ISA Extentions in RISC-V-Based Architectures

Yvan Tortorella

yvan.tortorella@chips.it

EFCL Winter School 2026

Introduction and Motivation

- Emerging application areas for AI-enabled IoT
  - **Personalized Healthcare**

Introduction and Motivation

- Emerging application areas for AI-enabled IoT
  - Personalized Healthcare
  - **Augmented Reality**

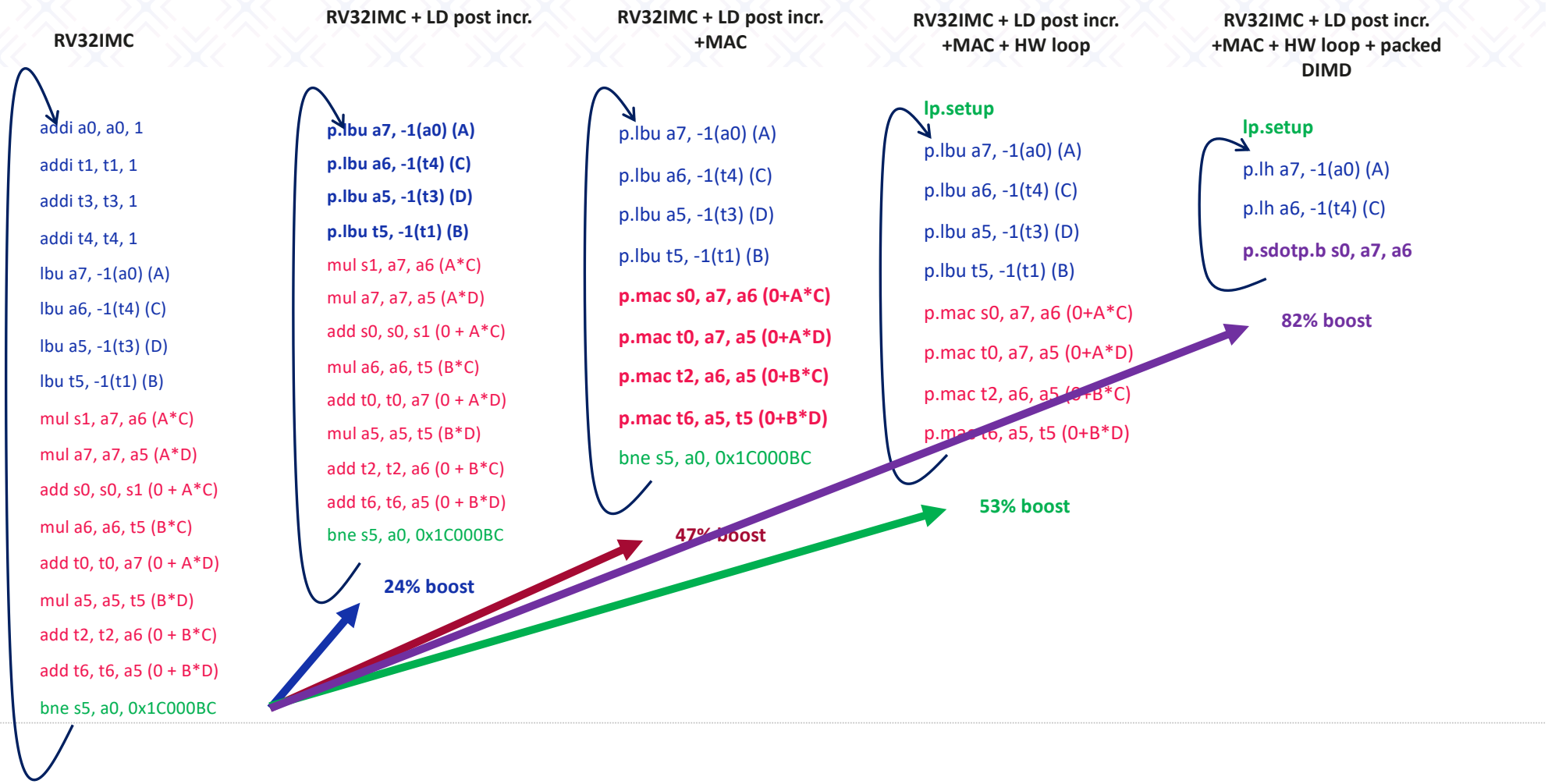
Introduction and Motivation

- Emerging application areas for AI-enabled IoT
  - Personalized Healthcare
  - Augmented Reality
  - **Nano-Robotics**

