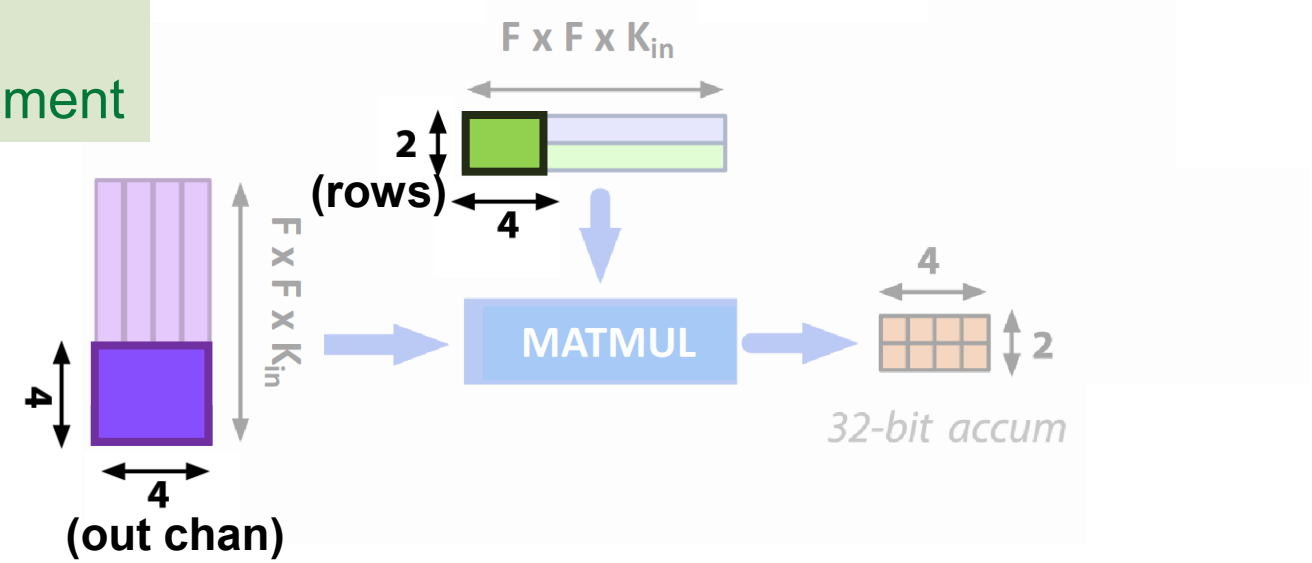
# PULP-NN: optimized computational back-end

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



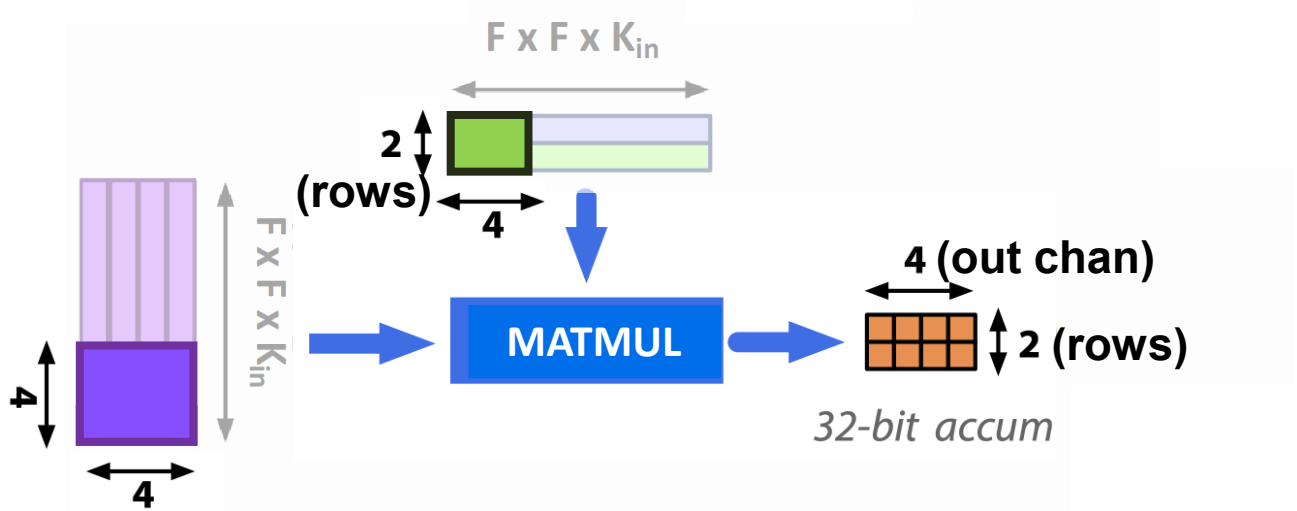

**PULP-NN** [Garofalo 19] https://arxiv.org/abs/1908.11263

# PULP-NN: optimized computational 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
```




**Compute 32 MAC over 8 accumulators**

dot-product instructions

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

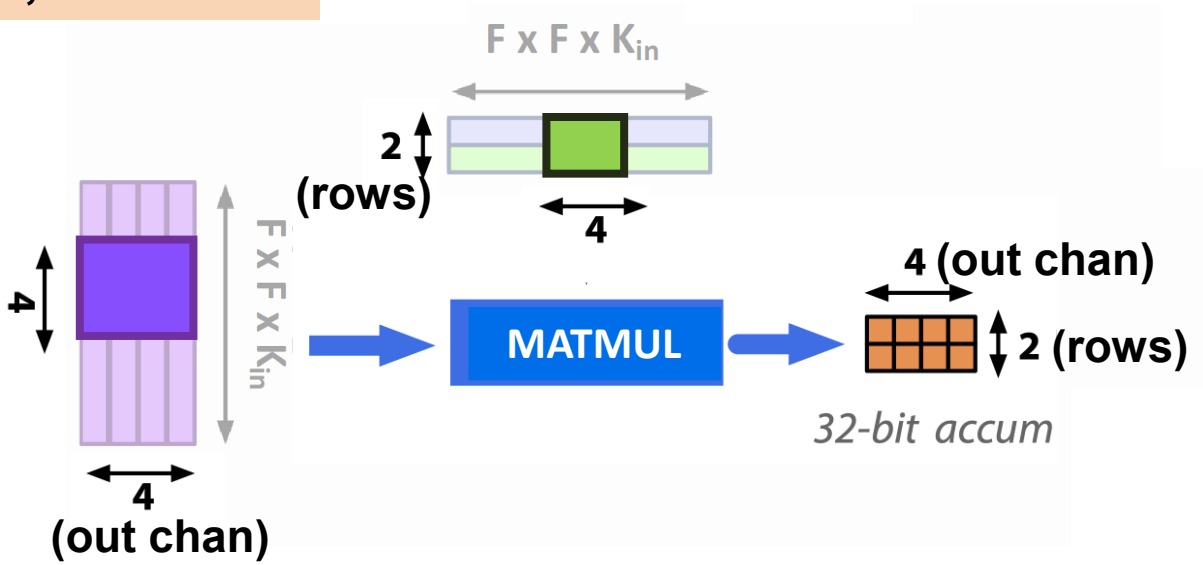
# PULP-NN: optimized computational back-end

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

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

**Loop over in chan, filter size**

```
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

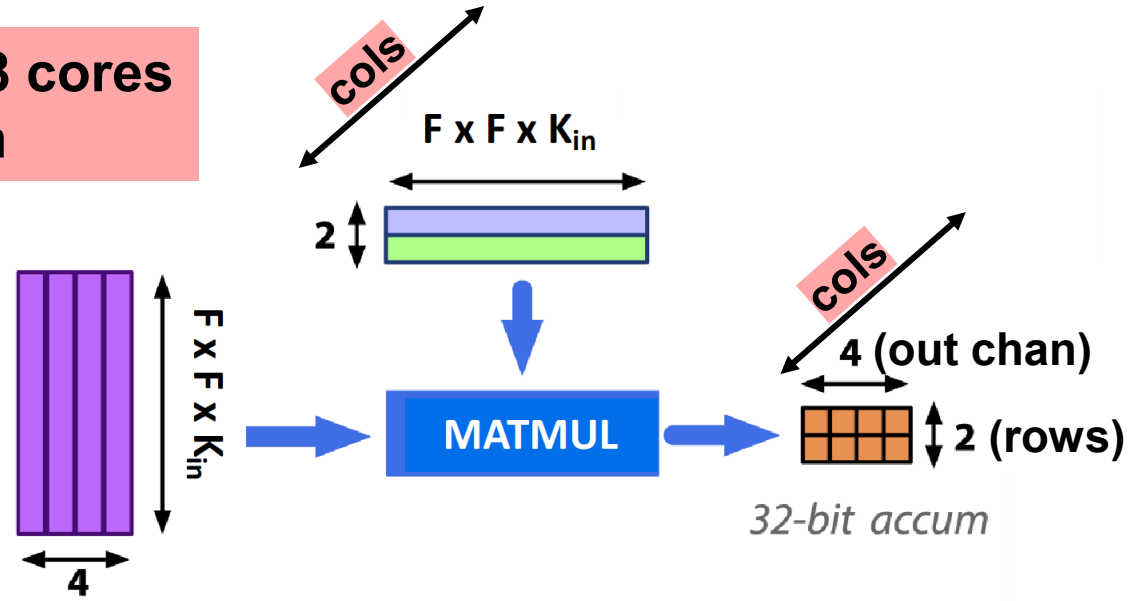
