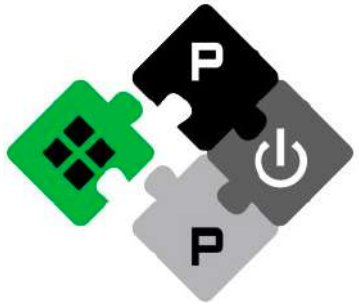


PULP PLATFORM

Open Source Hardware, the way it should be!

Low Power Multicore Solutions for Approximation

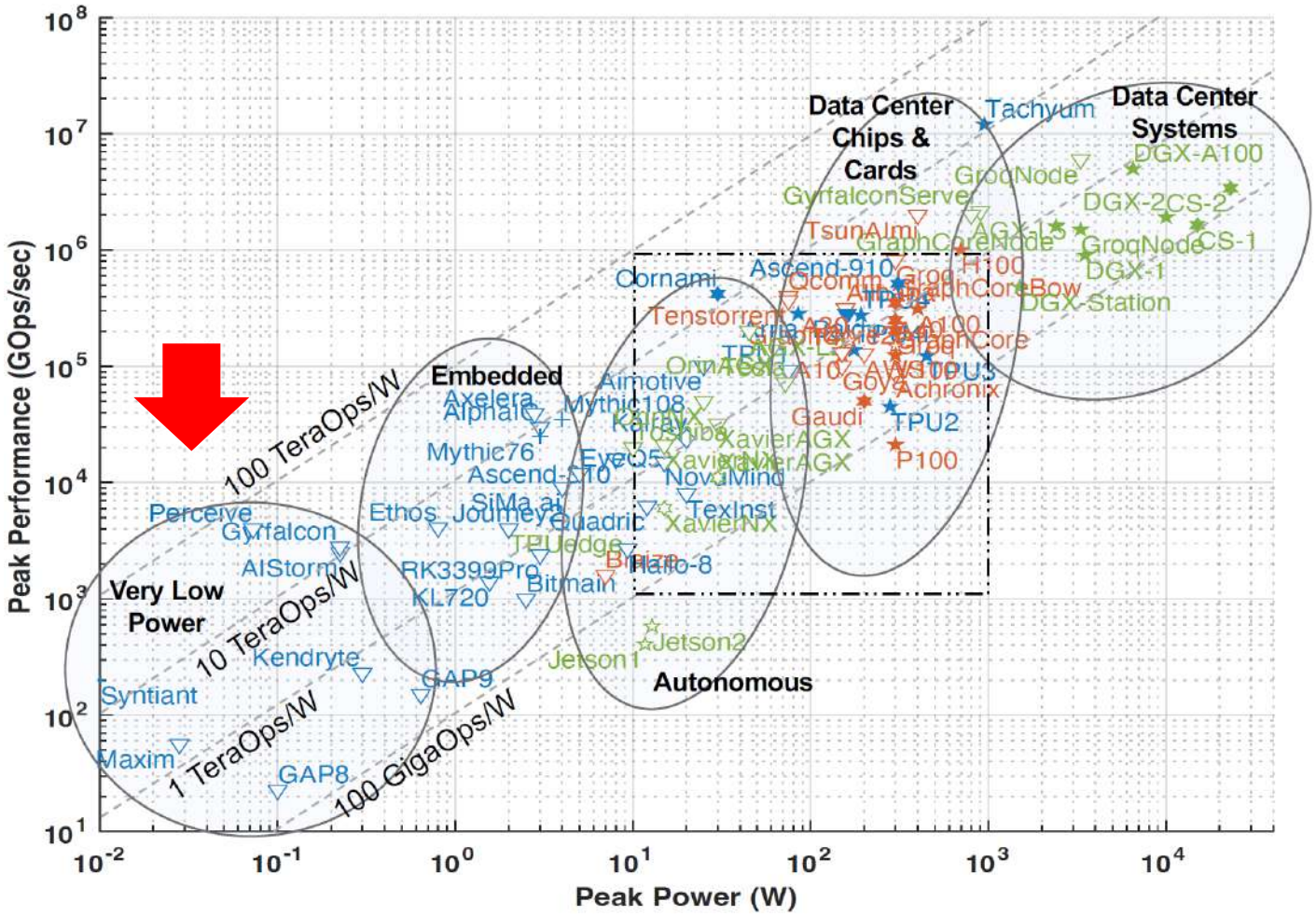
Luca Benini **<lbenini@iis.ee.ethz.ch,luca.benini@unibo.it>**

ETH zürich

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

ML Processors from Tiny to Huge

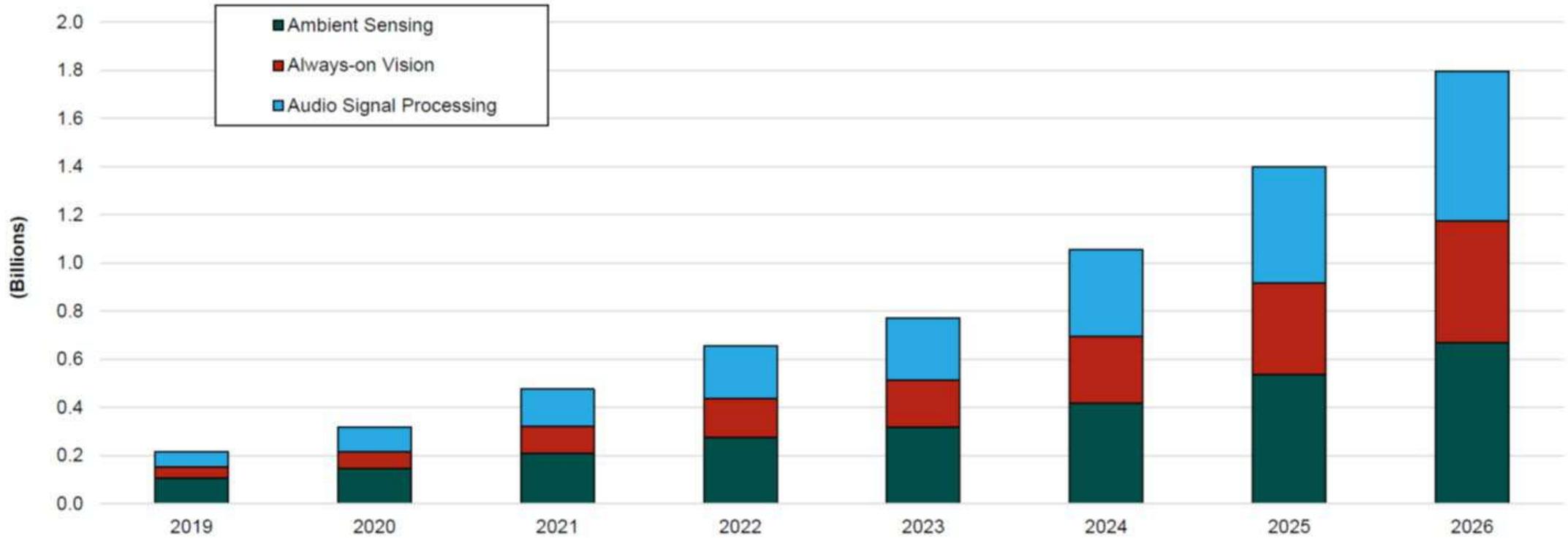
[ReutherHPEC22]



CloudML → TinyML

TinyML Opportunity

[ABI research 21]

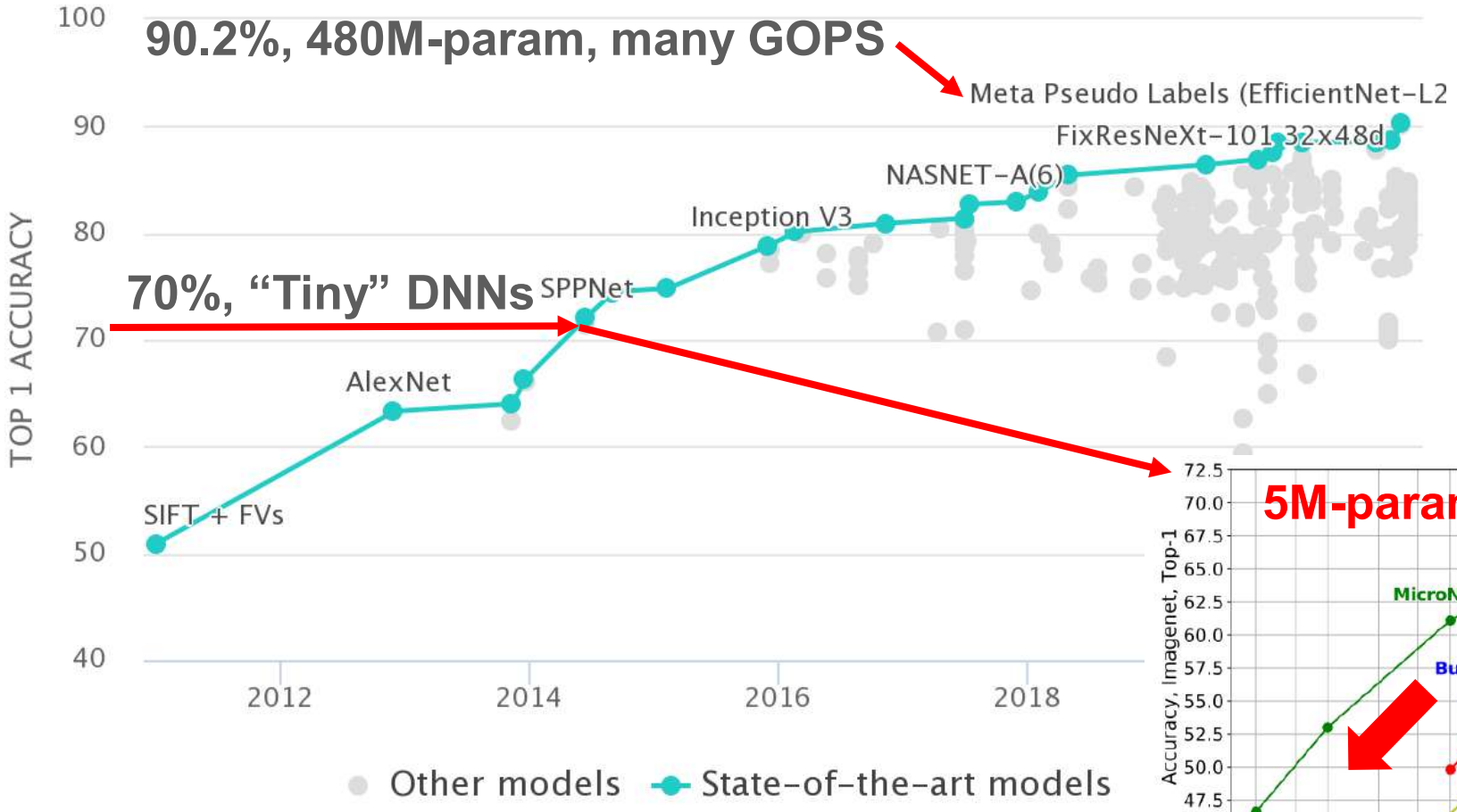


TinyML challenge

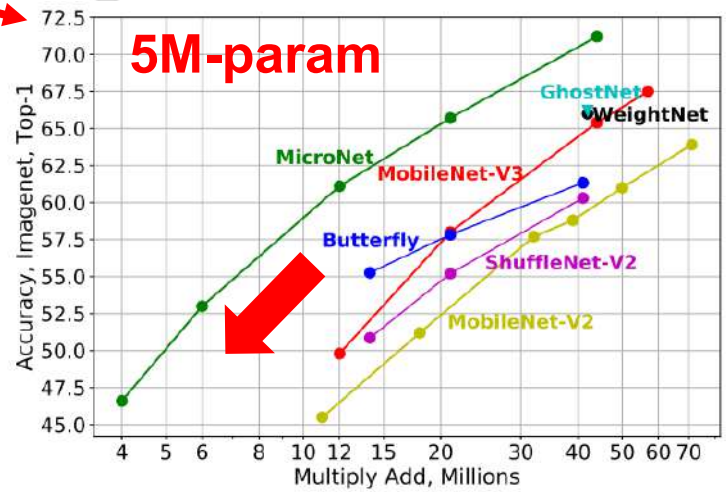
AI capabilities in the power envelope of an MCU: **10-mW peak (1mW avg)**

TinyML Workloads – DNNs (and More)

H Pham 2021(Google) arXiv:2003.10580v3



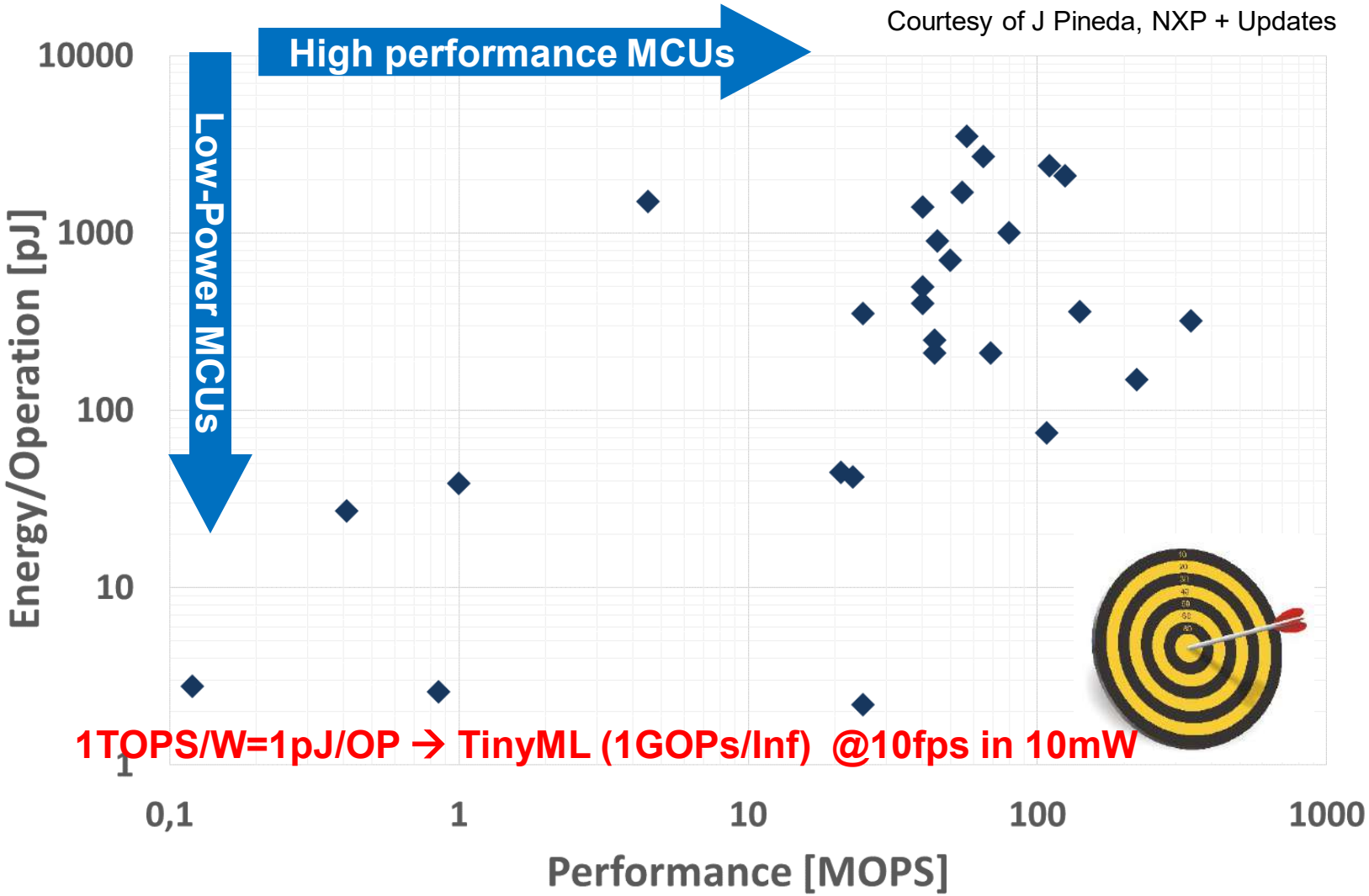
High OP/B ratio
Massive Parallelism
MAC-dominated
Low precision OK



"Model zoo": very fast evolution → need programmable solutions

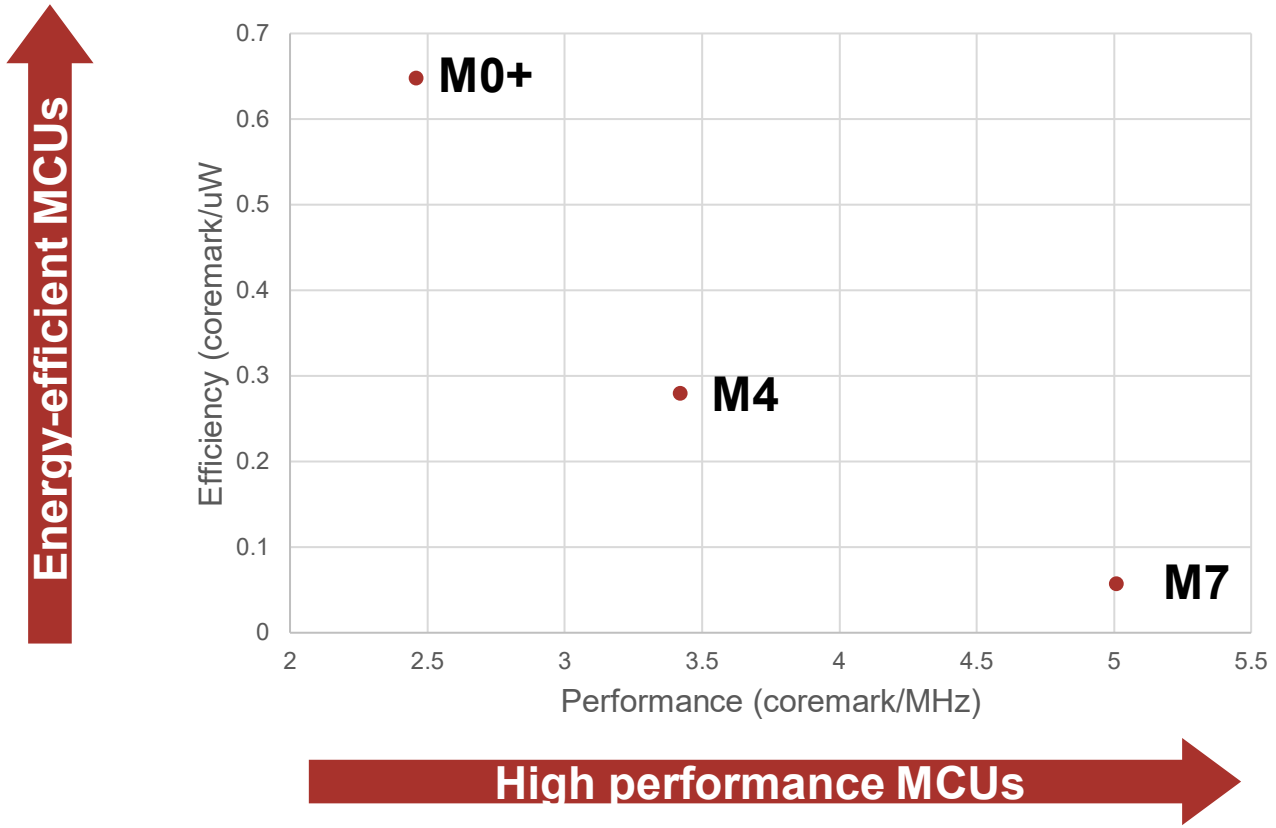
ML on MCUs?