Introduction and Motivation

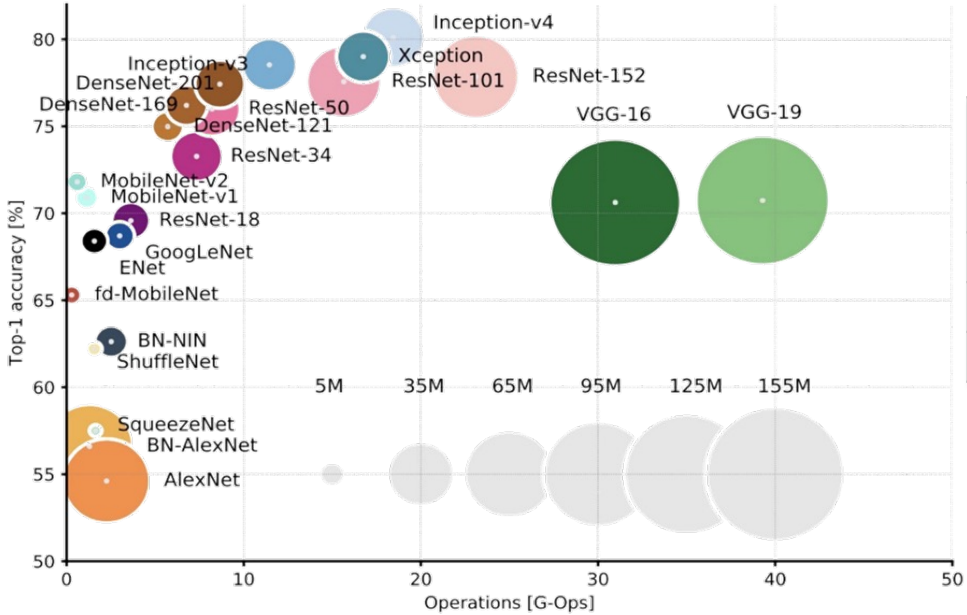
- Emerging application areas for AI-enabled IoT
  - Personalized Healthcare
  - Augmented Reality
  - Nano-Robotics
- **Challenges**
  - High computational demand from DNNs, other algorithms
  - Diverse computational patterns and requirements
- **Opportunities**
  - Bit-precision tolerance
  - Accelerable workloads

Xpulp: HW Loop Effect



Pushing Further: Quantized Neural Networks

Quantization (*)



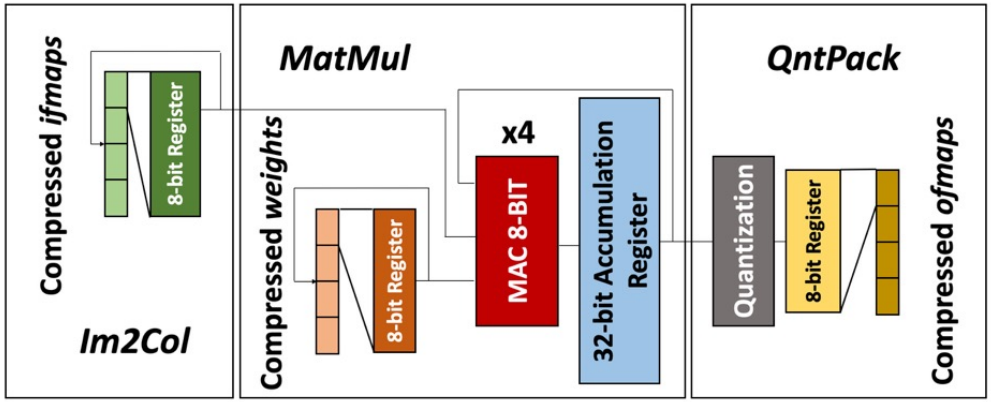
Quantization Method	Top1 Accuracy	Weight Memory Footprint
Full-Precision	70.9%	16.27 MB
INT-8	70.1%	4.06 MB
INT-4	66.46%	2.35 MB
Mixed-Precision	68%	2.09 MB

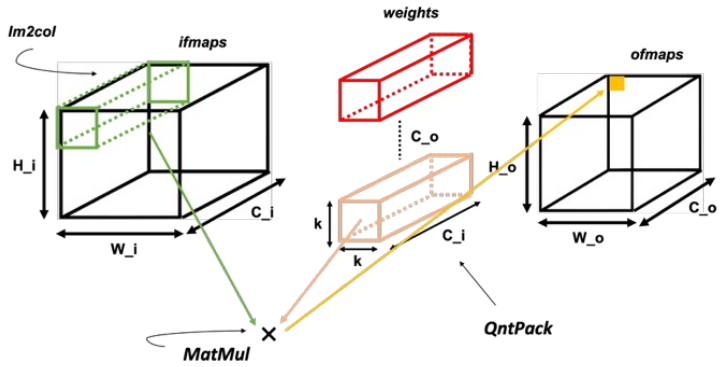
Quantized Neural Networks (QNNs) are the natural target for execution on constrained edge platforms.