# PULP-NN: optimized computational back-end

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
```

**Parallelize over 8 cores**
column dimension




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

# PULP-NN: optimized computational back-end

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

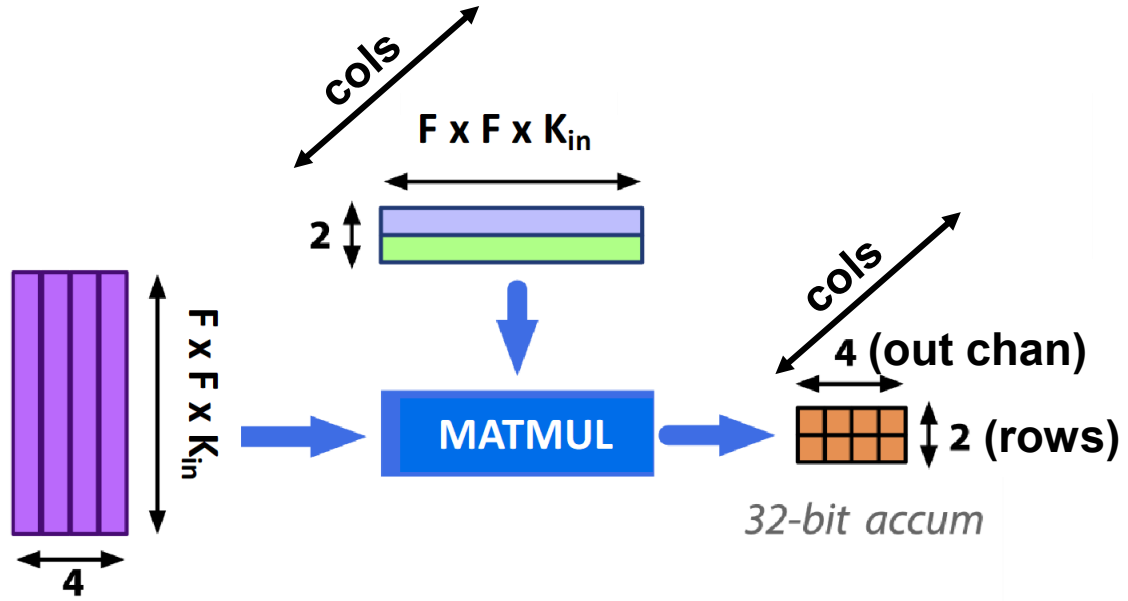
1) maximize **data reuse in register file** 2) improve **kernel regularity** 3) exploit **parallelism**

```
lp.setup
  p.lw  w0, 4(W0!)
  p.lw  w1, 4(W1!)
  p.lw  w2, 4(W2!)
  p.lw  w3, 4(W3!)
  pv.sdotsp.b   acc8, w3, x1
end
```

Peak Performance (8 cores)

**15.5 MAC/cycle**

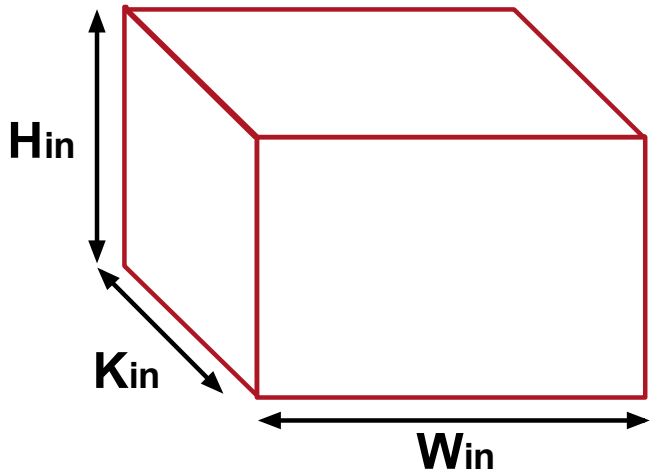
**7.8 Gop/s** @ 250 MHz



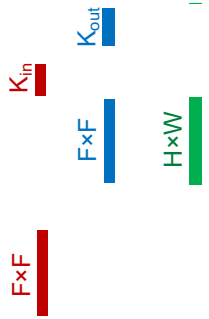

**PULP-NN** [Garofalo 19] https://arxiv.org/abs/1908.11263

# Data Tiling / Blocking

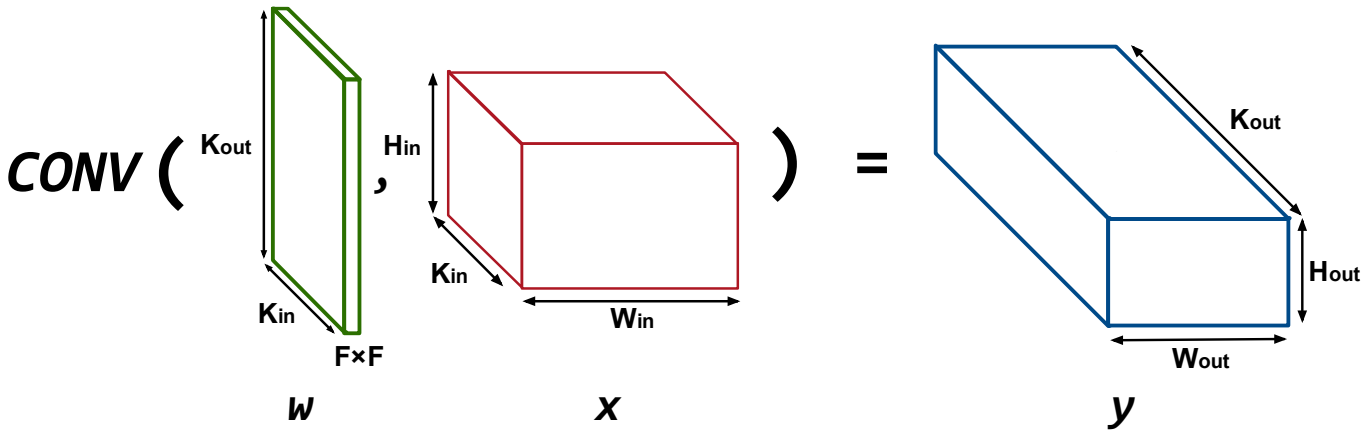