Courtesy of J Pineda, NXP + Updates



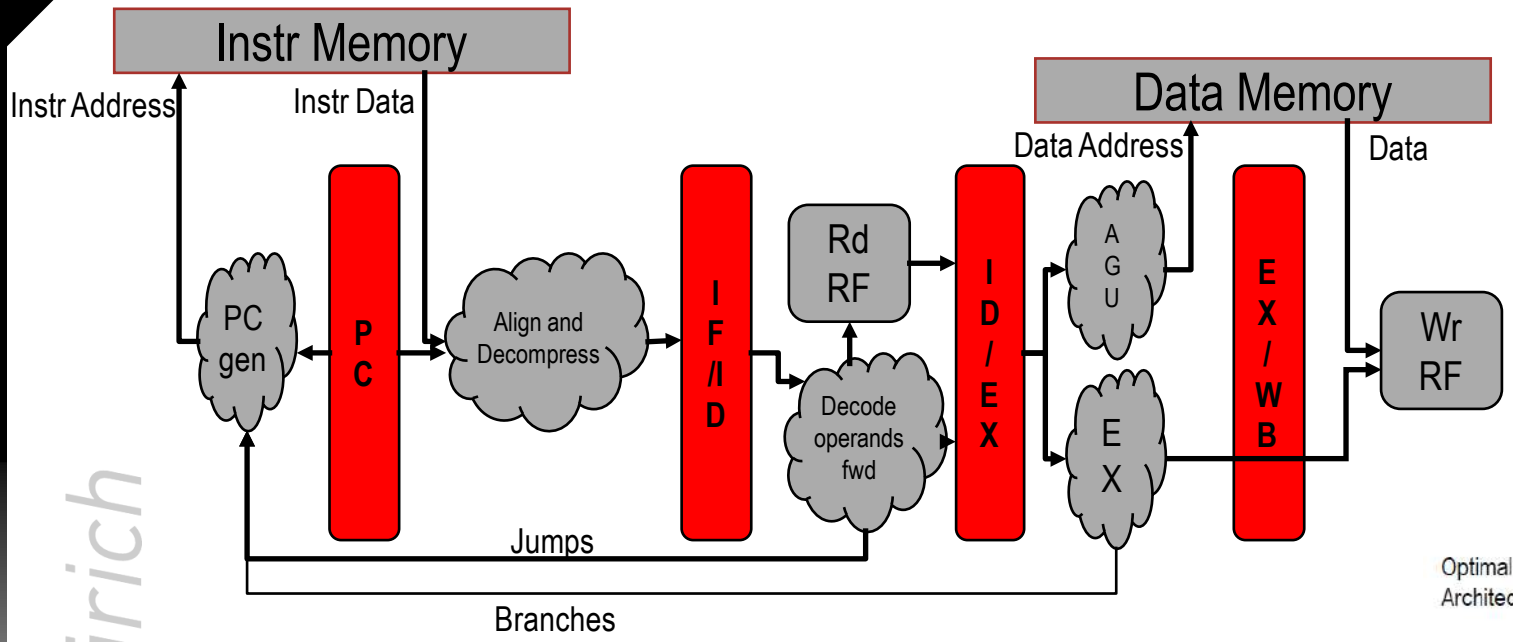
Energy efficiency @ GOPS is the Challenge

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

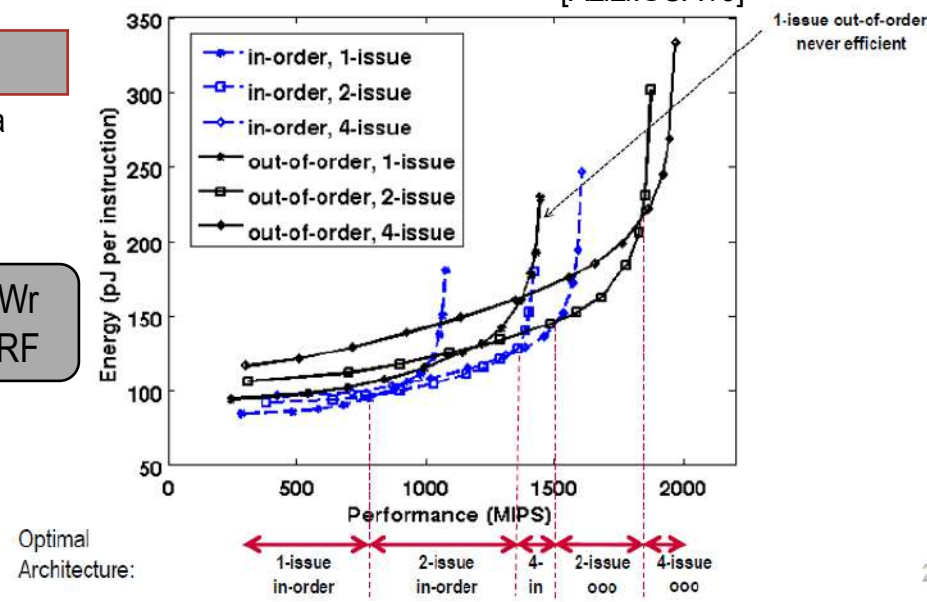


*data from ARMs web

The Tunnel: High-Performance vs. Energy-Efficient



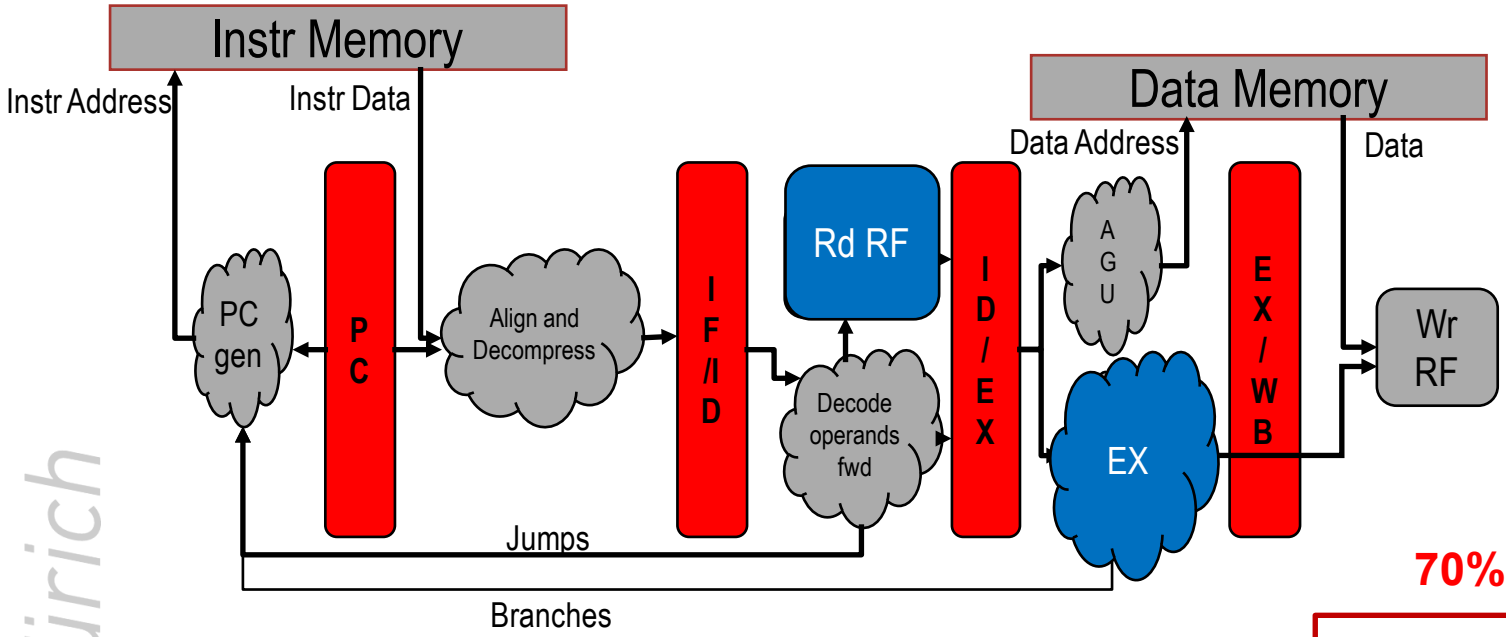
[AziziISCA10]



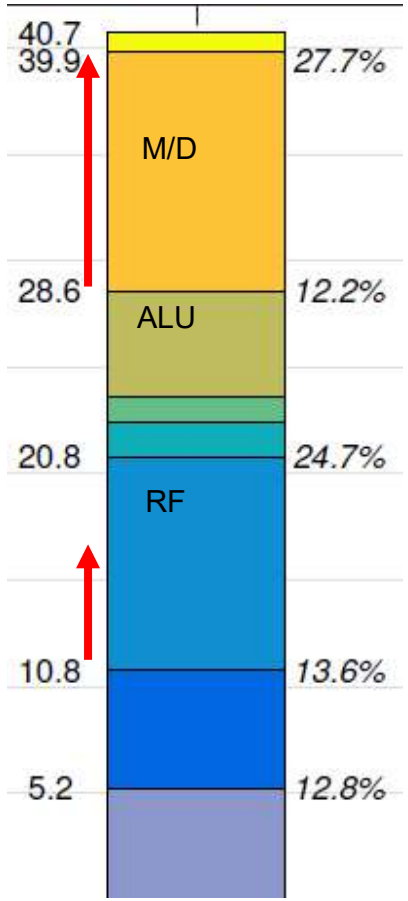
- **"Classical" core performance scaling trajectory**
  - Faster CLK → deeper pipeline → IPC drops
  - Recover IPC → superscalar → ILP bottleneck (dependencies)
  - Mitigate ILP bottlenecks → OOO → huge power, area cost!

A way Out: Processor Specialization

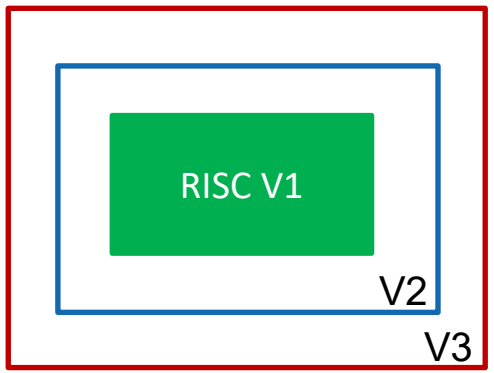
3-cycle ALU-OP, 4-cyle MEM-OP→ only IPC loss: LD-use, Branch



[Gautschi et al. TVLSI 2017]



70% RF+DP



V1 Baseline RISC **(not good for ML)**

V2
Extensions for Data Processing
Data motion (e.g. auto-increment)
Data processing (e.g. MAC)

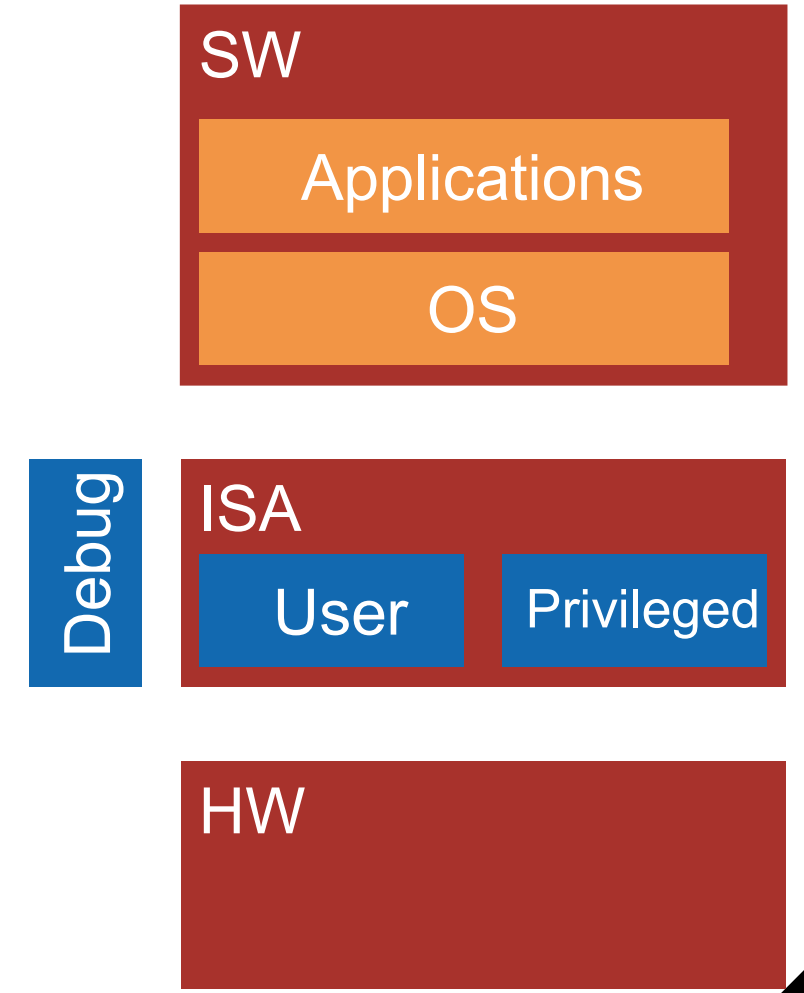
V3
Domain specific data processing
Narrow bitwidth
HW support for special arithmetic

ISA extension cost 25 kGE → 40 kGE (1.6x), energy efficient if $0.6T_{exec}$

ETH zürich

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)
- At ETHZ+UNIBO we specialize in **efficient implementations of RISC-V cores**



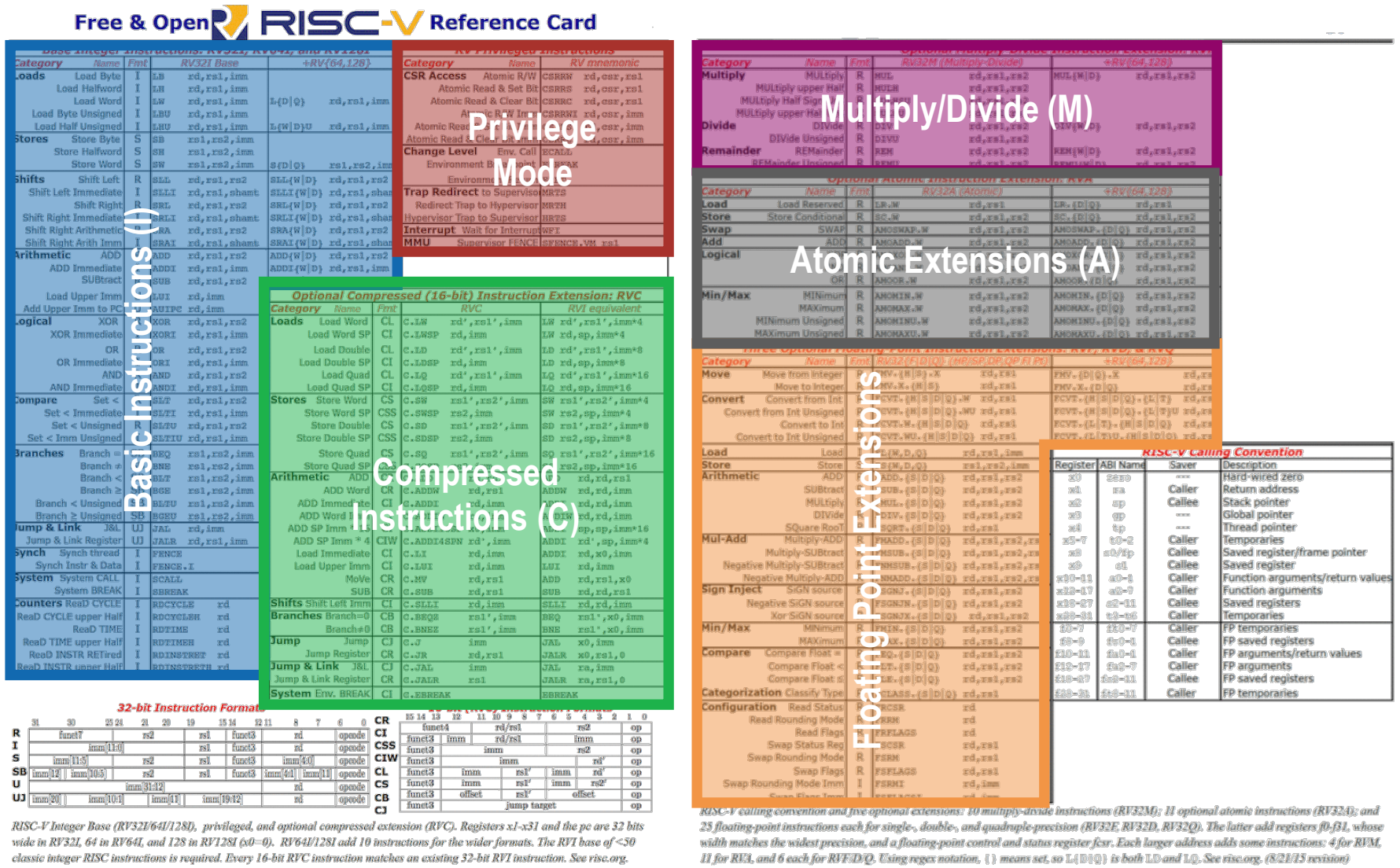
RISC-V Foundation Members

RISC-V ISA Baseline and 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

Baseline + Extensins in a Page

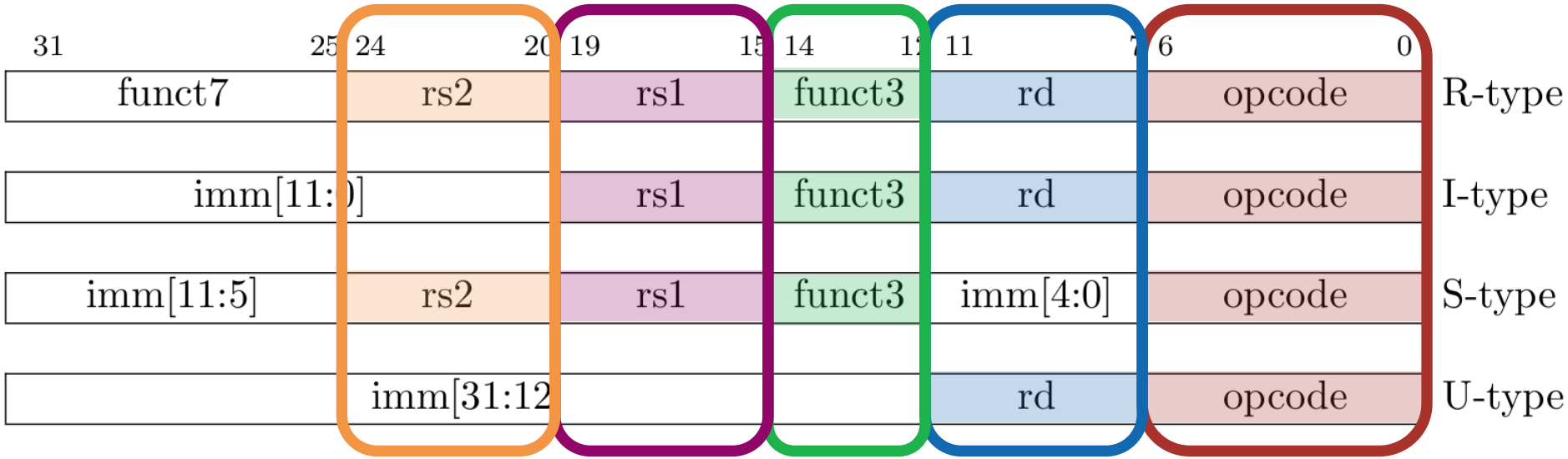


RISC-V Architectural State

- **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.

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 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 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)

Encoding of the instructions, main groups

- **Reserved** opcodes for standard extensions
- Rest of opcodes free for **custom** implementations
- Standard extensions will be frozen/not change in the future

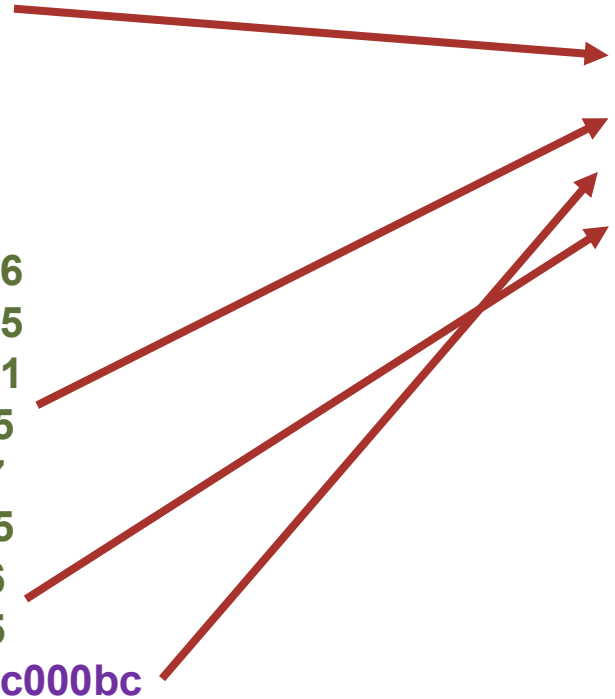
inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*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

Extensibility is integral to RISC-V ISA design!

How to get efficiency: ISA extensions

Only one MACs every 4 cycles!

```
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
```



1. Post modified LD/ST
2. MAC
3. HW loop
4. Packed-SIMD operations with dot product
5. Shuffle operations for vectors
6. Mac-load

M. Gautschi et al., "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in IEEE TVLSI, Oct. 2017.

Post increment LD/ST

- **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

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

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
```

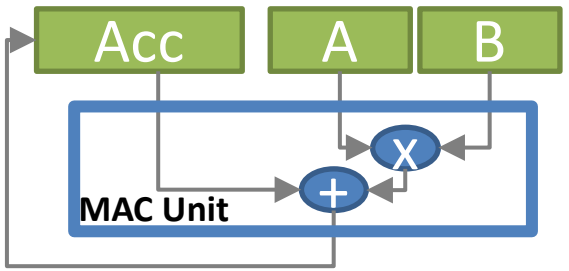
HW Loop Ext

```
// 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!

Multiply Accumulate

- **Accumulation on 32 bit data p.mac**

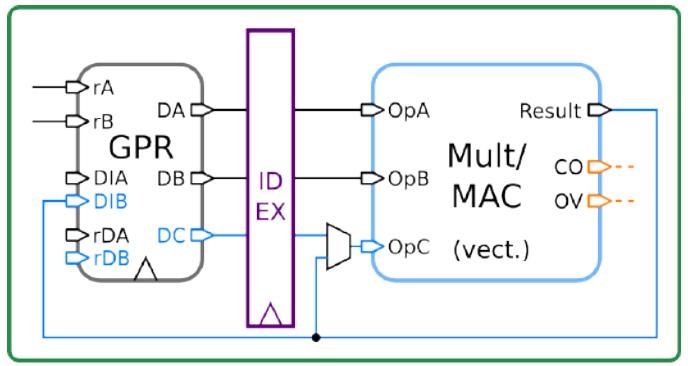



```
int acc=0, coeff[N], inp[N];
for(int i=0; i<N; i++)
    acc += coeff[i] * inp[i];
```

```
acc = __builtin_pulp_mac ( inp[i],
                           coeff[i],
                           acc);
```

Intrinsics: special functions that map directly to inlined DSP instructions.
However, the compiler can already place the p.mac instruction into the above code!