PULP-NN Convolution Kernels (8-bit)

- Leverages an optimized computational model based on CMSIS-NN library
- Exploits HWC organization + SIMD MAC of Xpulpv2 ISA extension

- At every MatMul iteration it fetches
- 2 im2col
- 4 filters
- And generates 8 output pixels

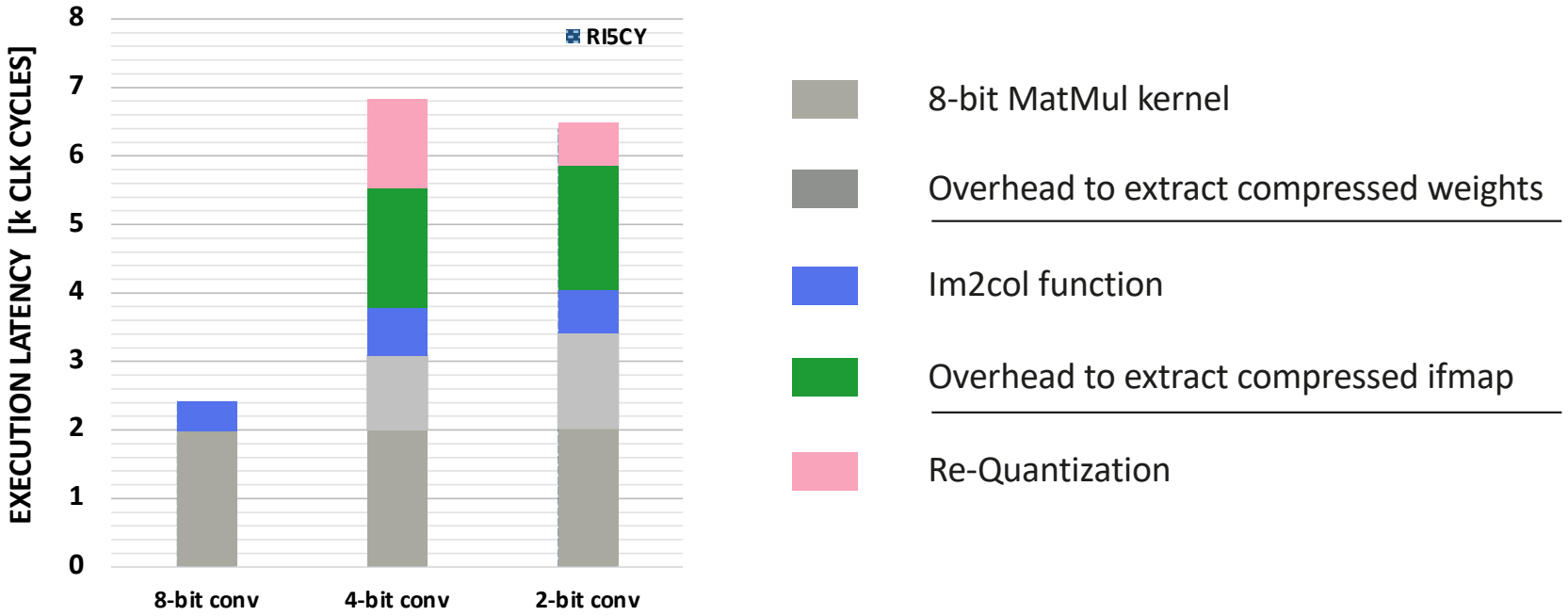




XpulpV2 Overheads (2-bit, 4-bit kernels)

Chips-IT

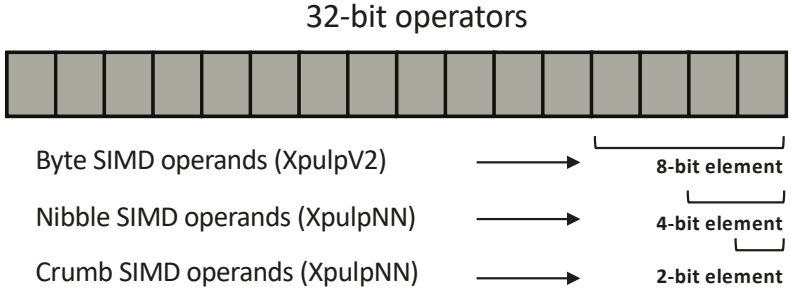
Sub-byte Convolution Kernel on XpulpV2 ISA