What if the characteristic tensors of the DNN layer (w, x, psum, y) do not fit L1?
Look at the loop nest as a specification on how the tensors are *scanned*…



# Data Tiling / Blocking




```
for m in range(0, K_out_tile):
  for n in range(0, K_in_tile):
    for i in range(0, H_out_tile):
      for j in range(0, W_out_tile):
        psum = b[m]
        for ui in range(0, F):
          for uj in range(0, F):
            psum += w[m,n,ui,uj] * x[n,i+ui,j+uj]
        y[m,i,j] = act(psum)
```

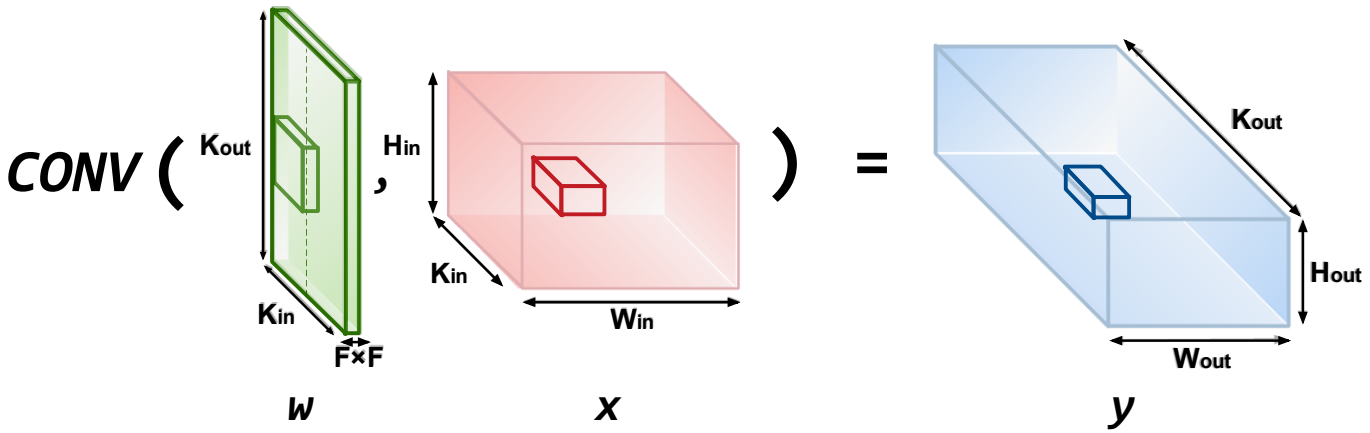


# Data Tiling / Blocking

```
for mm in range(0, nb_K_out_tiles):                    # Kout
    for nm in range(0, nb_K_in_tiles):                 # Kin
        for ii in range(0, nb_H_out_tiles):            # F×F, H×W
            for jj in range(0, nb_W_out_tiles):        # F×F, H×W
                                                       ---- tiling loops
                for m in range(0, K_out_tile):         # Kout
                    for n in range(0, K_in_tile):      # Kin
                        for i in range(0, H_out_tile): # F×F, H×W
                            for j in range(0, W_out_tile):
                                psum = b[m]
                                for ui in range(0, F): # F×F
                                    for uj in range(0, F):
                                        psum += w[m,n,ui,uj] * x[n,i+ui,j+uj]
                                y[m,i,j] = act(psum)
                                                       ---- tile-level operation
```

*tiling loops*

*tile-level operation*