- **Directly on the register file**
- **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



Xpulp Extentions: packed-SIMD

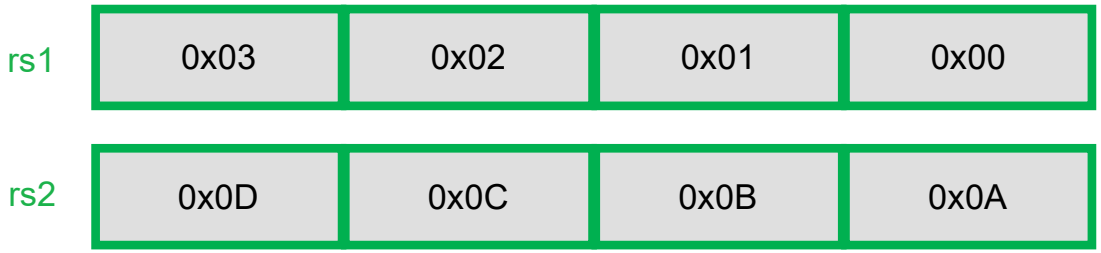
Remember: DNN inference is OK with low-bitdwidth operands

- **packed-SIMD extensions**
  - Make usage of resources the best in performance with little overhead
  - Target for embedded systems, RVV is for high performance
- pSIMD in 32bit machines
  - Vectors are either 4 8bits-elements or 2 16bits-elements
- pSIMD instructions

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

Xpulp Extensions: packed-SIMD

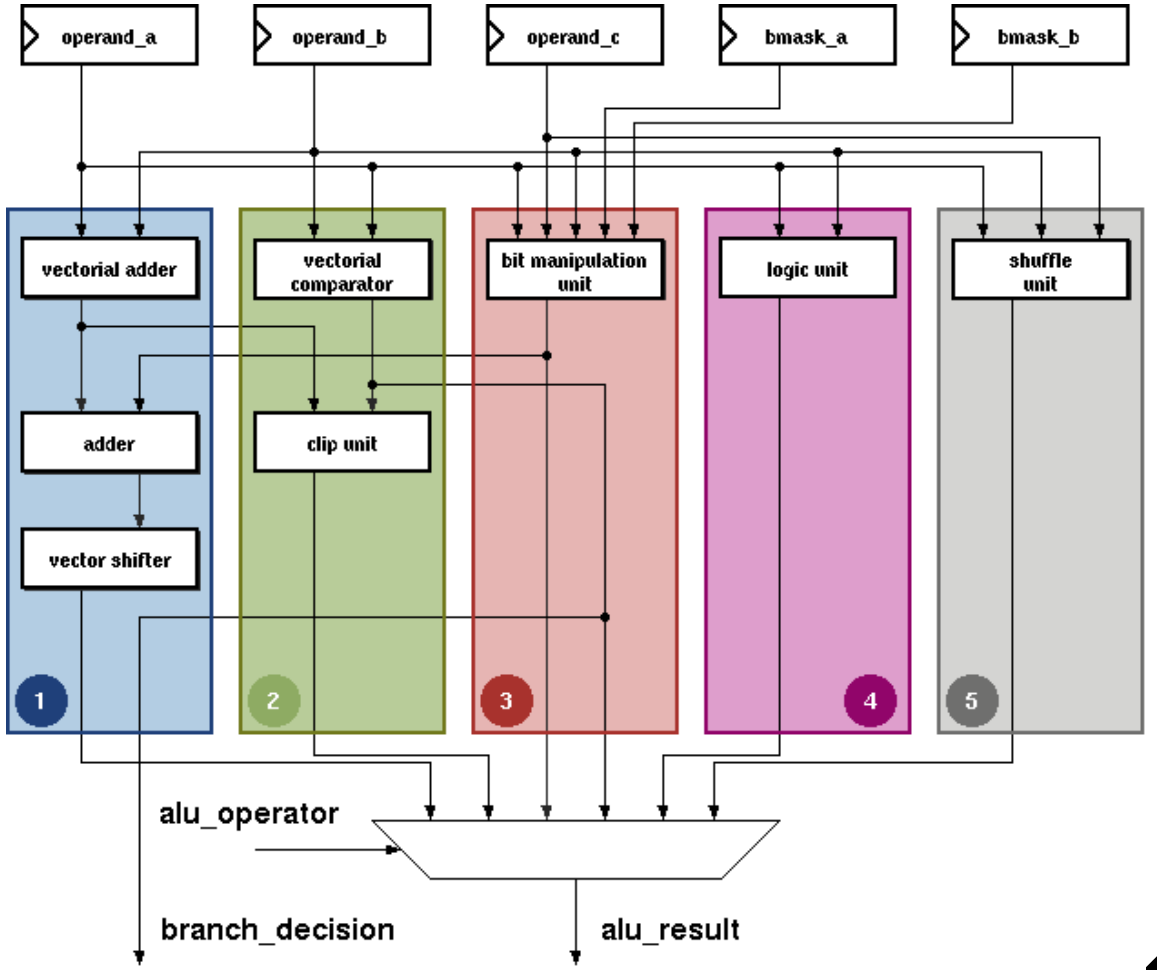
- **Same Register-file**
  - The instruction encode how to interpret the content of the register



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

ALU architecture

- Advanced ALU for Xpulp extensions
- Optimized datapath to reduce resources
- Multiple-adders for round
- Adder followed by shifter for fixed point normalization
- Clip unit uses one adder as comparator and the main comparator



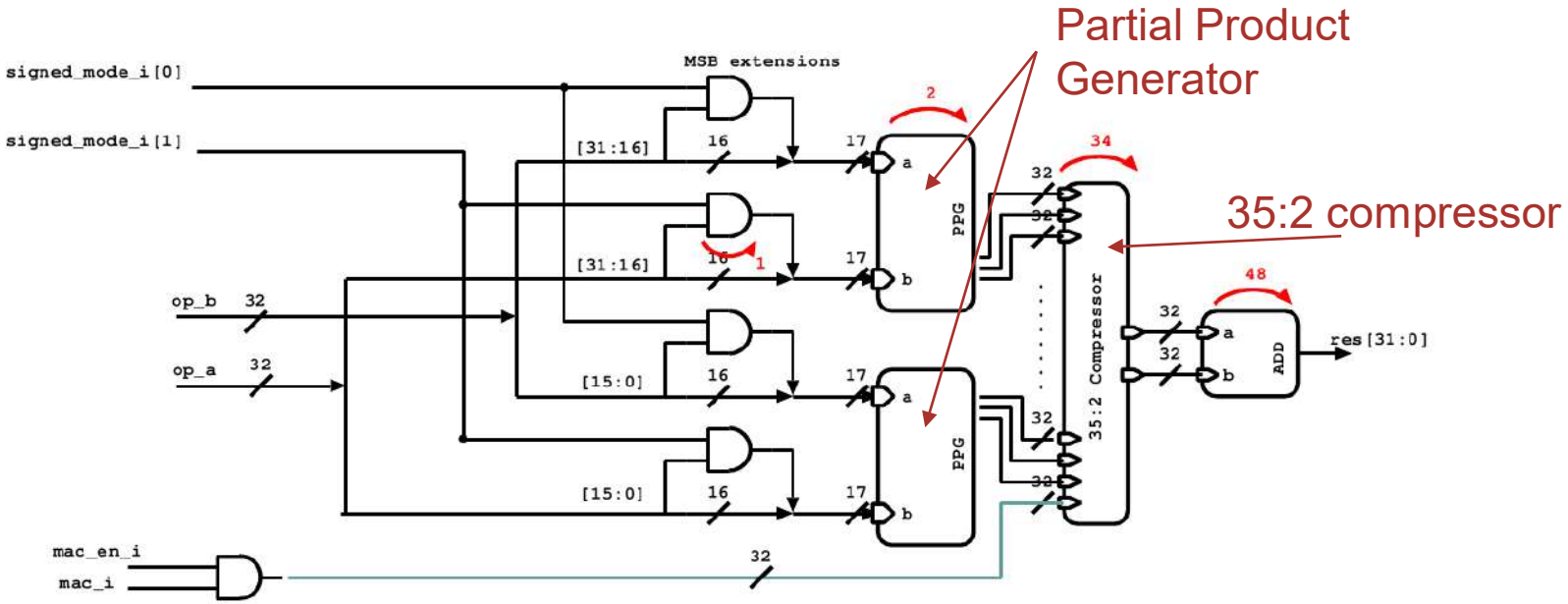
Expanding SIMD Dot Product

- Dot Product: (half word example)

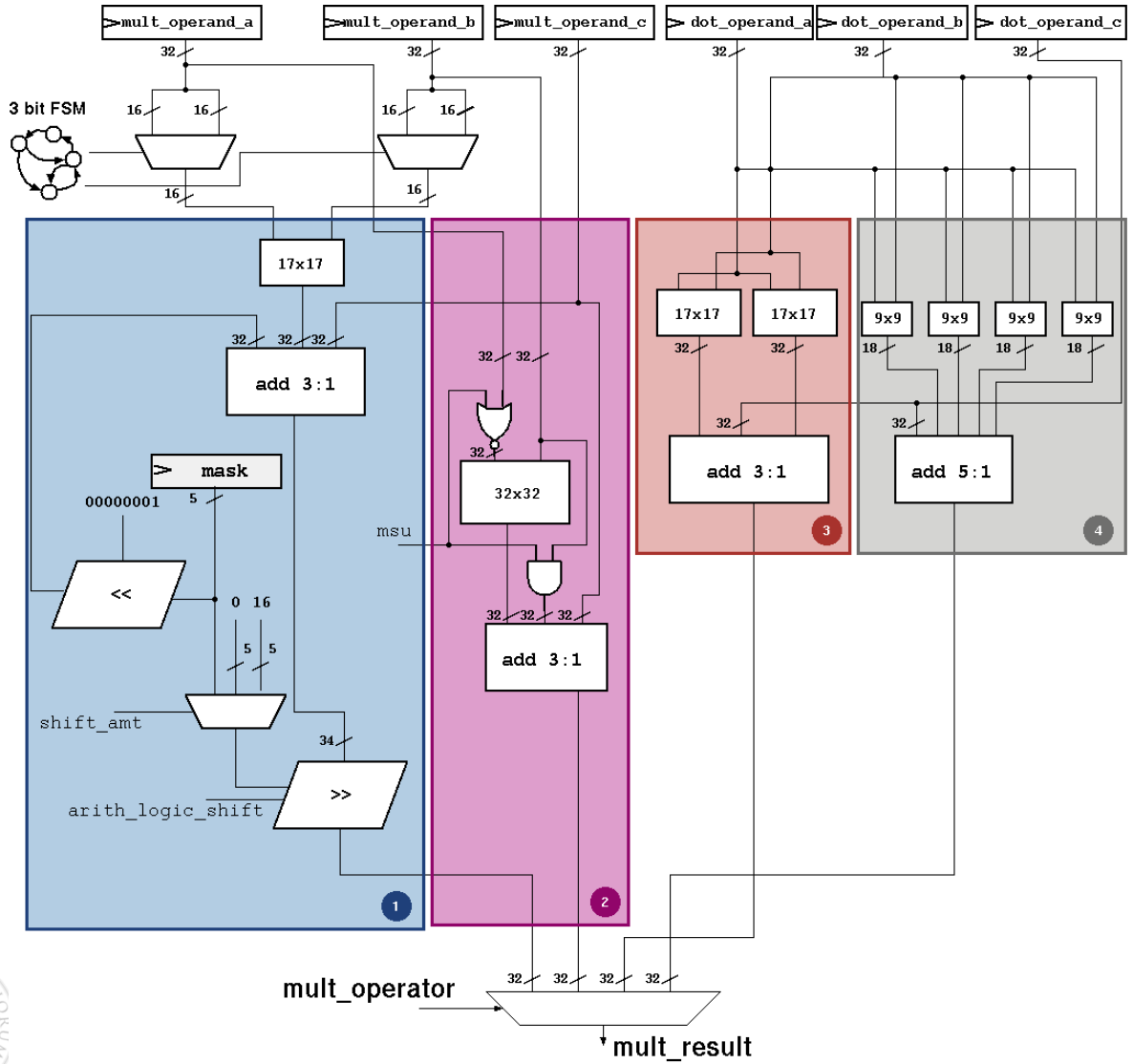
  C[31:0] = A[31:16]*B[31:16] + A[15:0]*B[15:0] + C[31:0]

  32 bit 32 bit 32 bit

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



MUL architecture



16x16b with sign selection for short multiplications [with round and normalization]. 5 cycles FSM for higher 64-bits (mulh* instructions)

32x32b single cycle MAC/MUL unit

16x16b short parallel dot product

8x8b byte parallel dot product

clock gating to reduce switching activity between the scalar and SIMD multipliers

Reference & Examples on Compiler Builtins

SIMD Instructions of the Xpulp ISA extension

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

Dot-product without accumulation between unsigned char vectors (v4u):

```
S = __builtin_dotup4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3], A is v4u, B is v4u
```

Dot-product without accumulation between signed char vectors (v4s):

```
S = __builtin_dotsp4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3] , A is v4s, B is 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
```

Similar builtins without accumulation for short vectors:

```
S = __builtin_dotup2(A, B); S = __builtin_dotsp2(A, B); S = __builtin_dotusp2(A, B);
```

All of these are also available 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);
```

ISA Extensions at Work

- **The innermost loop has 4x less iterations**
  - 4 bytes per matrix are loaded as a 32b word
  - Dot product with accumulation performs in 1 cycle 4 macs

```
...
lp.setup    x1,a4,stop1
p.lbu   a0,1(a3!)
p.lbu   a1,32(a2!)
stop1: p.mac   a5,a0,a1
....
```

```
… //iterate #COL/4
lp.setup x1,a6,stop1
p.lw    a1,4(t1!) //load 4-bytes with post inc
p.lw    a5,4(t3!)
stop1: pv.sdotsp.b a7,a1,a5 //4 mac
....
```

Extensions at work

```
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
```

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
```

HW Loop

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

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!)
```

11 cycles/output | **8 cycles/output** | **5 cycles/output** | **1,25 cycles/output**

Note: All-int computation – hard to quantize. What about FP?
See Luca Bertaccini's lecture!

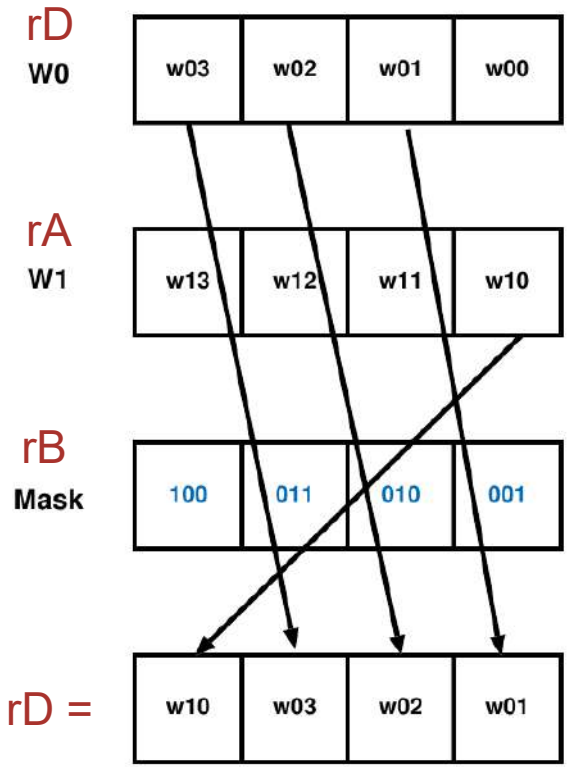
Advanced SIMD: Shuffle Instruction

- In order to use the vector unit the elements have to be aligned in the register file
- Shuffle allows to recombine bytes into 1 register:
- pv.shuffle2.b rD, rA, rB

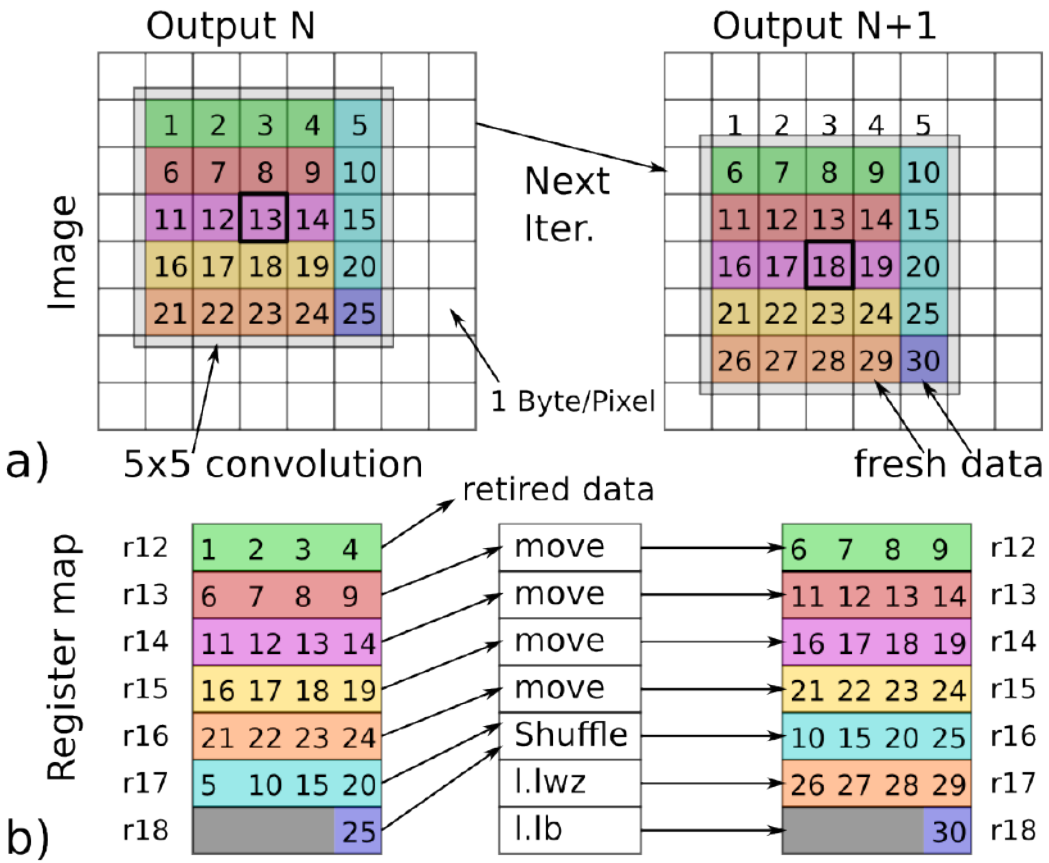
Mask bits

```
rD{3} = (rB[26]==0) ? rA:rD  {rB[25:24]}
rD{2} = (rB[18]==0) ? rA:rD  {rB[17:16]}
rD{1} = (rB[10]==0) ? rA:rD  {rB[ 9: 8]}
rD{0} = (rB[ 2]==0) ? rA:rD  {rB[ 1: 0]}
```

- With rX{i} = rX[(i+1)*8-1:i*8]



Shuffle for Direct SIMD Convolution



Convolution in registers
5x5 convolutional filter

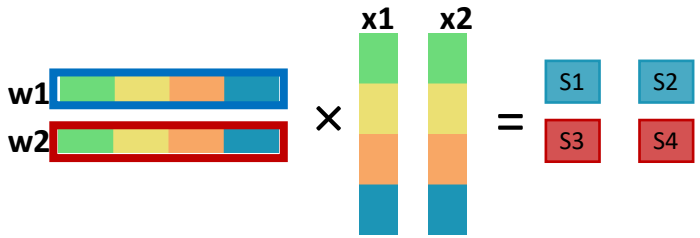
- 7 Sum-of-dot-product
- 4 move
- 1 shuffle
- 3 lw/sw
- ~ 5 control instructions

Significant benefit in reuse of registers and less LD/ST

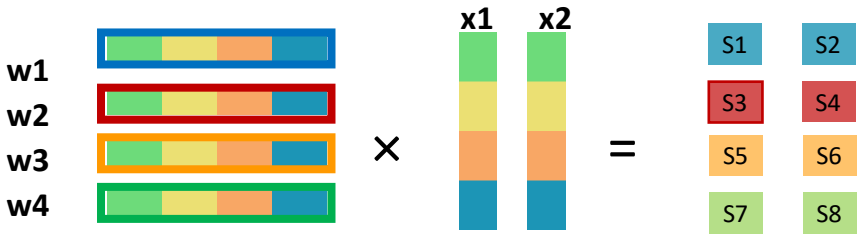
GEMM-based Convolution

8-bit Convolution example

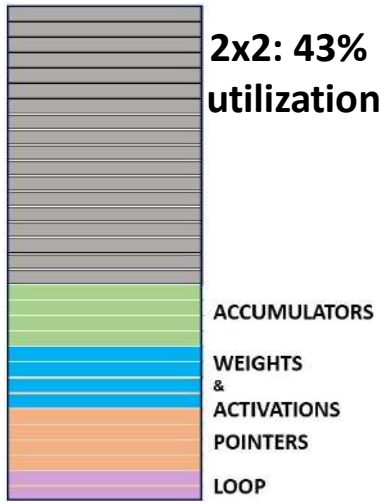
CMSIS-NN based Matrix Multiplication Layout: 2x2



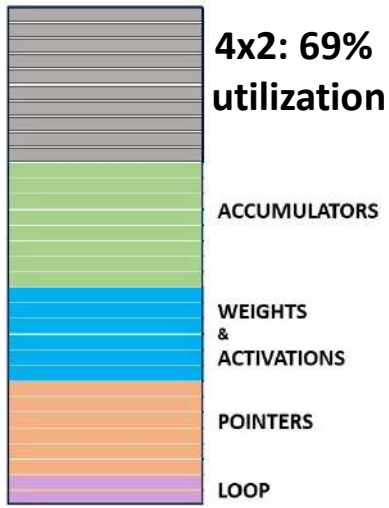
PULP-NN Matrix Multiplication Layout: 4x2



RegisterFile of the RI5CY core: 32 general purpose registers



2x2: 43% utilization



4x2: 69% utilization

More Data Reuse
&
Higher utilization of the RF

Peak Performance (8 cores)

2x2 :12.8 MAC/cyc

4x2 :**15.5 MAC/cyc**

Never underestimate the importance of registers! How to get "more"?

Achieving 100% dotp Unit Utilization

8-bit Convolution

HW Loop

LD/ST with post increment

8-bit SIMD sdotp

8-bit sdotp + LD

RV32IMC

N

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

RV32IMCXpulp

N/4

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

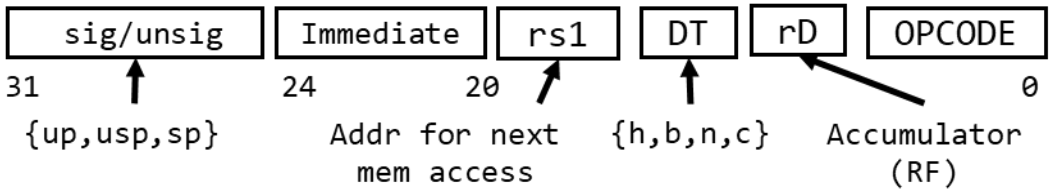
can we remove?

Yes! dotp+ld

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
```