# Why tiling?

- Match the **reuse distance** of a tensor to the available storage to **exploit data reuse** at that level





# Why tiling?

- Match the **reuse distance** of a tensor to the available storage to **exploit data reuse** at that level

- Align the range of tile-level loops to **hardware features**
  - limited **number of cores** → parallelism
  - limited **scalability of interconnects** → broadcast / multicast
  - limited **cache / scratchpad size** → temporal reuse (see point above)





# Why tiling?

- Match the **reuse distance** of a tensor to the available storage to **exploit data reuse** at that level

- Align the range of tile-level loops to **hardware features**
  - limited **number of cores** → parallelism
  - limited **scalability of interconnects** → broadcast / multicast
  - limited **cache / scratchpad size** → temporal reuse (see point above)

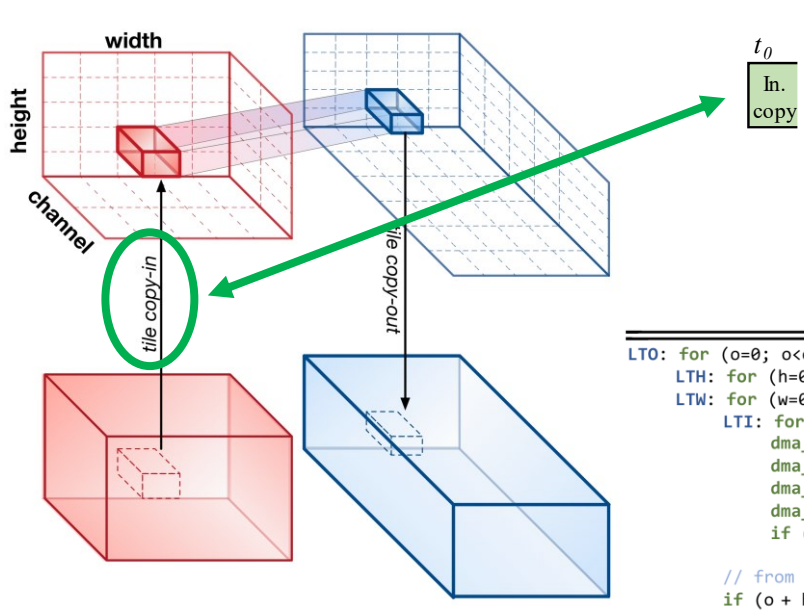
- Hide **memory transfer costs**
  - pipelining of computation and data movement with **balanced stages**





# Tiling example: DORY

- *Special purpose SW cache* for a cache-less microcontroller platform DRAM + 512kB L2 + 128kB L1

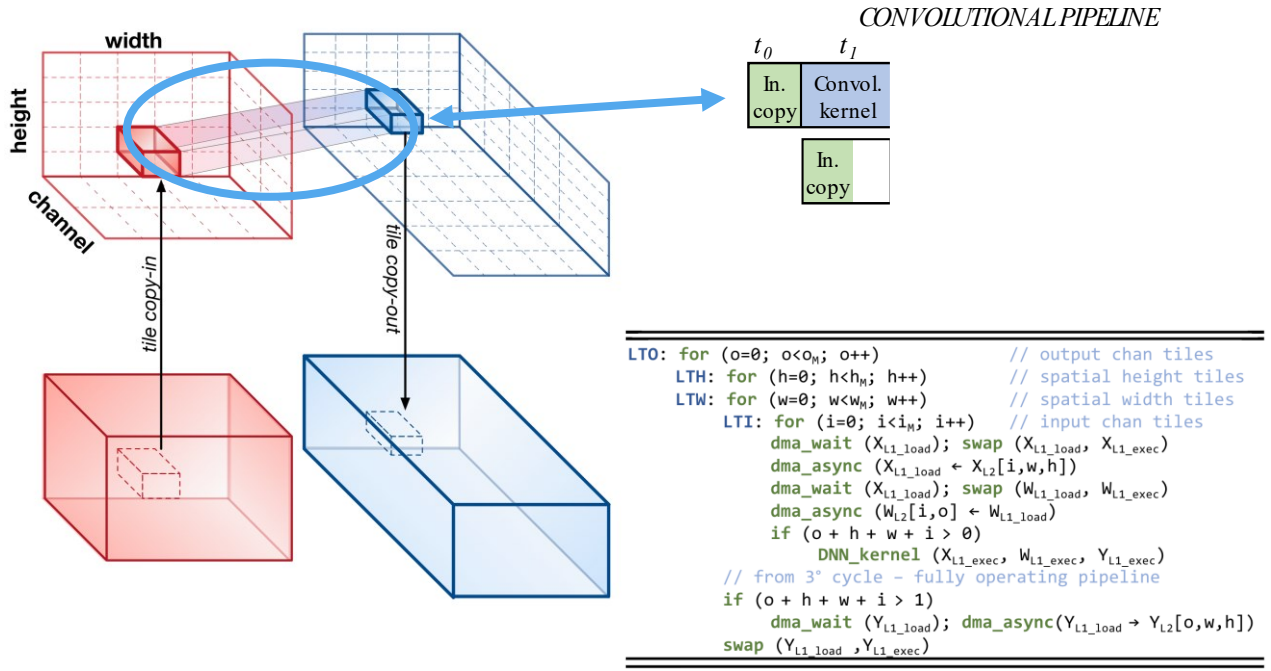



```
LTO: for (o=0; o<o_M; o++)                    // output chan tiles
     LTH: for (h=0; h<h_M; h++)               // spatial height tiles
     LTW: for (w=0; w<w_M; w++)               // spatial width tiles
          LTI: for (i=0; i<i_M; i++)          // input chan tiles
               dma_wait (X_L1_load); swap (X_L1_load, X_L1_exec)
               dma_async (X_L1_load ← X_L2[i,w,h])
               dma_wait (X_L1_load); swap (W_L1_load, W_L1_exec)
               dma_async (W_L2[i,o] ← W_L1_load)
               if (o + h + w + i > 0)
                    DNN_kernel (X_L1_exec, W_L1_exec, Y_L1_exec)
          // from 3° cycle - fully operating pipeline
          if (o + h + w + i > 1)
               dma_wait (Y_L1_load); dma_async(Y_L1_load → Y_L2[o,w,h])
          swap (Y_L1_load ,Y_L1_exec)
```

*Burrello et al.* [https://arxiv.org/pdf/2008.07127.pdf](https://arxiv.org/pdf/2008.07127.pdf)

# Tiling example: DORY

- *Special purpose SW cache* for a cache-less microcontroller platform DRAM + 512kB L2 + 128kB L1

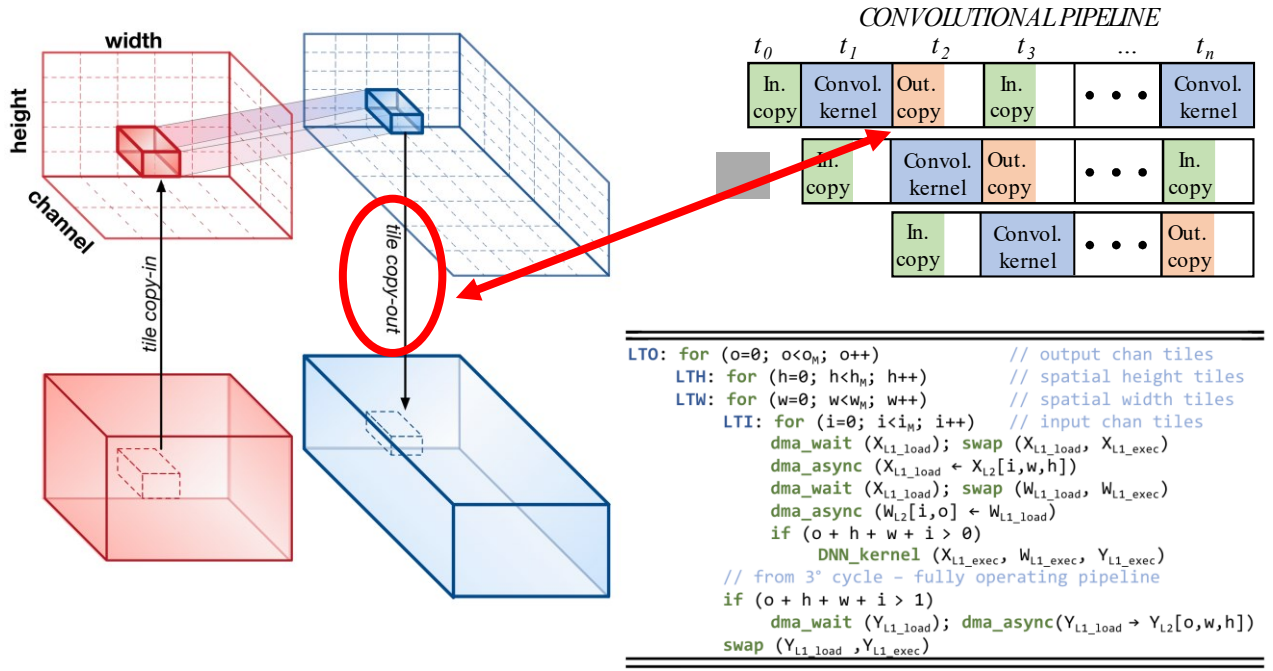



```
LTO: for (o=0; o<o_M; o++)                    // output chan tiles
     LTH: for (h=0; h<h_M; h++)               // spatial height tiles
     LTW: for (w=0; w<w_M; w++)               // spatial width tiles
          LTI: for (i=0; i<i_M; i++)          // input chan tiles
               dma_wait (X_L1_load); swap (X_L1_load, X_L1_exec)
               dma_async (X_L1_load ← X_L2[i,w,h])
               dma_wait (X_L1_load); swap (W_L1_load, W_L1_exec)
               dma_async (W_L2[i,o] ← W_L1_load)
               if (o + h + w + i > 0)
                    DNN_kernel (X_L1_exec, W_L1_exec, Y_L1_exec)
          // from 3° cycle - fully operating pipeline
          if (o + h + w + i > 1)
               dma_wait (Y_L1_load); dma_async(Y_L1_load → Y_L2[o,w,h])
          swap (Y_L1_load ,Y_L1_exec)
```

*Burrello et al.* [https://arxiv.org/pdf/2008.07127.pdf](https://arxiv.org/pdf/2008.07127.pdf)