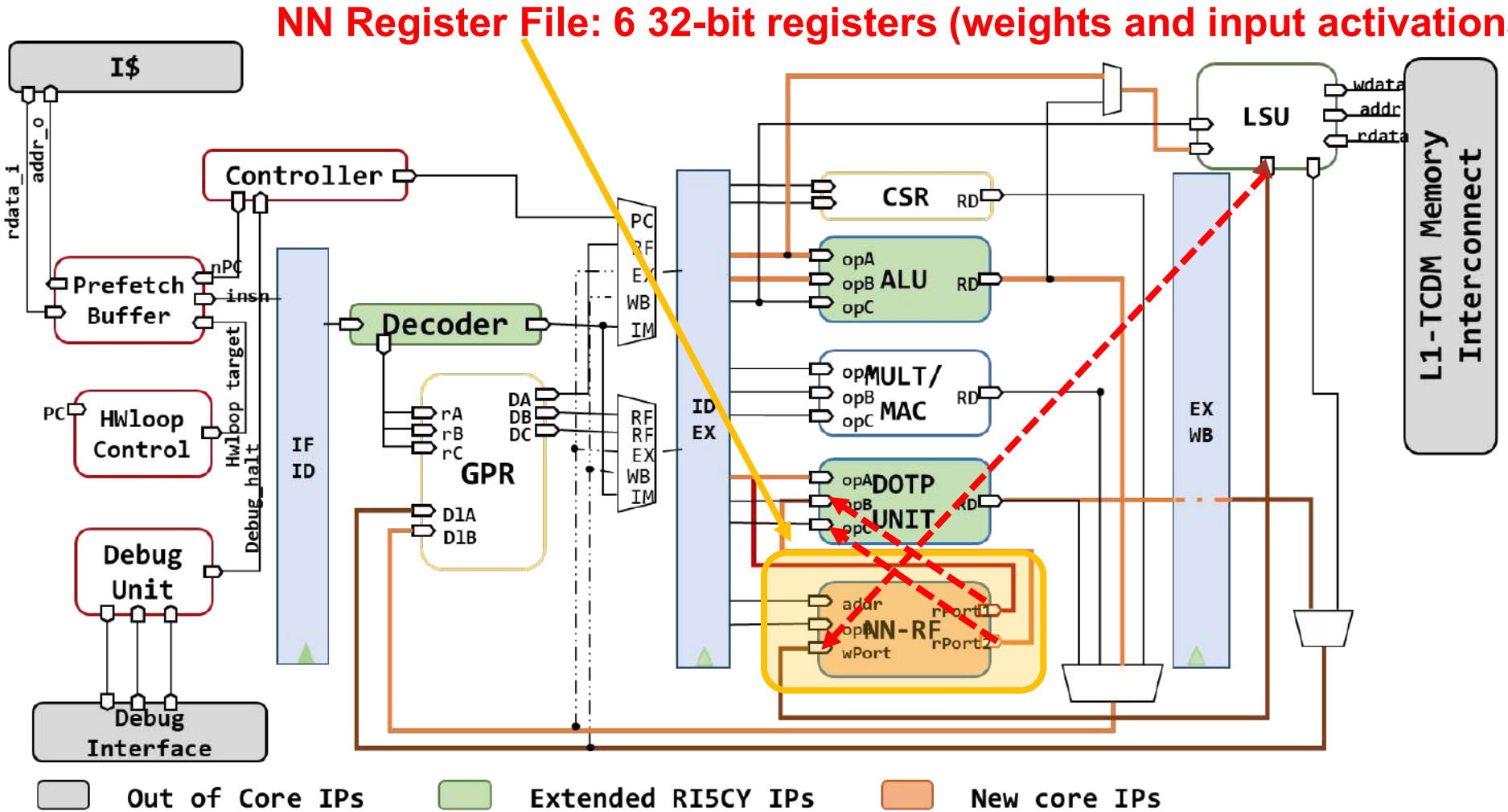
`pv.nnsdot{up,usp,sp}.{h,b,n,c} rD, rs1, Imm`



9x less instructions than RV32IMC

14.5x less instructions at an extra 3% area cost (~600GEs)

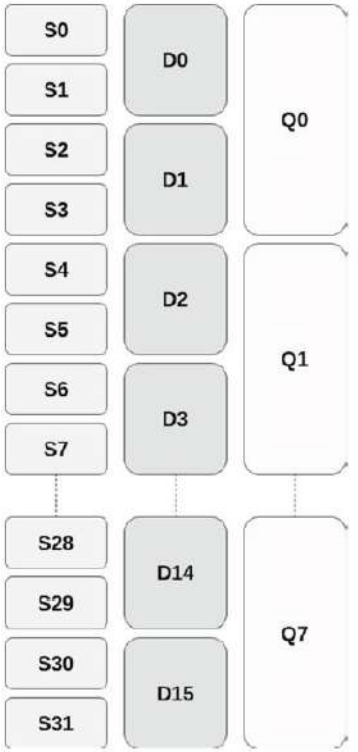
Hardware for dotp+ld



Special-purpose registers

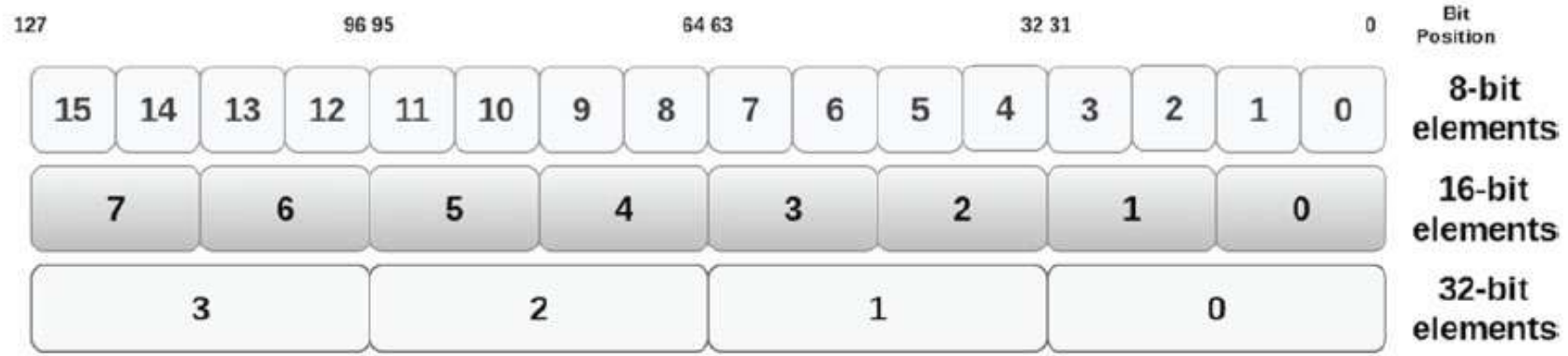
Not only RISC-V: Armv8.1-M

- New embedded vector ISA **Helium (MVE)**
  - **Uses 8 128-bit vector registers** (reuses the 32 FP registers)
- ISA enhancements for loops, branches (Low Overhead Branch Extension)
- Instructions for half-precision floating-point support
- Enhancements in debug including performance monitoring unit (PMU) and additional debug support to focus on signal processing application developments.
  - Being able to set a breakpoint which triggers (halts code execution and passes control to the debugger) when a certain count value is reached and being able to set a data watchpoint with a bit mask for data value comparison (for example, for looking for a signal value to be within a certain range).



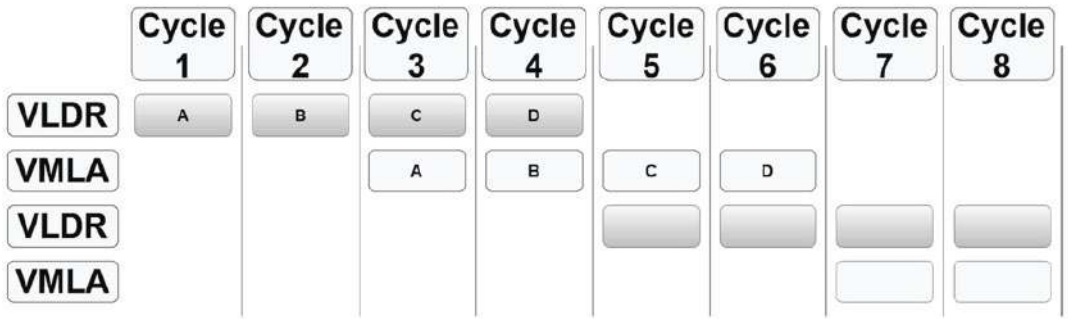
ARM MVE Vectors

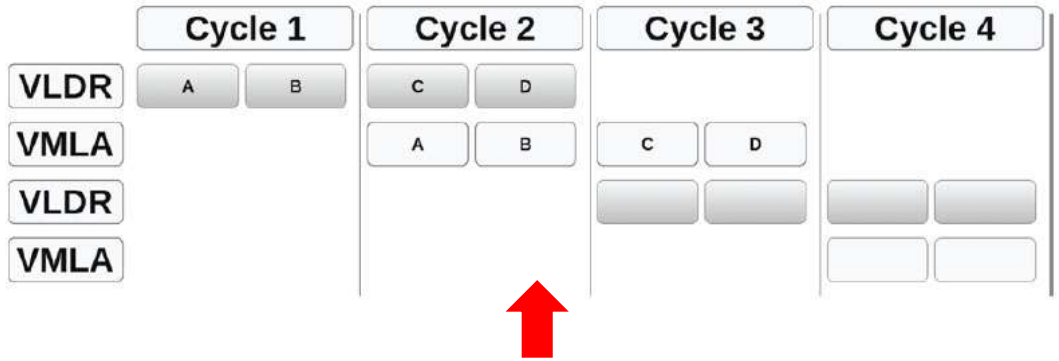
- Helium provides a SIMD capability for Cortex-M CPUs: a set of 128-bit registers are provided which can be used to hold, e.g. 16 separate 8-bit values. A single instruction can operate on each value independently (with predication)
- Extension of Arm Thumb
- Helium instructions operate on vectors of elements of the same data type: Int/FP
  - Integer elements may be signed or unsigned 8-, 16-, 32-bit, fixed-point saturating (Q7, Q15, Q31)
  - Floating-point elements may be single (32-bit) or half precision (16-bit).
- The position of an element in a vector is called *lane*



ARM MVE Vector Execution Model

- MVE permits instruction execution to be interleaved
  - Multiple instructions may overlap in the pipeline execute stage. For example, a Vector Load (VLDR) instruction which reads multiple words from memory into a vector register may execute at the same time as a Vector Multiply (VMUL) instruction which uses that data
- It is up to the CPU hardware designer to decide how many “beats” are executed on each clock cycle (eg. 32-bit datapath vs 64-bit datapath)



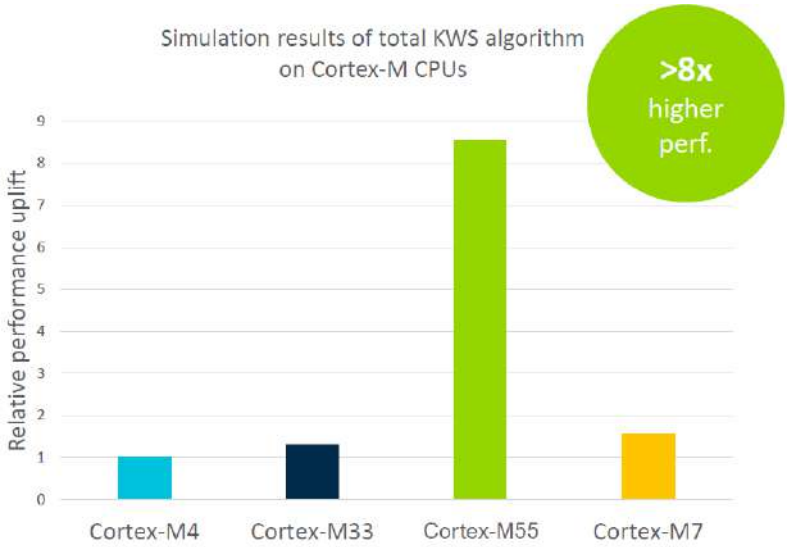


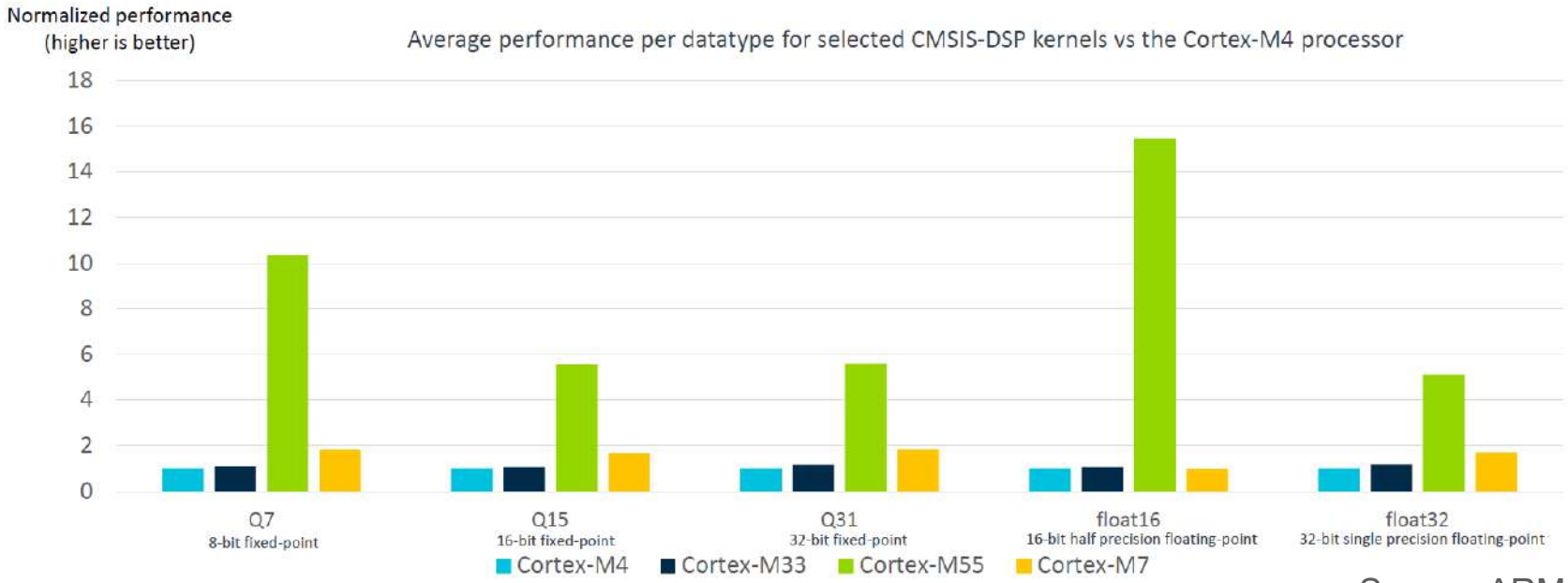
- Complicates exception handling

D of the VLDR happens after beat A of the VMLA has completed. If memory for beat D triggers a fault, the processor needs to remember that the following instruction was part executed (storing a value which shows which beats have already been executed). If fter exception handling, the program returns to this location, the hardware already knows which beats should not be re-executed

Cortex-M55 Performance

- Performance relative to Cortex-M4
- Major improvements for Q7, FP16 (new datatypes in HW)
- ML benchmark (KWS): MFCC, DNN (2 conv, 3 FC layers), 8-bit (w, act), 80-500KB, accuracy 90%-95%

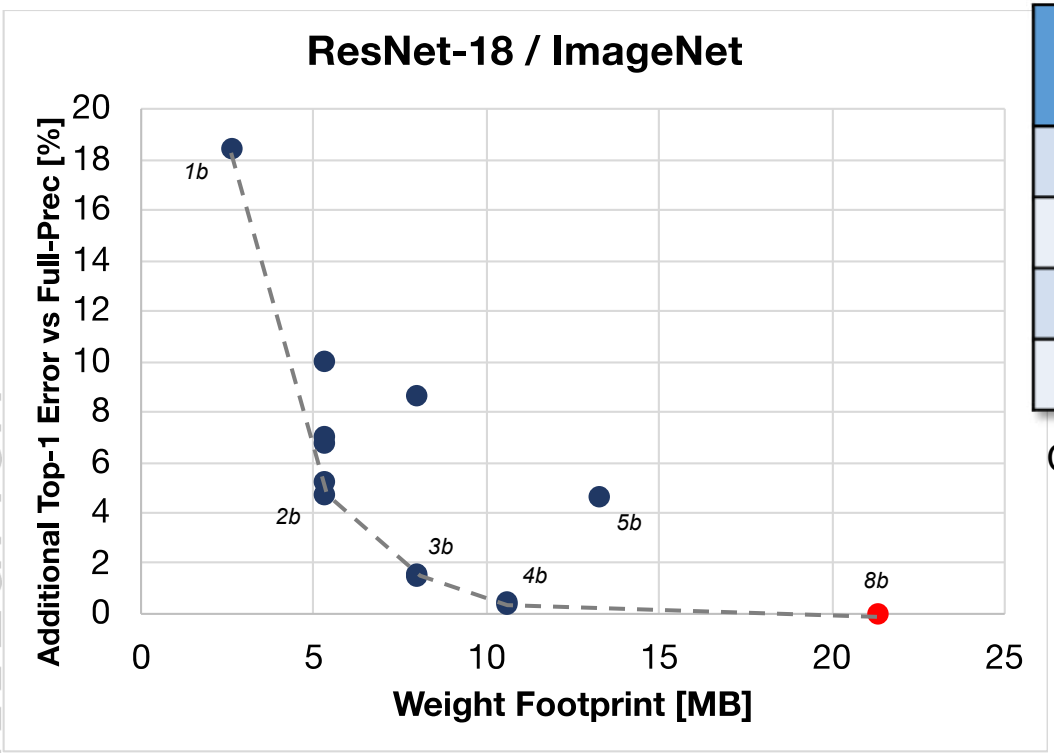




Source: ARM at HC20

Next: Sub-byte precision

SoA Quantization Results



Quantizazion of a MobilenetV1_224_1.0 (*)

Quantization Method	Top1 Accuracy		Weight Memory Footprint	
Full-Precision	70.9%		16.27 MB	
INT-8	70.1%	0.8%	4.06 MB	4x
INT-4	66.46%	4.4%	2.35 MB	7x
Mixed-Precision	68%	2.9%	2.09 MB	8x

Courtesy of Rusci M. «Example on MobilenetV1_224_1.0.»C

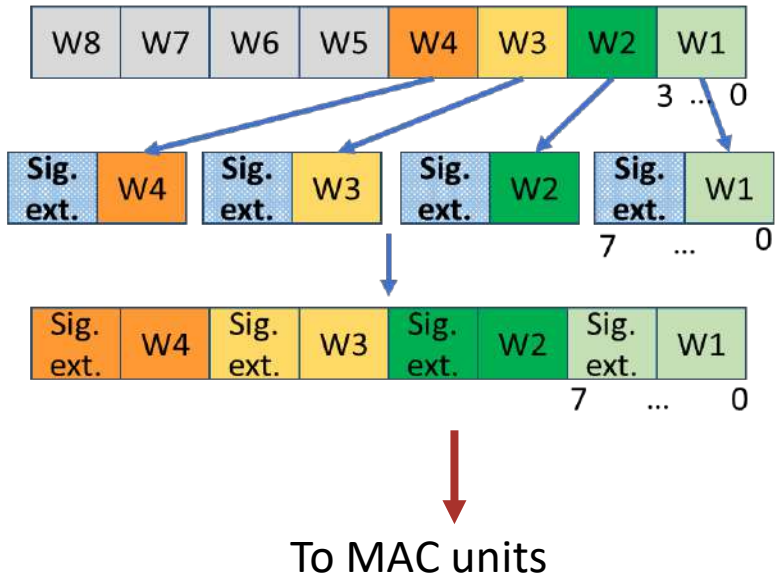
Mixed-precision approach key to meet the memory constraints of tiny devices

Quantized Neural Networks (QNNs) are a natural target for execution on constrained extreme edge platforms.

(*) Rusci M. et al., Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers. . arXiv preprint arXiv:1905.13082.

Sub-byte operands manipulation

32-bit data load with post increment (one cycle)



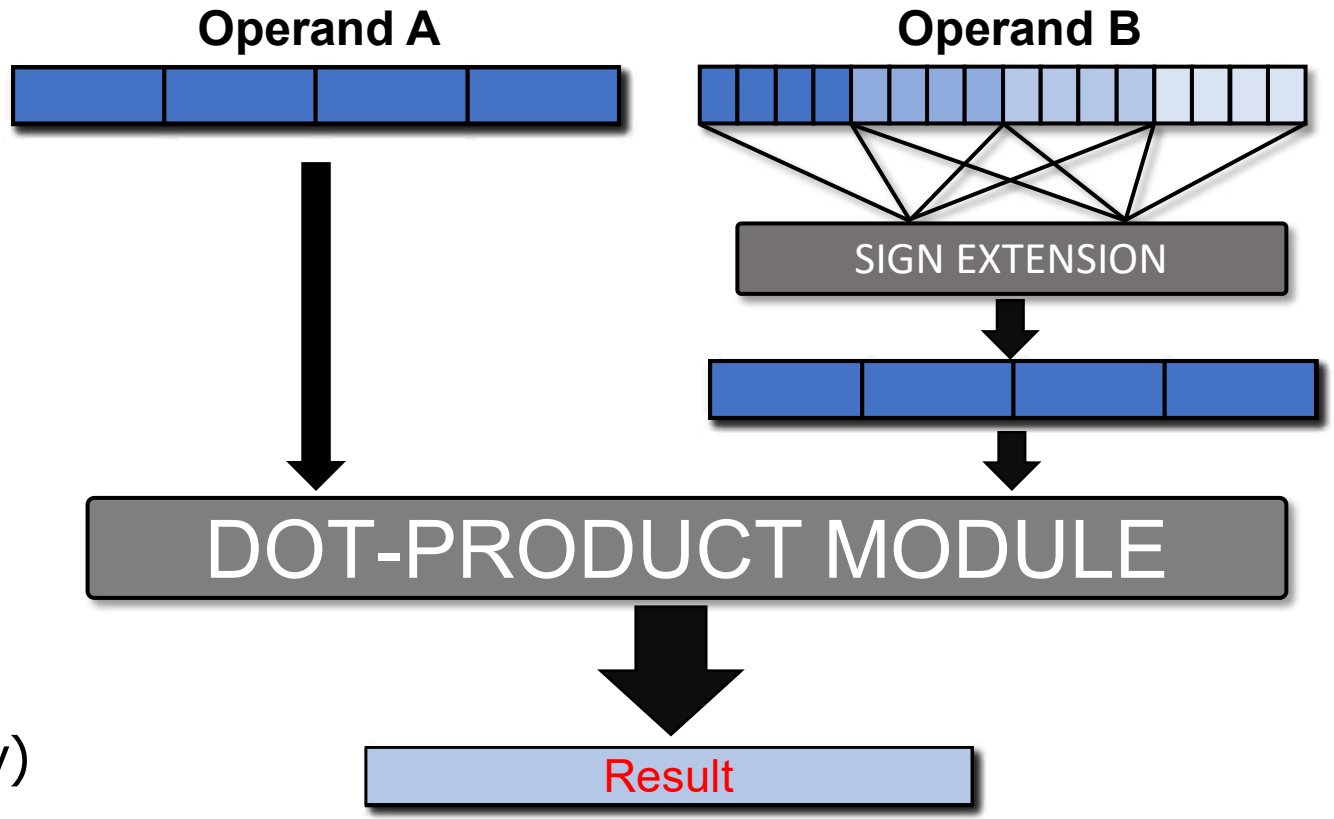
Disassembled Pseudocode

```
p.lw  Src, 0(a0);    // vectorial load
p.bextract  w1e, Src, 4, 0;  //bextract built-in
p.bextract  w2e, Src, 4, 4;
p.bextract  w3e, Src, 4, 8;
p.bextract  w4e, Src, 4, 12;
p.packhi.b  Res, w3e, w4e;   //pack built-in
p.packlo.b  Res, w1e, w2e;   //(two assembly insns)
```