# Tiling example: DORY

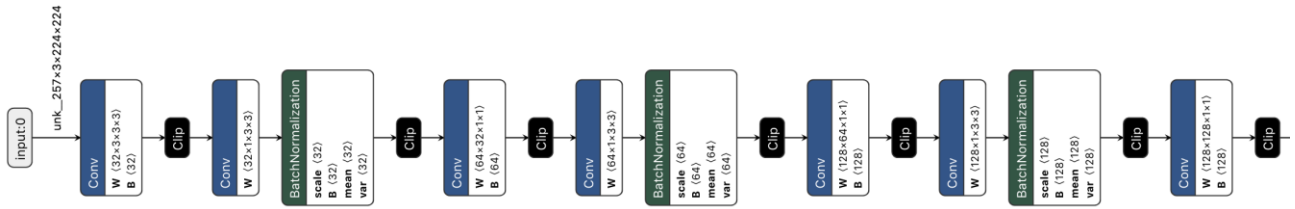
- *Special purpose SW cache* for a cache-less microcontroller platform DRAM + 512kB L2 + 128kB L1

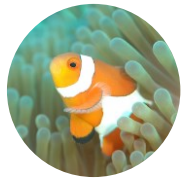



```
LTO: for (o=0; o<o_M; o++)                    // output chan tiles
     LTH: for (h=0; h<h_M; h++)               // spatial height tiles
     LTW: for (w=0; w<w_M; w++)               // spatial width tiles
          LTI: for (i=0; i<i_M; i++)          // input chan tiles
               dma_wait (X_L1_load); swap (X_L1_load, X_L1_exec)
               dma_async (X_L1_load ← X_L2[i,w,h])
               dma_wait (X_L1_load); swap (W_L1_load, W_L1_exec)
               dma_async (W_L2[i,o] ← W_L1_load)
               if (o + h + w + i > 0)
                    DNN_kernel (X_L1_exec, W_L1_exec, Y_L1_exec)
          // from 3° cycle - fully operating pipeline
          if (o + h + w + i > 1)
               dma_wait (Y_L1_load); dma_async(Y_L1_load → Y_L2[o,w,h])
          swap (Y_L1_load ,Y_L1_exec)
```

*Burrello et al.* [https://arxiv.org/pdf/2008.07127.pdf](https://arxiv.org/pdf/2008.07127.pdf)

# PULP SW stack for quantized DNNs



***NEMO***

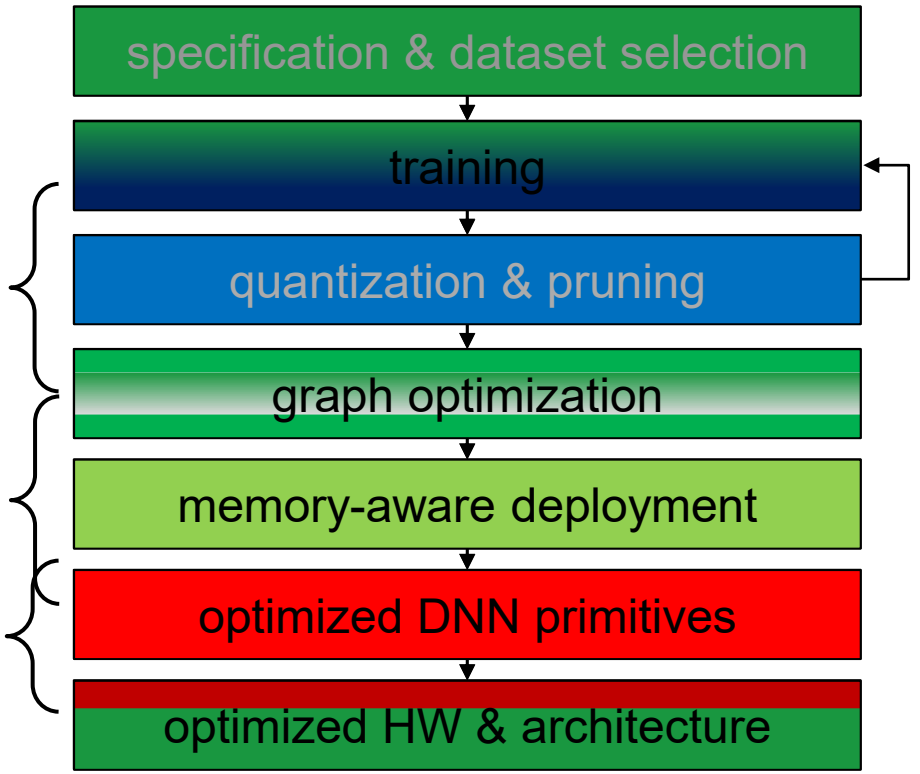
*NEural Minimization for pytOrch*

***DORY***

*Deployment Oriented to memoRY*

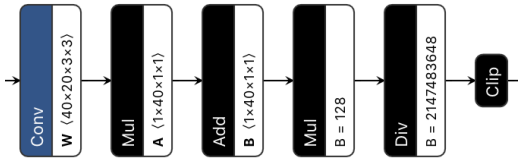
***PULP-NN***

*PULP Neural Network backend*




ETH zürich

# Example: MobileNet v1




**ID Network + tensor sizes**

**Code Generation**
from templates

**Network-level C code**
- L3/L2 transfer boilerplate
- double buffering for weights
- calls to layer-level code