Mixed Precision SIMD Processor

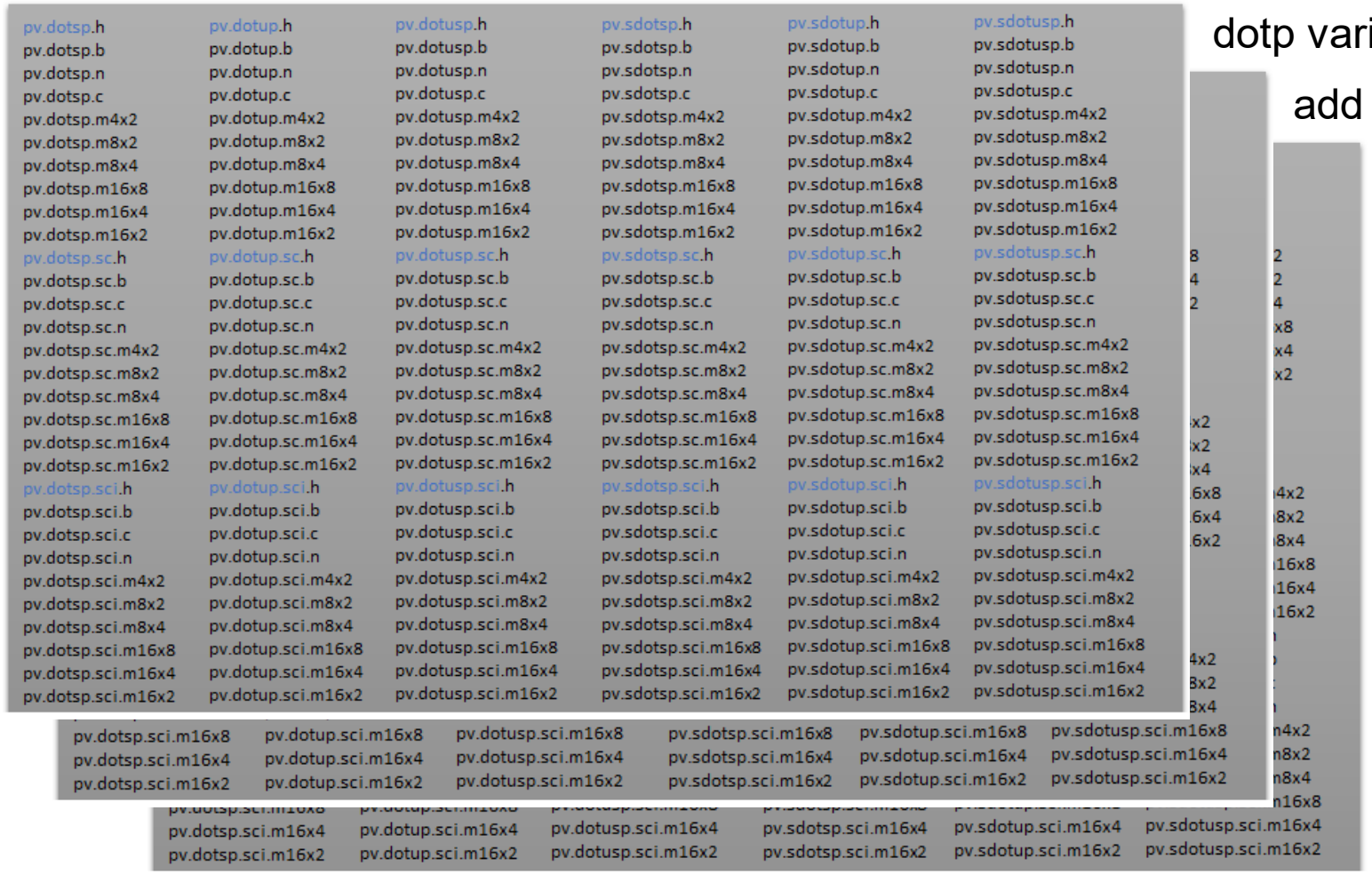
- Can support all variants:
  - 16x16, 16x8, 16x4, 16x2
  - 8x8, 8x4, 8x2
  - 4x4, 4x2
  - 2x2

- Avoids Pack/unpack Overheads
- Maximized performance (SIMD)
- Maximizes RF use (Data Locality)



How to encode all these instructions?

Mixed-Precision Core: New Formats Required



dotp variants

add variants

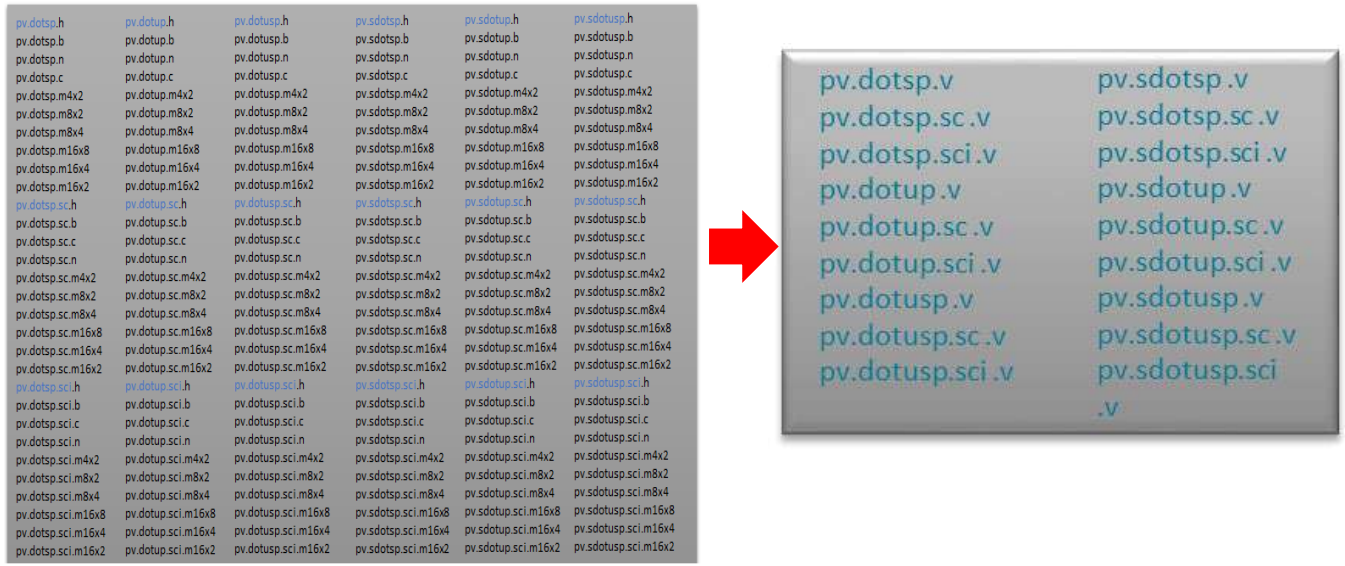
sub variants
avg variants
shift variants
max variants
min variants
abs variants
…

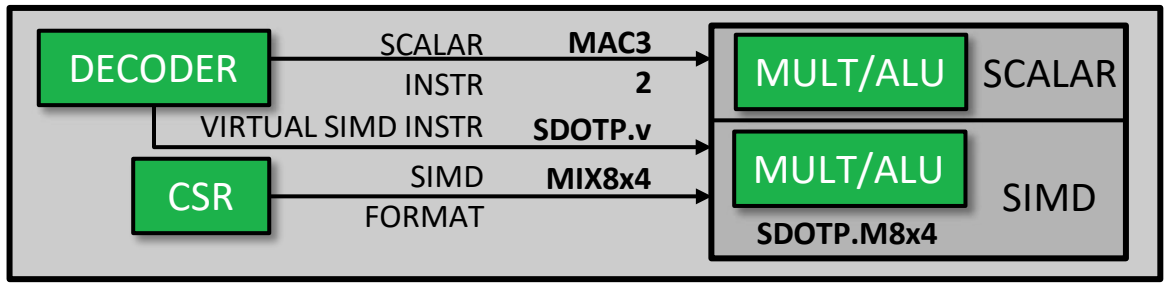
> 500 instructions

Virtual SIMD Instructions

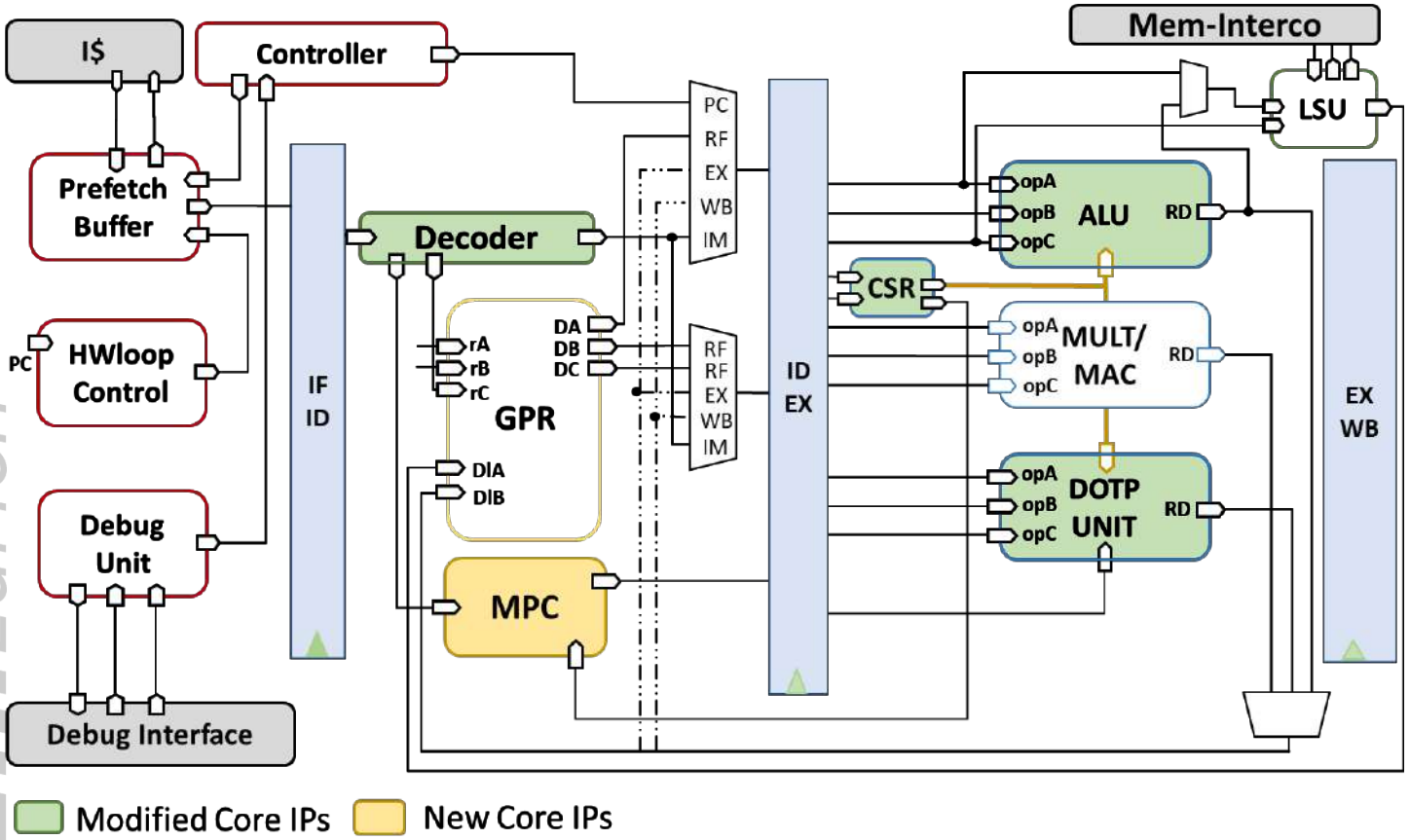
- Encode operation as a virtual SIMD in the ISA (e.g. `sdotsp.v`)
- Format specified at runtime by a Control Register (e.g. 4x4)
- 180→18 Instructions needed for SIMD DOTP
- Potential to avoid code replication for different formats
- Tiny Overhead on QNN for Switching format

  Format switch not frequent in DNN, e.g. every layer.





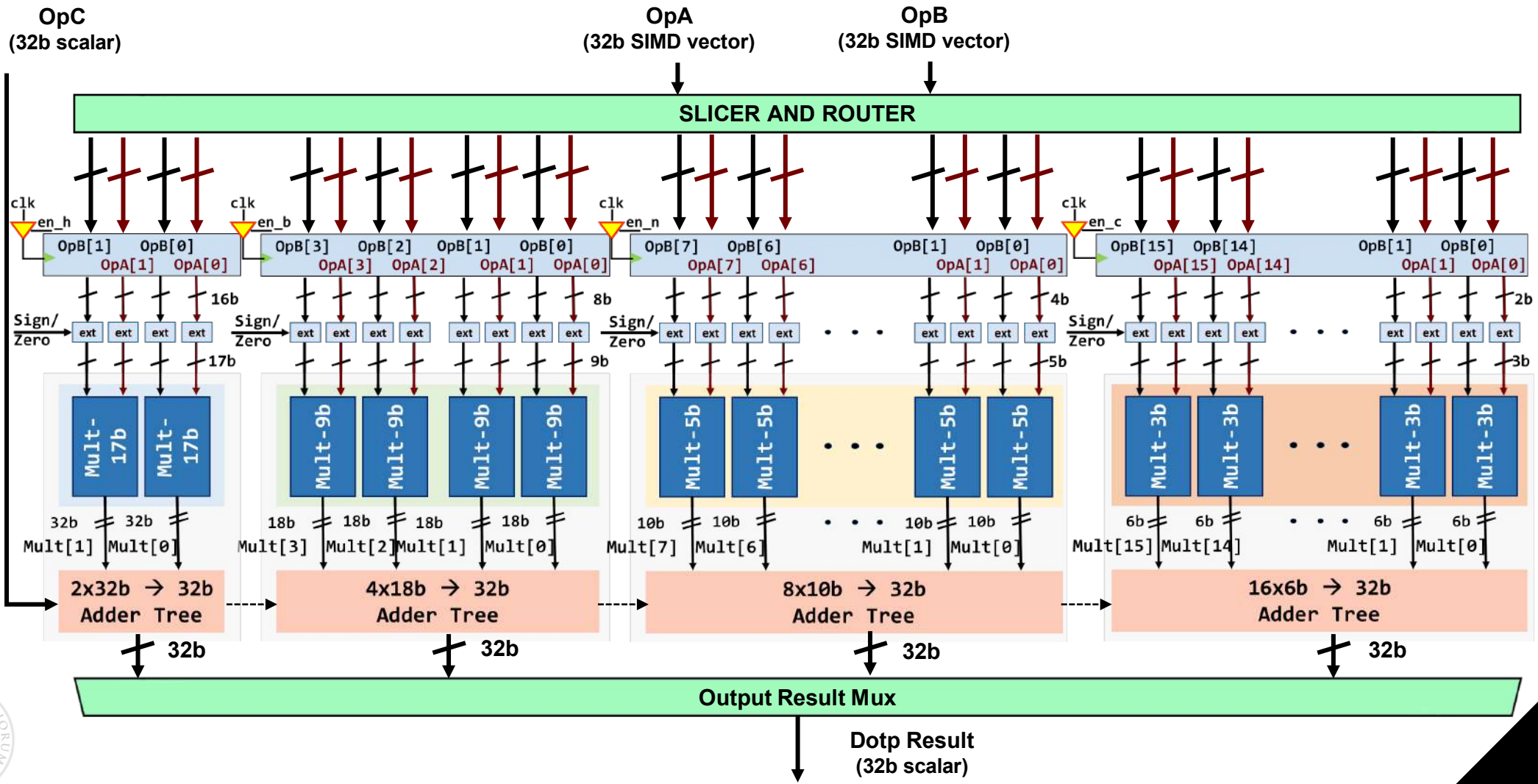
Processor HW extension



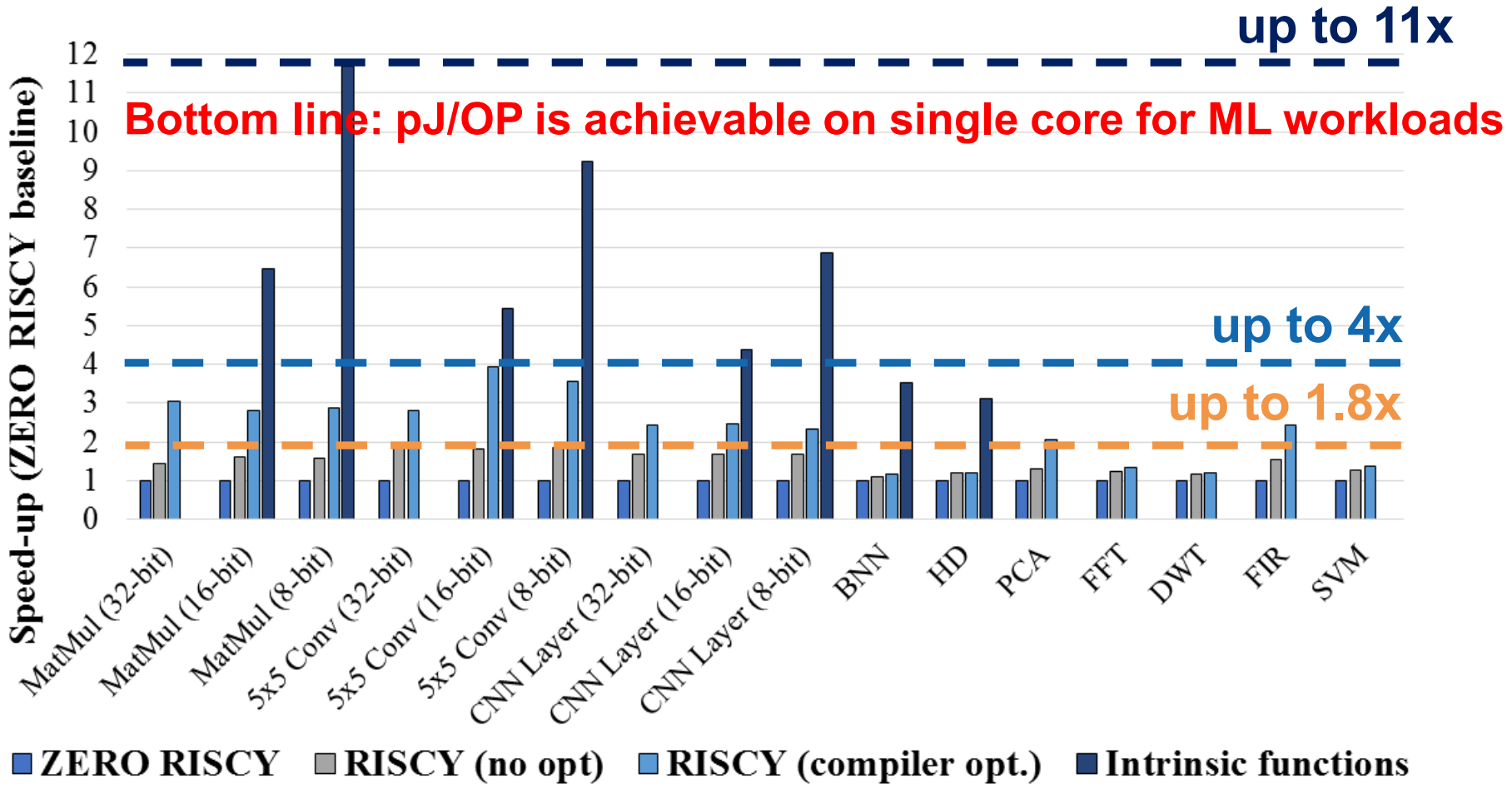
- **Goal**
  - HW support for mixed-precision SIMD instructions;
- **Challenge**
  - Enormous number of instructions to be encoded in the ISA;
- **Solution**
  - Status-based execution.