**Layer-level C code**
- L2/L1 transfer boilerplate
- calls to PULP-NN backend library

GREENWAVES TECHNOLOGIES

**1.0-MobileNet-128** on GAP8 (59% top-1 accuracy on ImageNet)

**1.13 GMAC/s** *@ 200 MHz*
**5.67 MAC/cycle** on 8 cores
**165 ms/frame**

peak throughput ~12.6 MAC/cycle

*to put in perspective:*
*MobileNet v1 on STM32-H7 up to* ***~0.52 MAC/cycle*** *[Capotondi 19]*

[Capotondi 19] https://github.com/EEESlab/MobileNet_v1_x_cube_ai_4.1.0

ETH zürich

# Approaching the PULP ecosystem

- The PULP SDK (https://github.com/pulp-platform/pulp-sdk) includes a **platform simulator** (**GVSOC**) and **a set of basic SW libraries**
- GVSOC performance:
  - Around 1MIPS simulation speed
  - Functionally aligned and calibrated with HW
  - Timing accuracy is within 10-20% of target HW
- The PULP SDK adopts the **same low-level programming interface (PMSIS)** and the **same build system** (Make) for all the supported targets (GVSOC, RTL, FPGA)

ETH zürich



# Looking for collaborators!

- **Available HW Projects**
  - Advanced Interconnects for Multi-Accelerator Support
  - FPGA Emulation of Noisy Memories for Ultra-Low-Voltage Error-Resilient Inference
  - FP8 and Breakable Multiply-Add units for ULP Learning
  - SIMD/MIMD Support for RISC-V Cores
  - Zero-skipping HW Acceleration Support

- **Available SW Projects**
  - Compiler and run-time support for leveraging the VLEM mode
  - Compiler and run-time support for mixed-precision arithmetic
  - Compiler-level optimization of DSP and accelerator-style extensions (e.g., MAC&load instruction)

- **…and many many others!**