Extended Dot-Product Unit

Multi-Precision Integer Dotp-Unit



Xpulp Extensions Performance (Single Core)



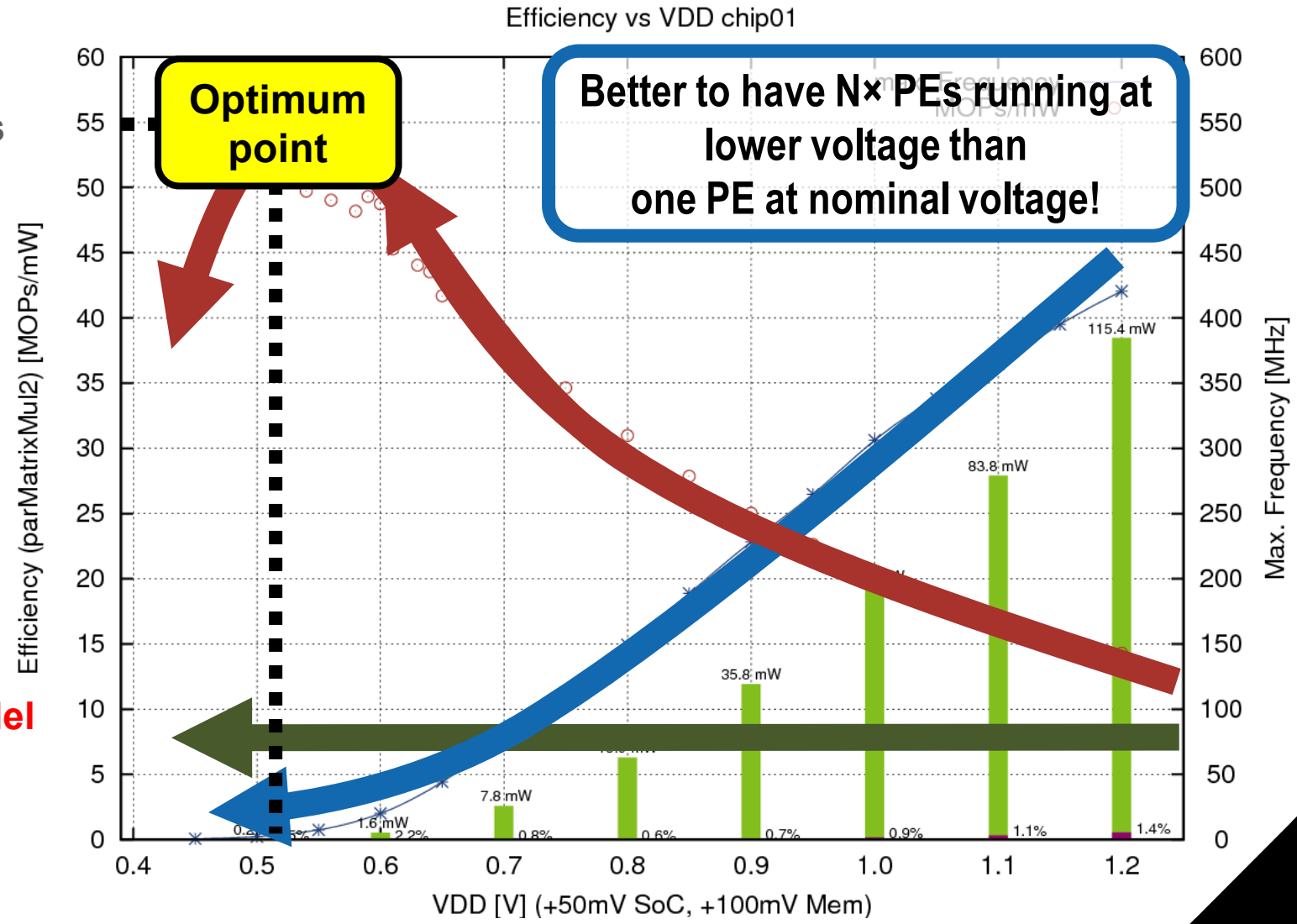
Nice – But what about the GOPS? M7: 5.01 CoreMark/MHz-58.5 µW/MHz

Faster+Superscalar is not efficient! M4: 3.42 CoreMark/MHz-12.26 µW/MHz

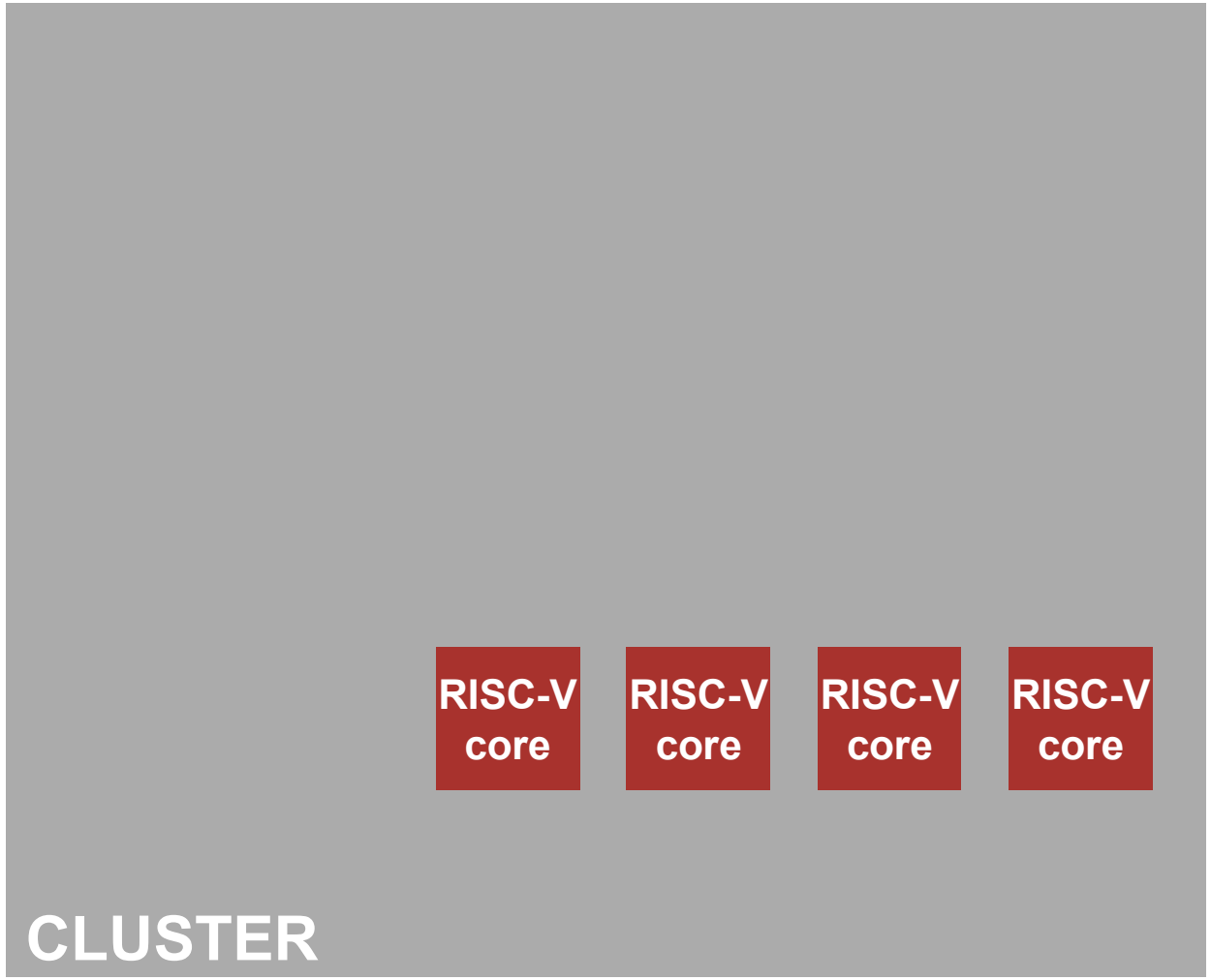
ML & Parallel Near Thredhold → PULP

- 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

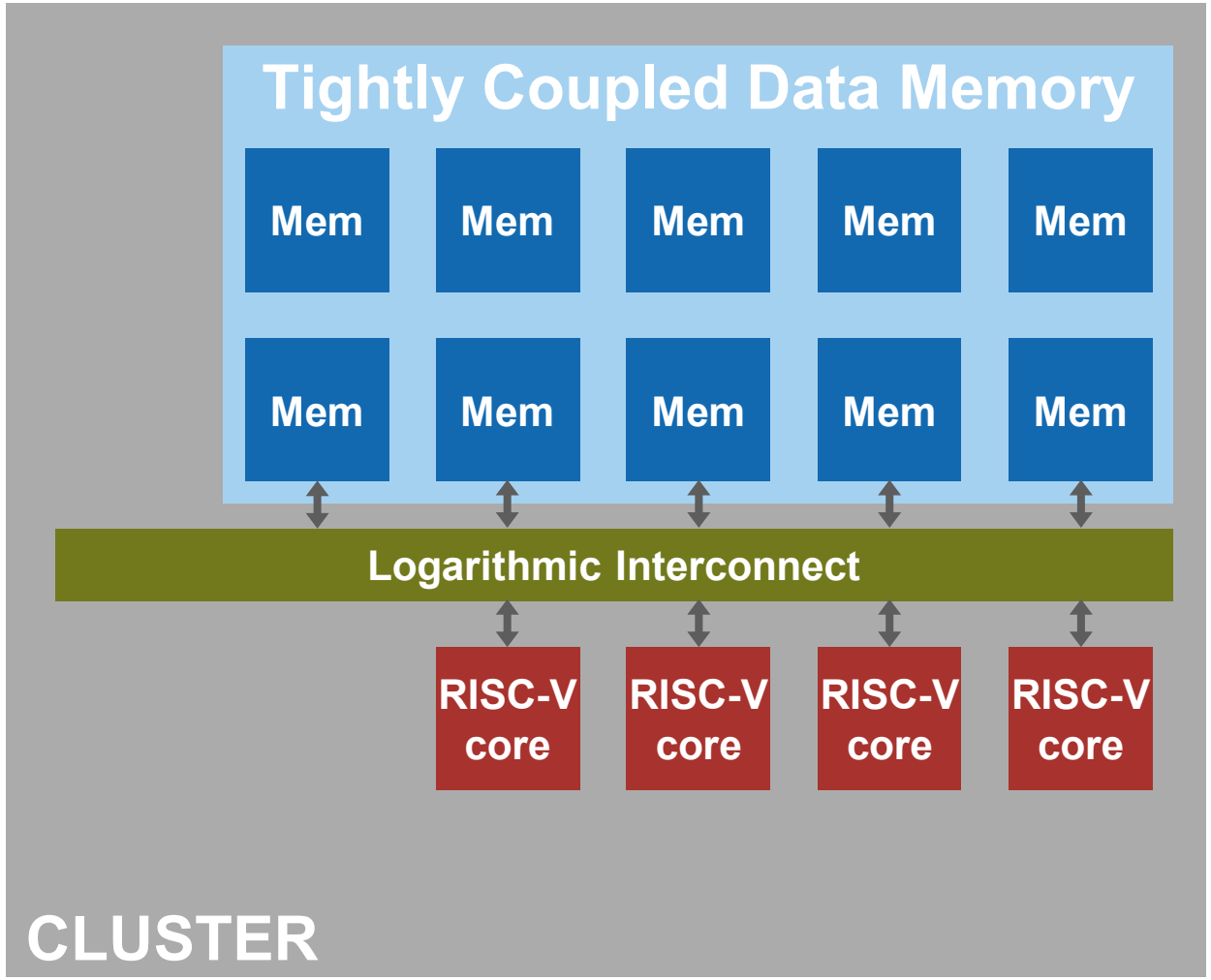
ML is massively parallel and scales well (P/S ↑ with NN size)



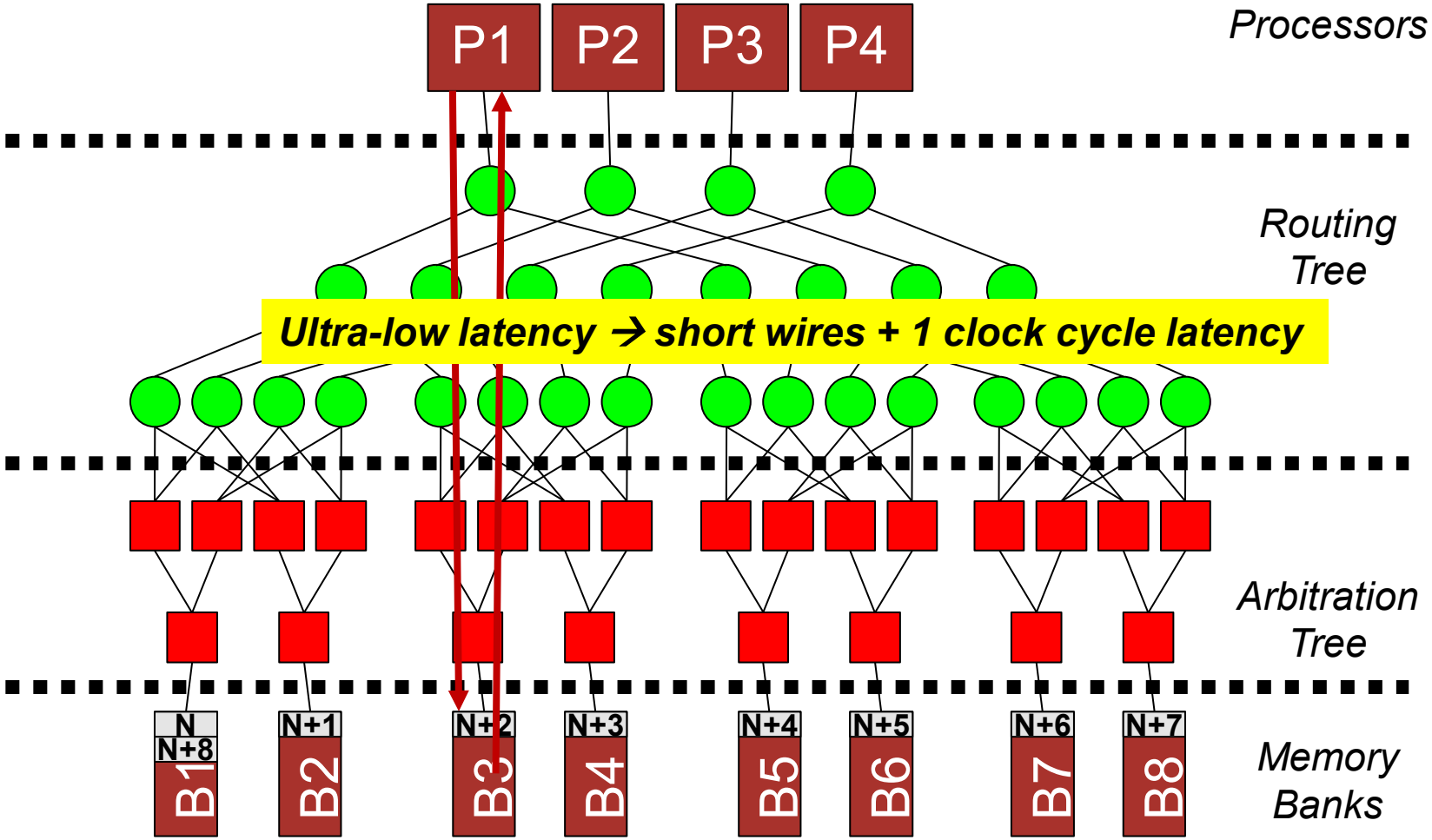
Multiple RI5CY Cores (1-16)



Low-Latency Shared TCDM

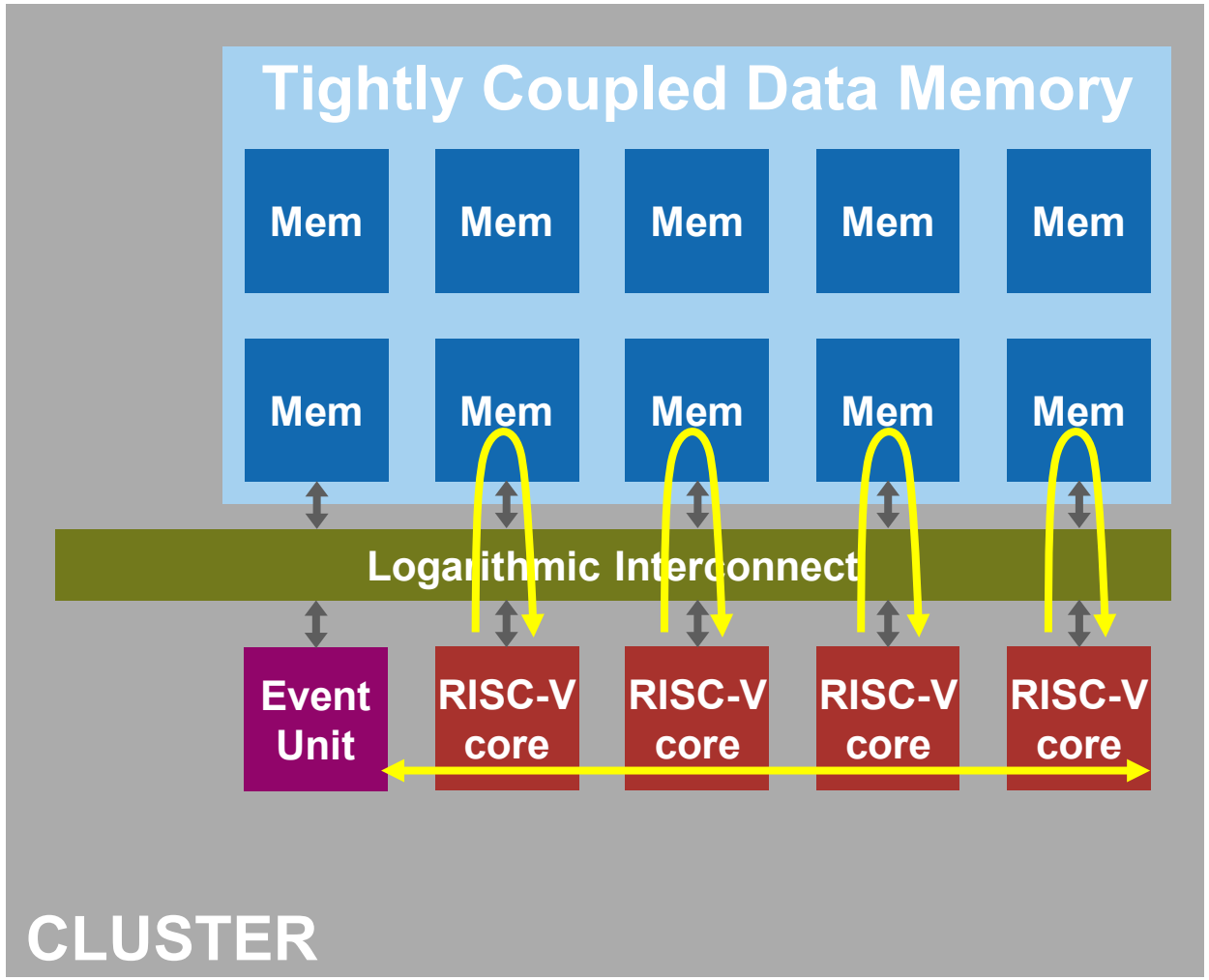


High speed single clock logarithmic interconnect



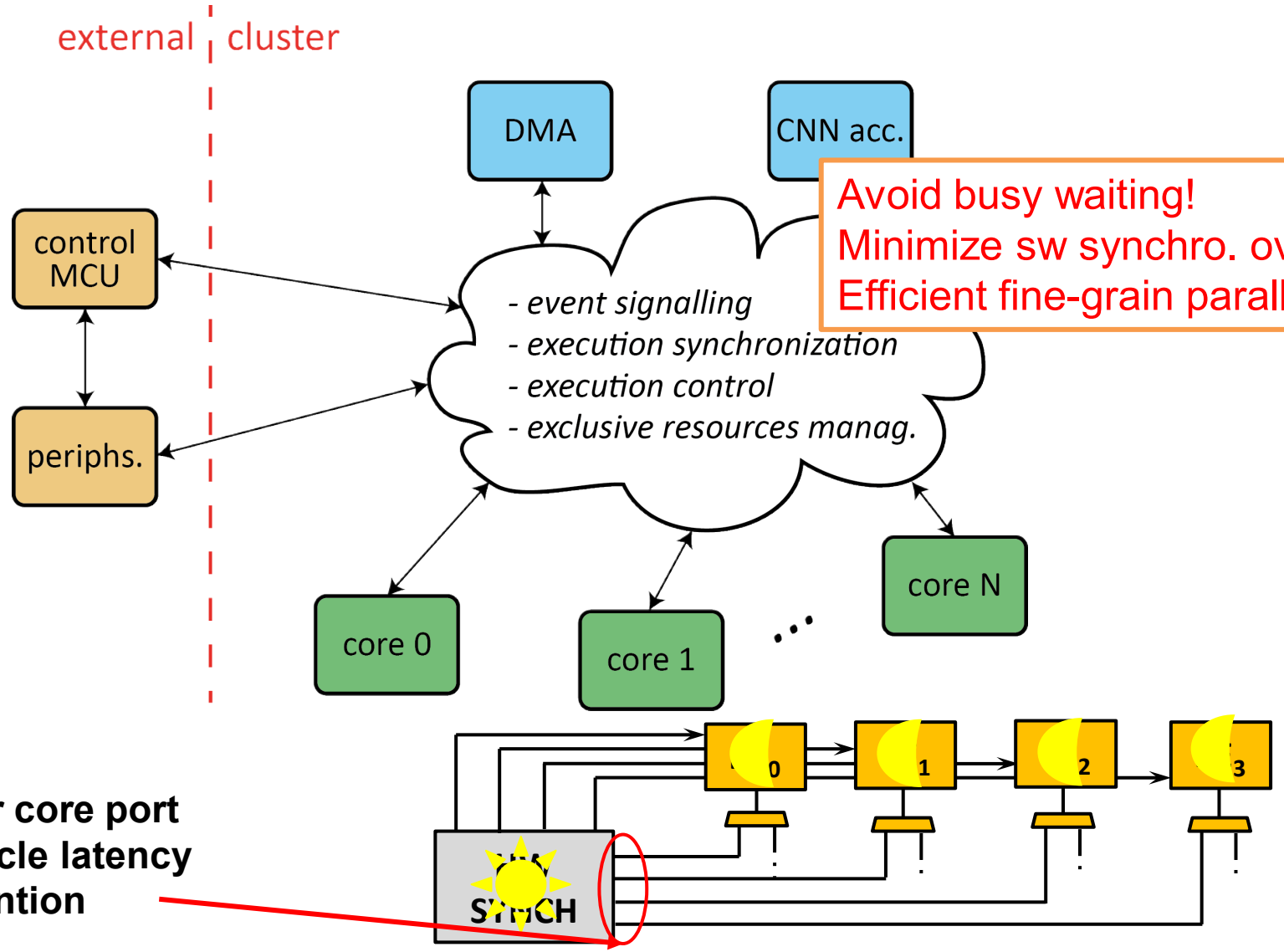
A. Rahimi, I. Loi, M. R. Kakoee and L. Benini, "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters," *2011 Design, Automation & Test in Europe*, 2011, pp. 1-6.

Fast synchronization and Atomics



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.

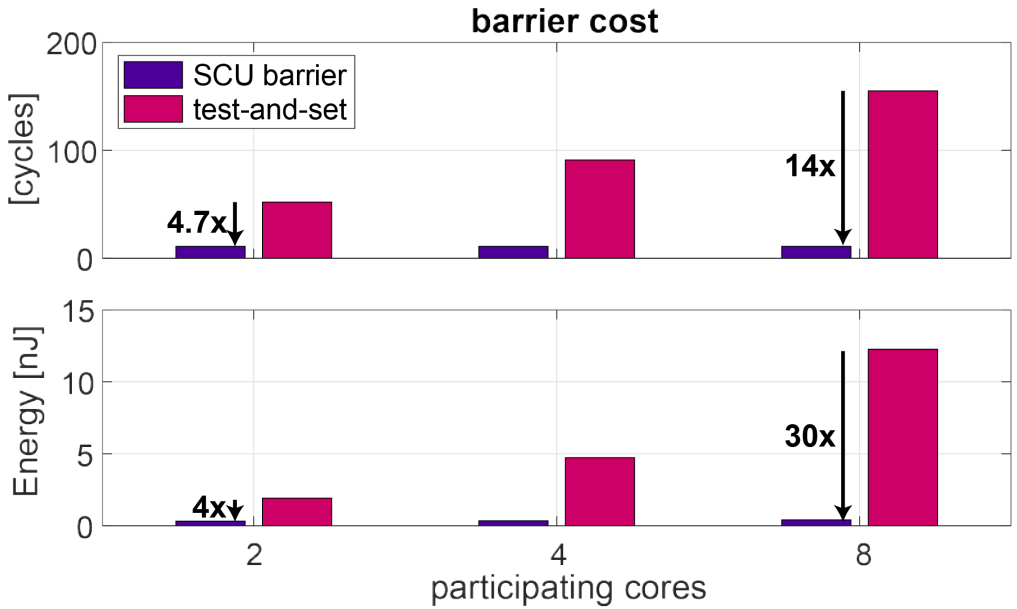
Synchronization & Events

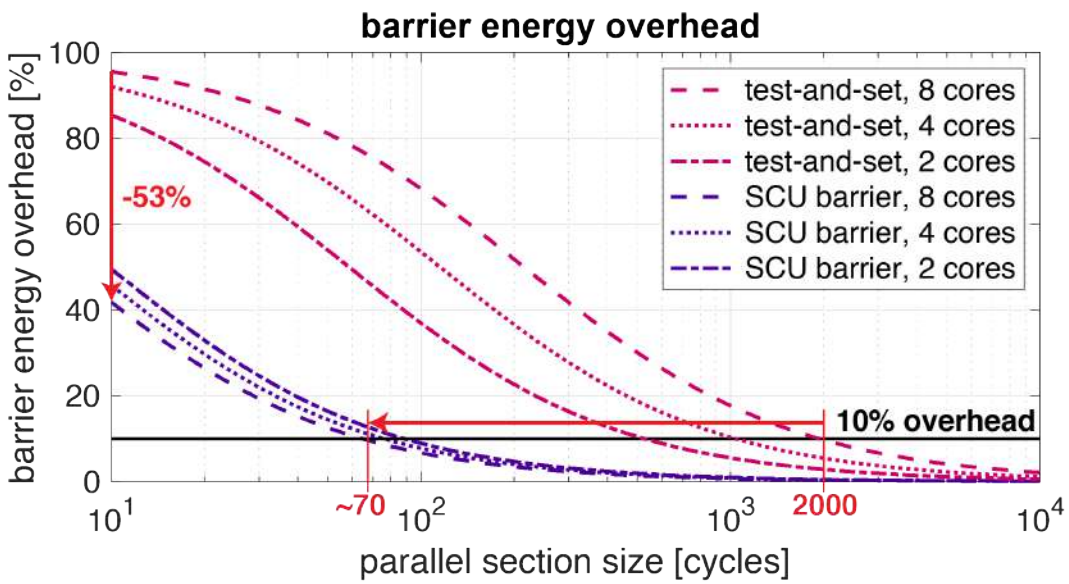


Avoid busy waiting!
Minimize sw synchro. overhead
Efficient fine-grain parallelization

Private, per core port
→single cycle latency
→no contention

Results: Barrier



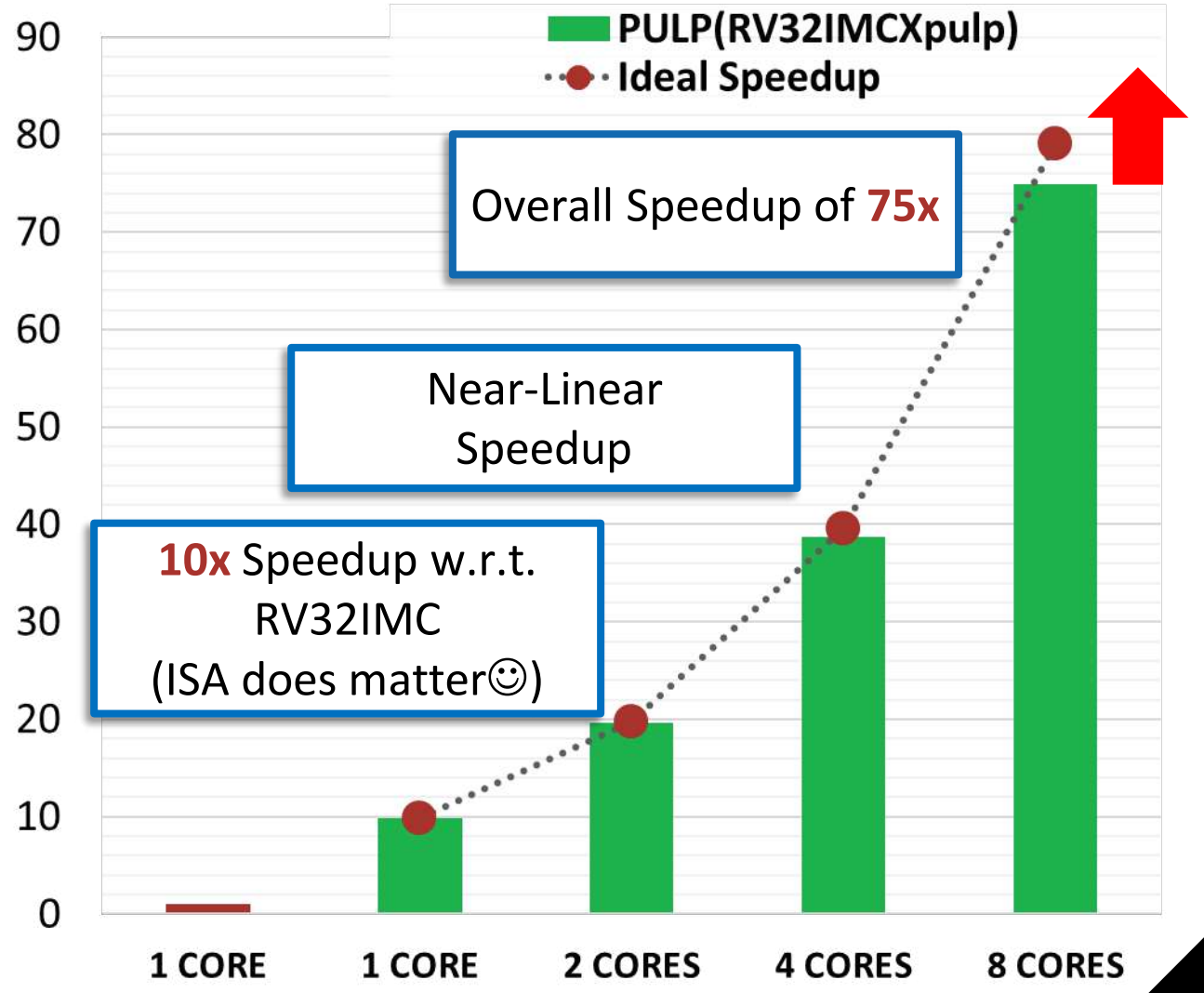


- Fully parallel access to SCU: Barrier cost constant
- Primitive energy cost: Down by up to 30x
- Minimum parallel section for 10% overhead in terms of …
  - … cycles: ~100 instead of > 1000 cycles
  - … energy: ~70 instead of > 2000 cycles

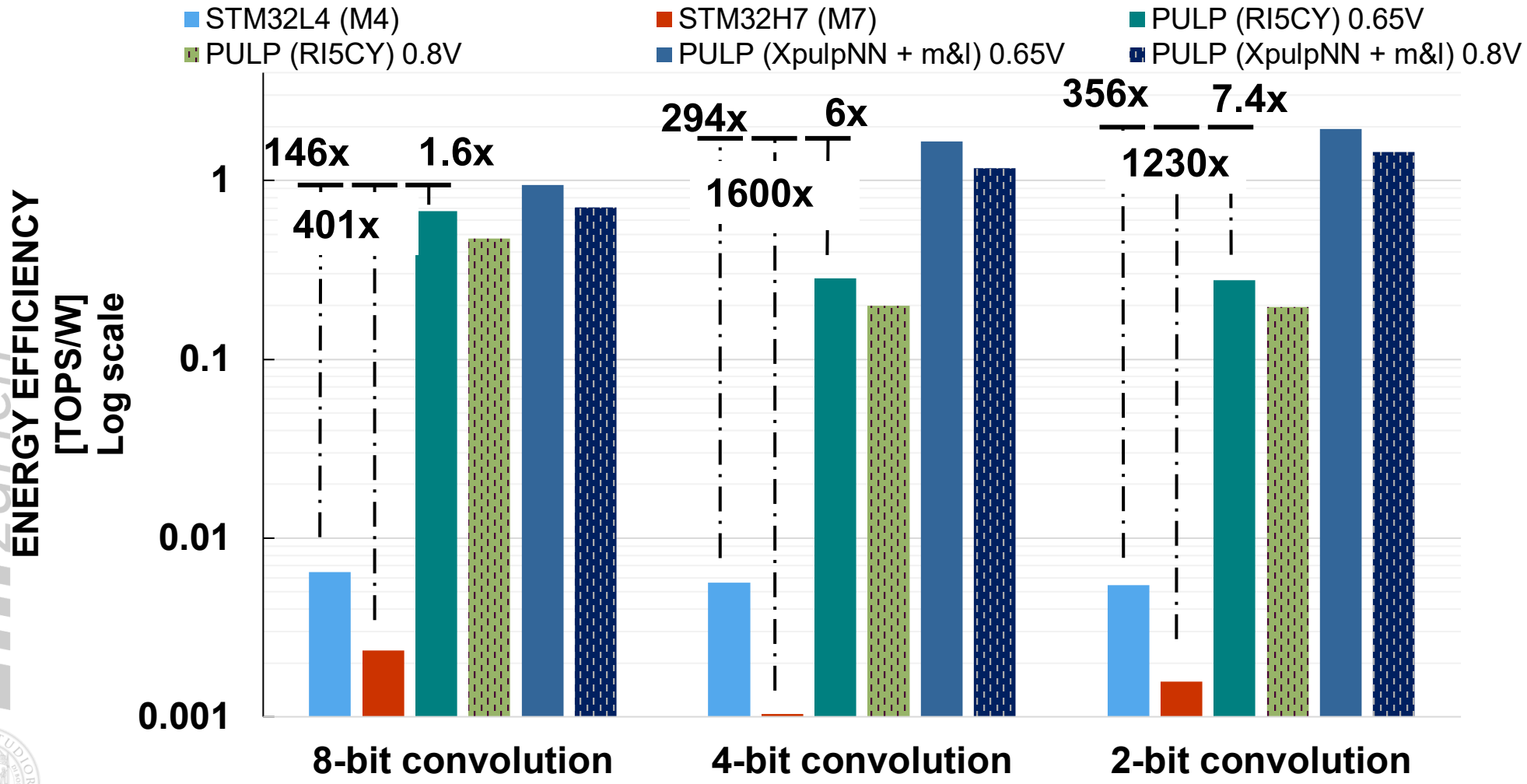
PULP for ML (DNNs) Speedup

[Garofalo et al. Philos. Trans. R. Soc 20]

- **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**

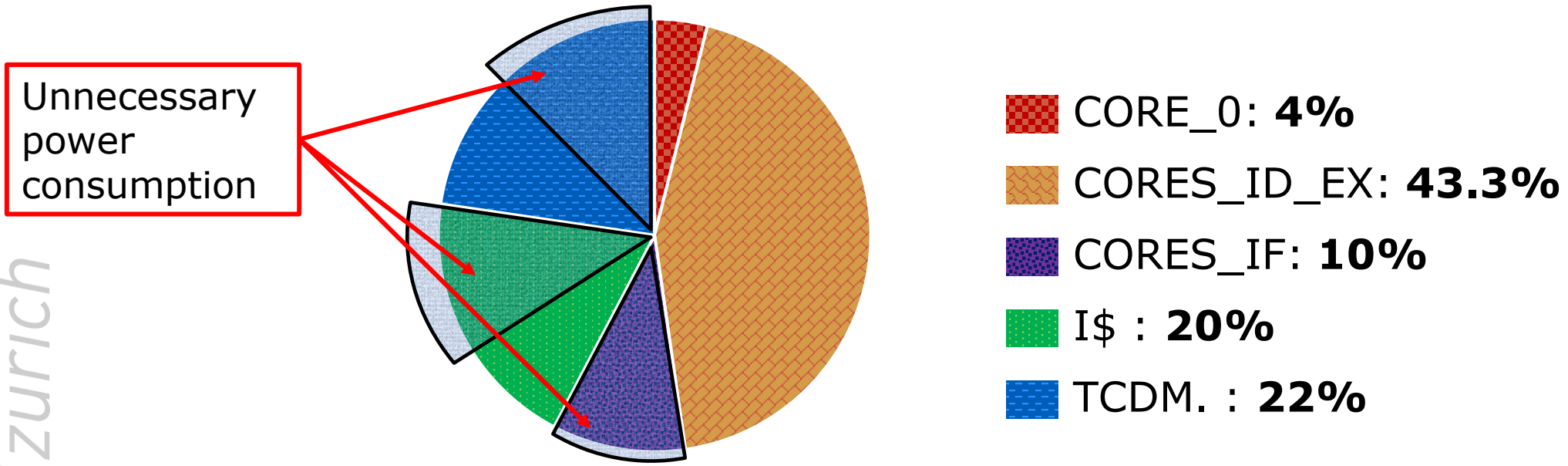


8-Cores Cluster + XpulpNN + M&L (22nm)



Addressing Multicore Inefficiencies

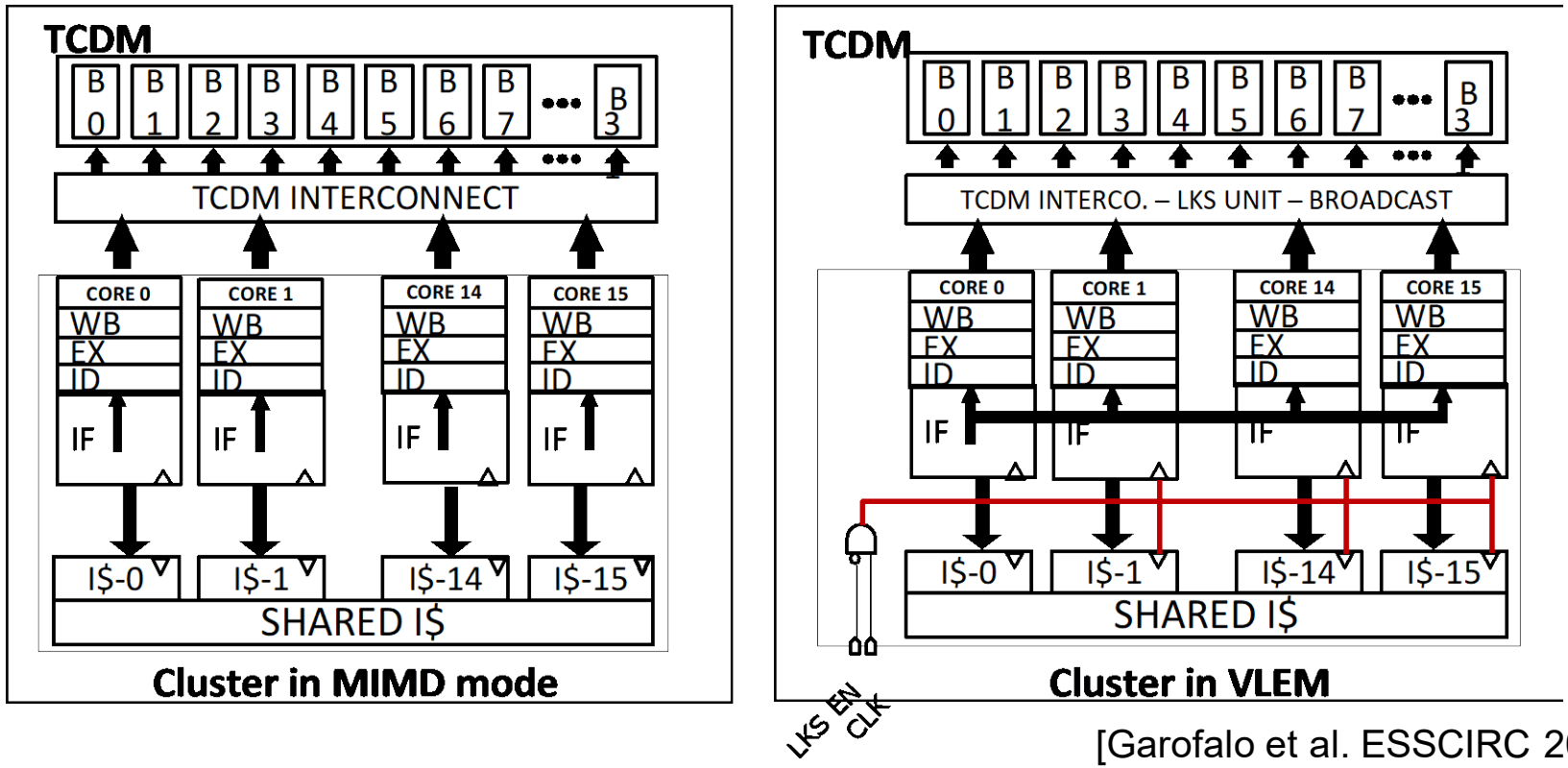
Power analysis of a parallel 8-bit x 4-bit convolution



- Reduce unnecessary power consumption (not spent in computation)
- Exploit convolution's instruction and memory data access pattern regularity
- Increase energy efficiency at low extra-area cost

→ reconfigurable MIMD/SIMD architecture

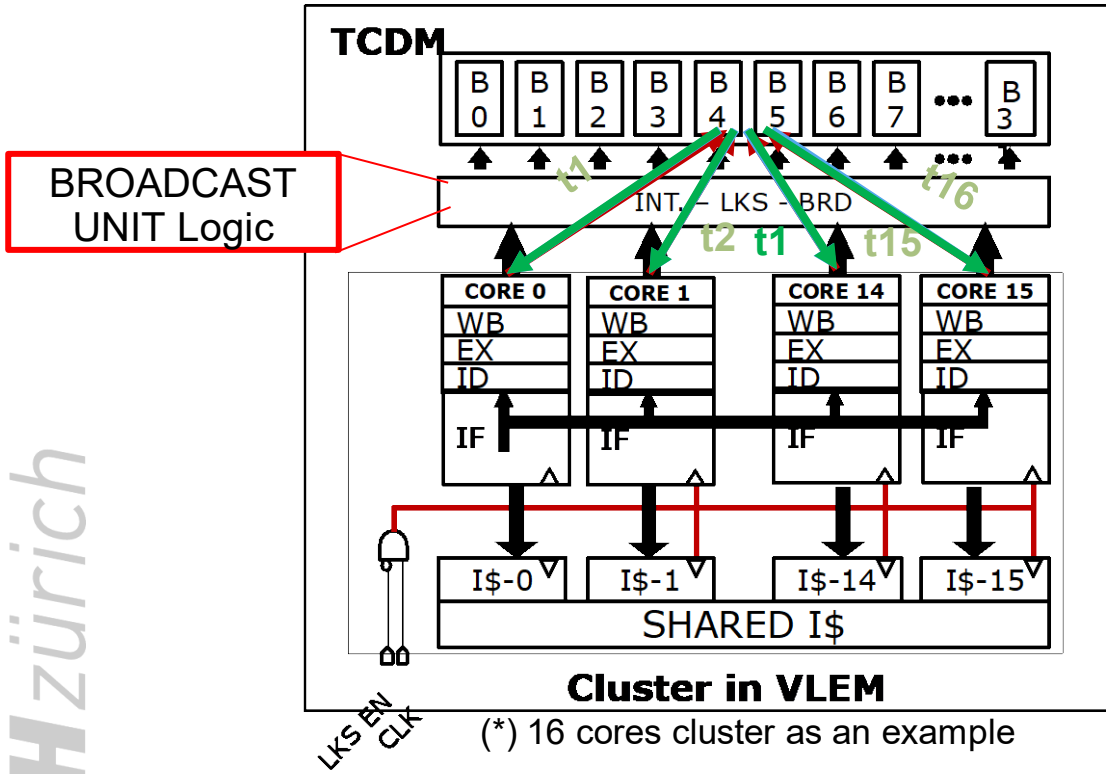
The Power of SIMD



[Garofalo et al. ESSCIRC 2021]

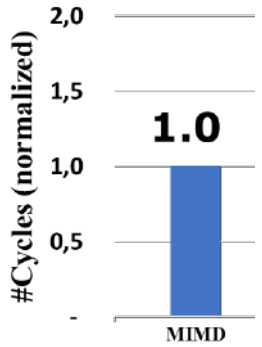
- Cores enter in SIMD (VLEM) mode when executing regular kernels (in two clock cycles)
- In SIMD, instruction flow orchestrated only by the MAIN core → Less energy
- Cores resume in MIMD mode on divergent branches (..or control tasks) → Flexibility

Broadcasting Share Data



(*) 16 cores cluster as an example

Convolution exec. kernels

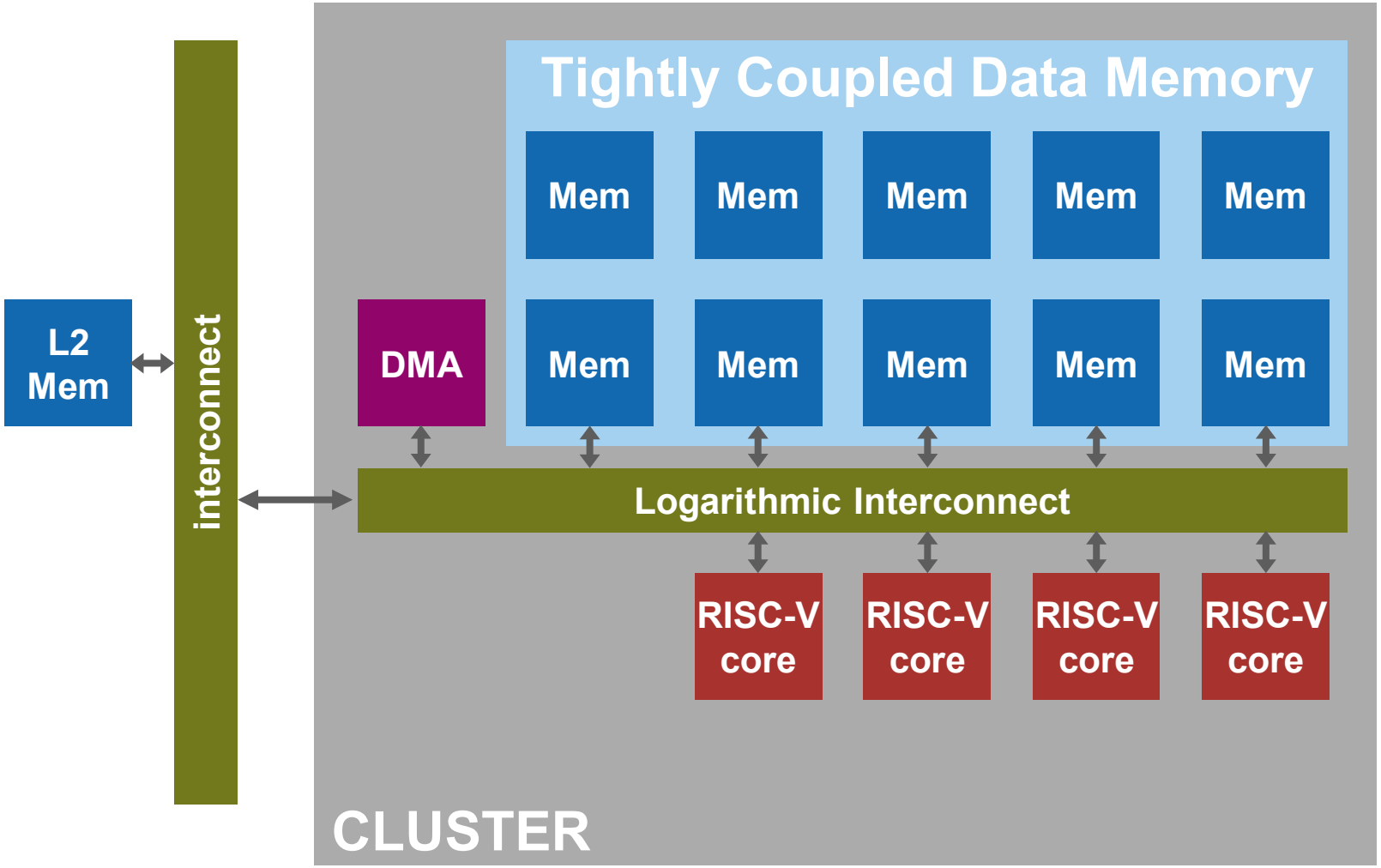


~44% energy saving w.r.t. MIMD mode

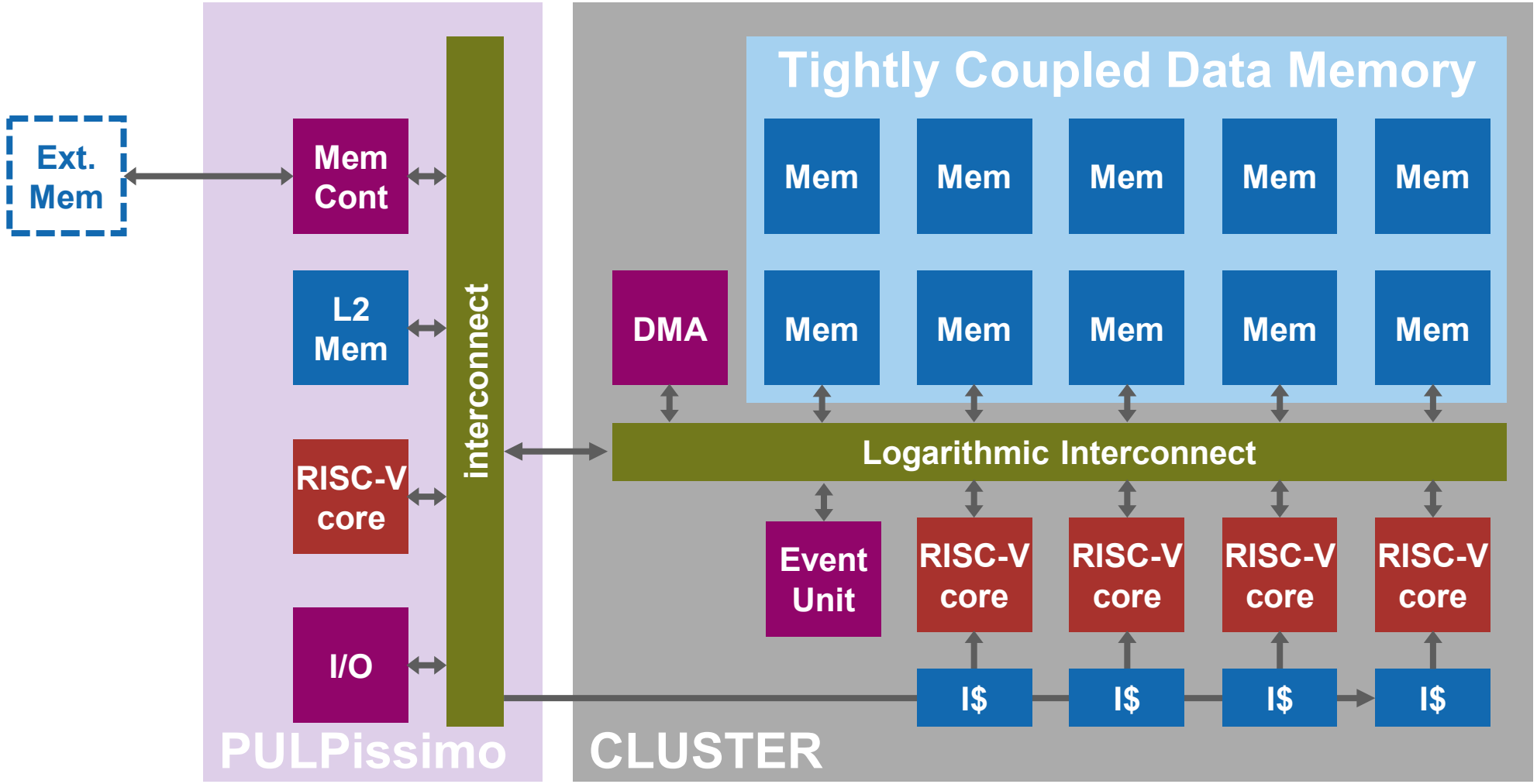
<10% area overhead w.r.t. MIMD only cluster

- 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

Data memory Hierarchy: DMA-based, SW managed



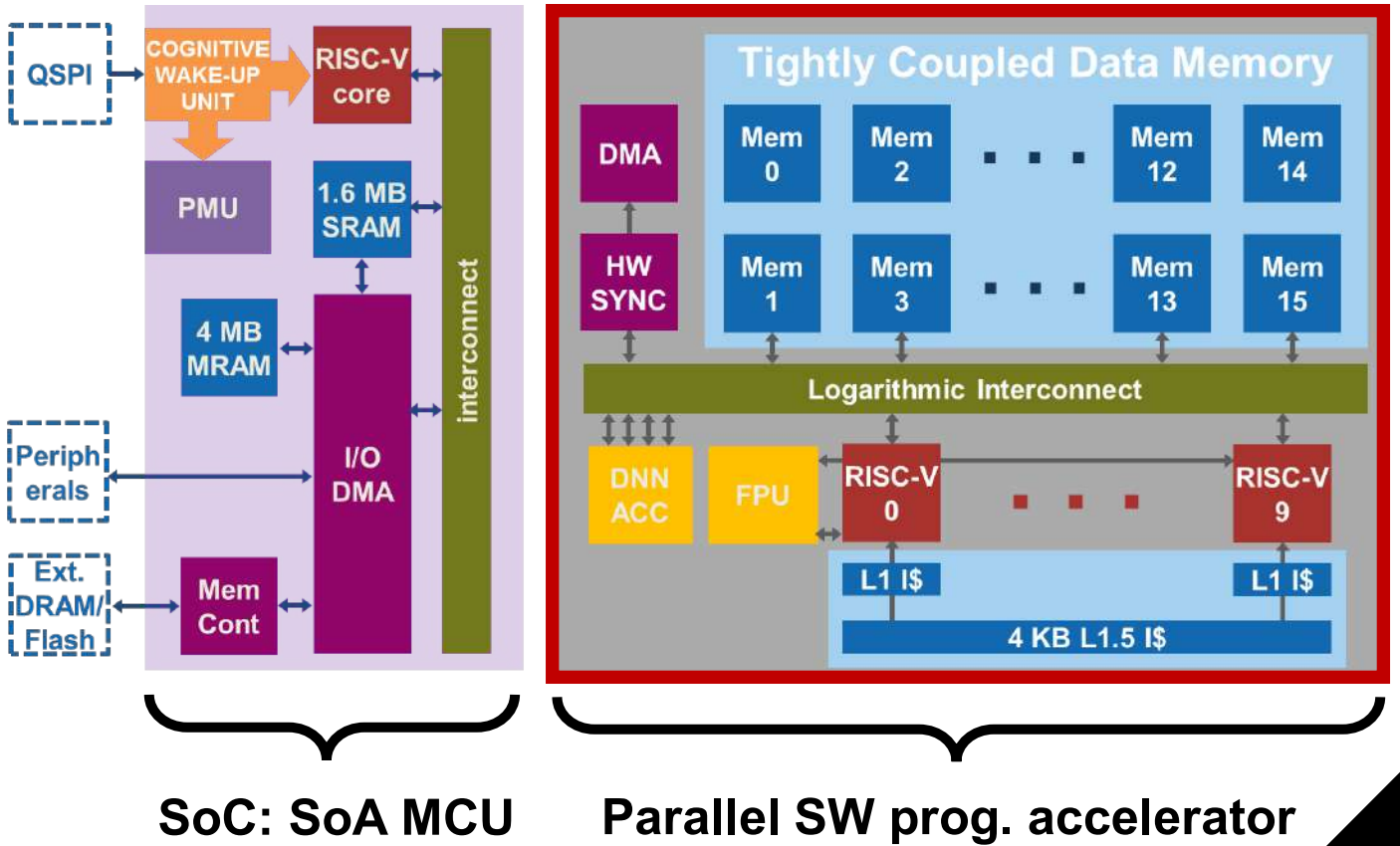
An additional I/O controller for IO, off-chip Memory



All together in VEGA: Extreme Edge IoT Processor

In cooperation with GREENWAVES TECHNOLOGIES

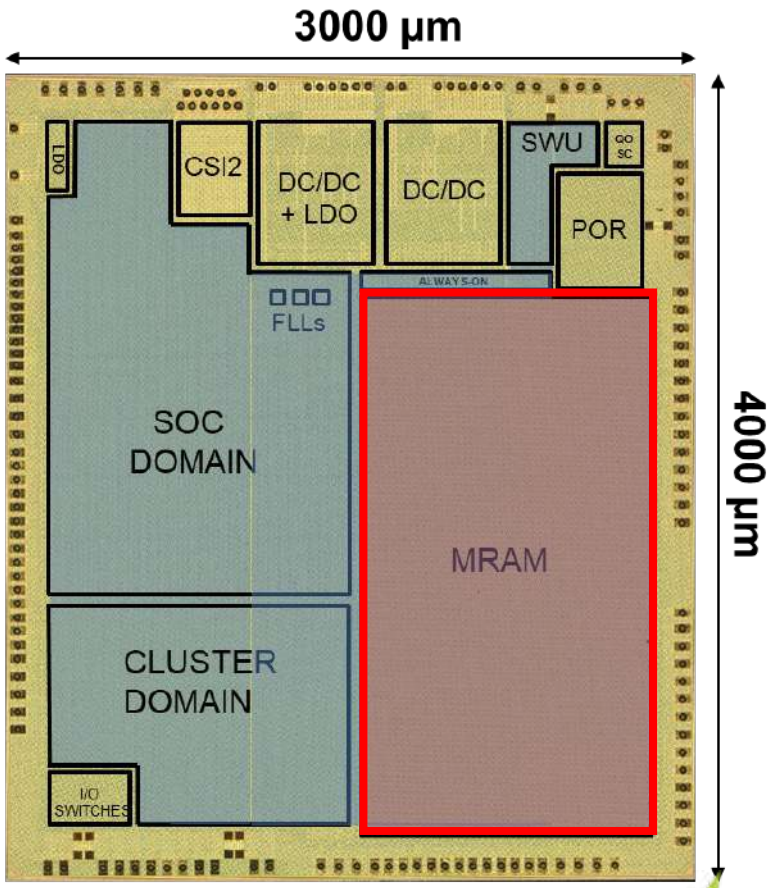
- RISC-V cluster (8cores +1) 614GOPS/W @ 7.6GOPS (8bit DNNs), 79GFLOPS/W @ 1GFLOP (32bit FP appl)
- Multi-precision HWCE(4b/8b/16b) 3×3×3 MACs with normalization / activation: 32.2GOPS and 1.3TOPS/W (8bit)
- 1.7 µW cognitive unit for autonomous wake-up from retentive sleep mode



D. Rossi et al., "Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode," in IEEE Journal of Solid-State Circuits, vol. 57, no. 1, pp. 127-139, Jan. 2022

All together in VEGA: Extreme Edge IoT Processor

- RISC-V cluster (8cores +1) 614GOPS/W @ 7.6GOPS (8bit DNNs), 79GFLOPS/W @ 1GFLOP (32bit FP appl)
- Multi-precision HWCE(4b/8b/16b) 3×3×3 MACs with normalization / activation: 32.2GOPS and 1.3TOPS/W (8bit)
- 1.7 µW cognitive unit for autonomous wake-up from retentive sleep mode
- **Fully-on chip DNN inference with 4MB MRAM**

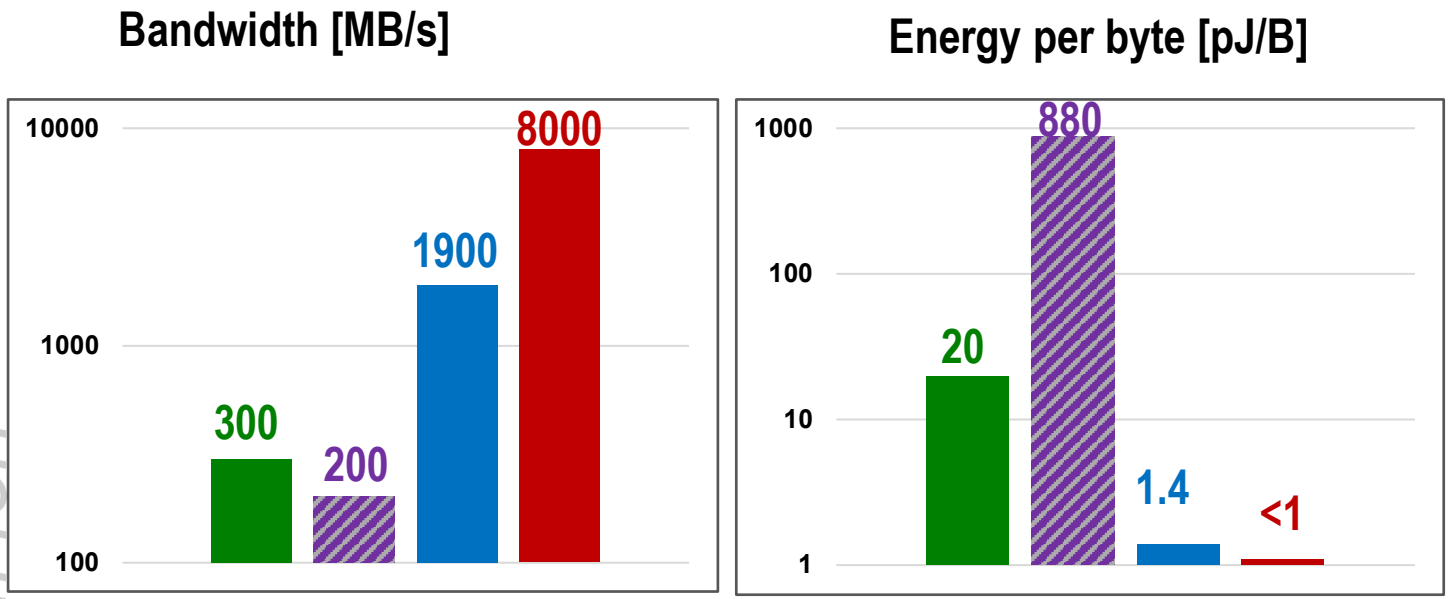


In cooperation with GREENWAVES TECHNOLOGIES

Technology	22nm FDSOI
Chip Area	12mm^2
SRAM	1.7 MB
MRAM	4 MB
VDD range	0.5V - 0.8V
VBB range	0V - 1.1V
Fr. Range	32 kHz - 450 MHz
Pow. Range	1.7 µW - 49.4 mW

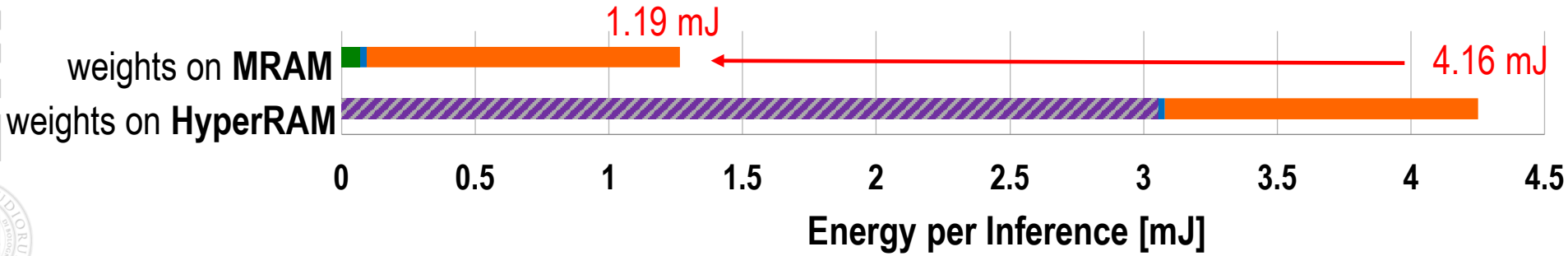
ETH zürich

Full DNN Energy (MobileNetV2)

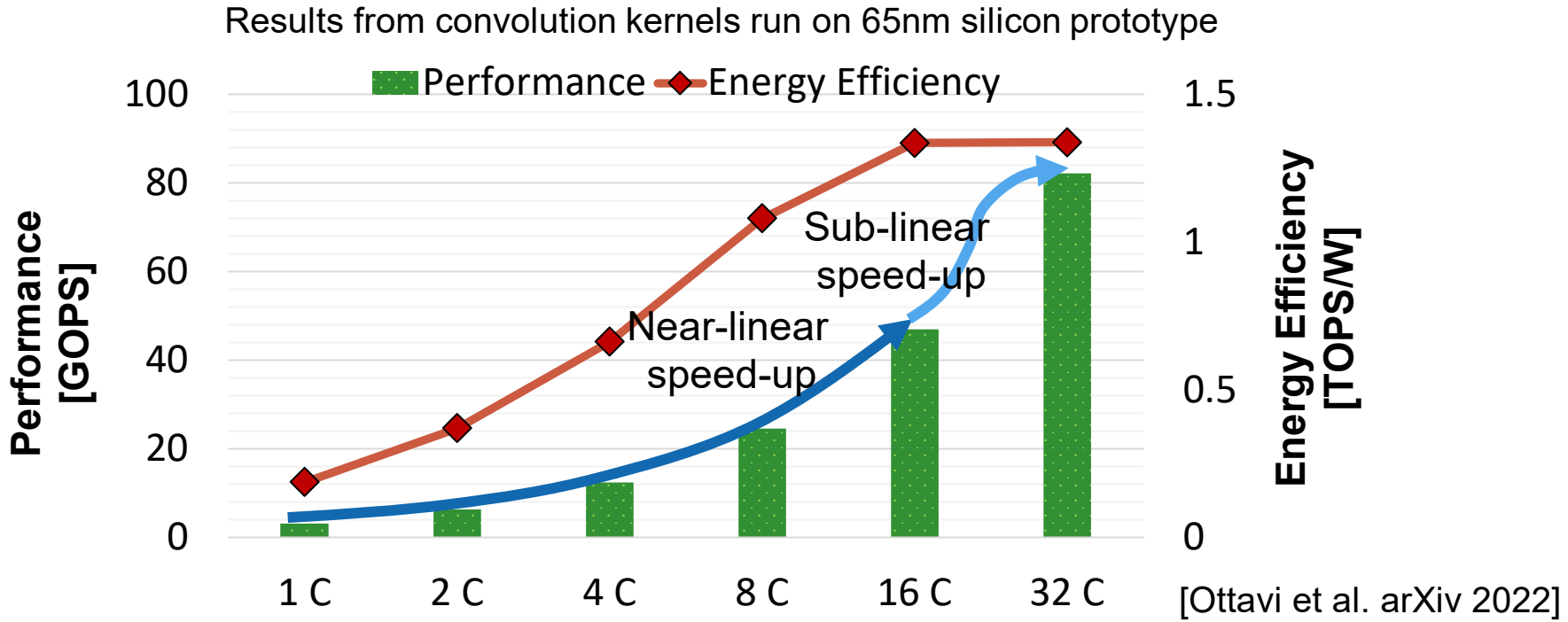


HyperRAM (ext)↔L2 w/ I/O DMA
MRAM↔L2 w/ I/O DMA
L2↔L1 w/ Cluster DMA
L1 access

end-to-end on-chip computation **3.5x less energy**



System Scalability… How many processors?



[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 interco
- **Custom datapaths** to improve throughput and efficiency..

Note: ..but at which cost? See Gianna Paulin's Lecture

PULP

Parallel Ultra Low Power

Luca Benini, Alessandro Capotondi, Alessandro Ottaviano, Alessio Burrello, Alfio Di Mauro, Andrea Borghesi, Andrea Cossettini, Andreas Kurth, Angelo Garofalo, Antonio Pullini, Arpan Prasad, Bjoern Forsberg, Corrado Bonfanti, Cristian Cioflan, Daniele Palossi, Davide Rossi, Fabio Montagna, Florian Glaser, Florian Zaruba, Francesco Conti, Georg Rutishauser, Germain Haugou, Gianna Paulin, Giuseppe Tagliavini, Hanna Müller, Luca Bertaccini, Luca Valente, Manuel Eggimann, Manuele Rusci, Marco Guermandi, Matheus Cavalcante, Matteo Perotti, Matteo Spallanzani, Michael Rogenmoser, Moritz Scherer, Moritz Schneider, Nazareno Bruschi, Nils Wistoff, Pasquale Davide Schiavone, Paul Scheffler, Philipp Mayer, Robert Balas, Samuel Riedel, Segio Mazzola, Sergei Vostrikov, Simone Benatti, Stefan Mach, Thomas Benz, Thorir Ingolfsson, Tim Fischer, Victor Javier Kartsch Morinigo, Vlad Niculescu, Xiaying Wang, Yichao Zhang, Frank K. Gürkaynak,
all our past collaborators and many more that we forgot to mention

http://pulp-platform.org @pulp_platform