XpulpNN: ISA Extensions for QNN on PULP

Chips-IT

Arithmetic SIMD instructions



Supported Ops: ALU, Comparison, Shift, abs, Dot Product

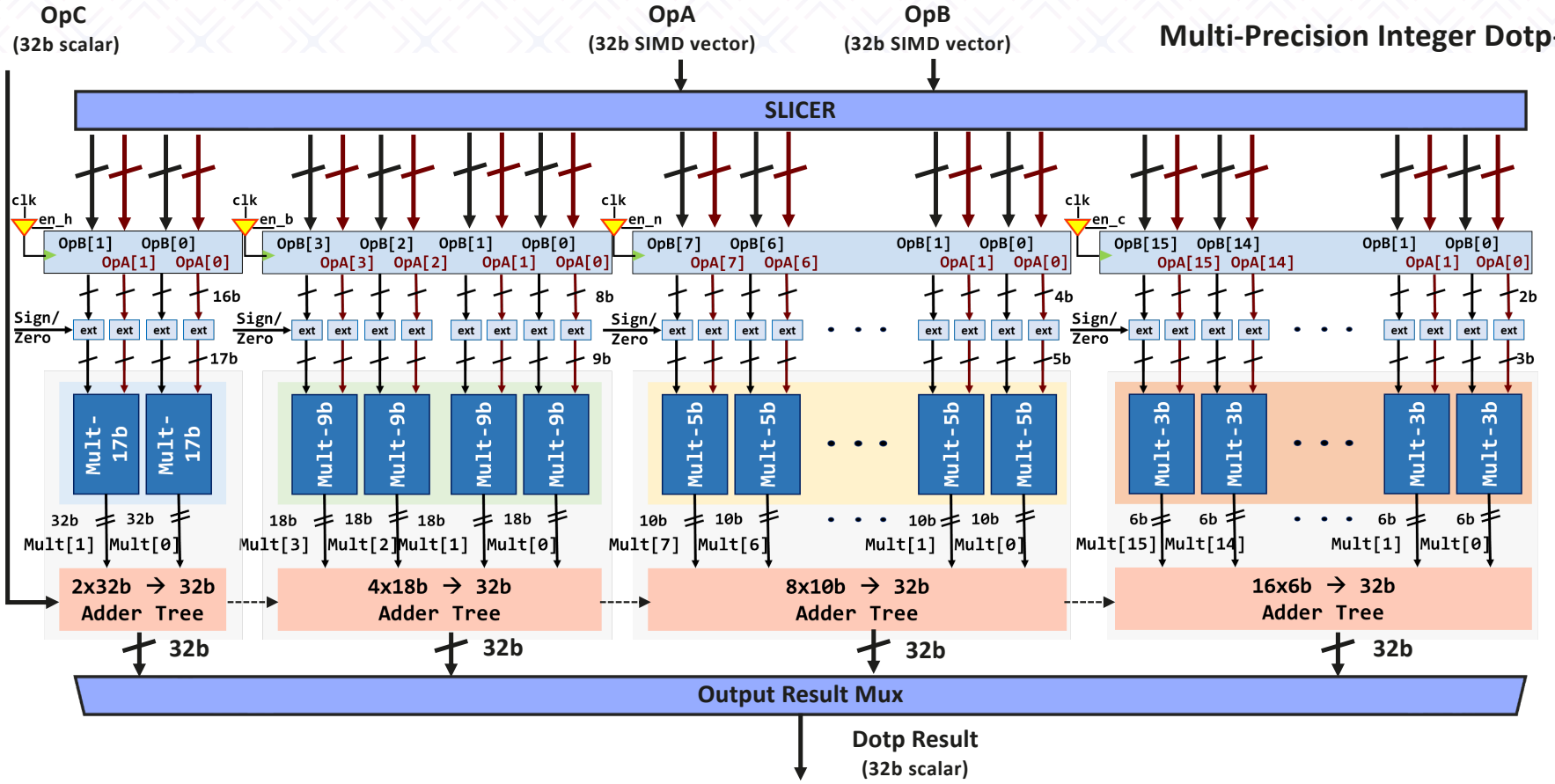
No need to unpack sub-byte data

Multi-cycle instruction to efficiently handle the quantization process in HW

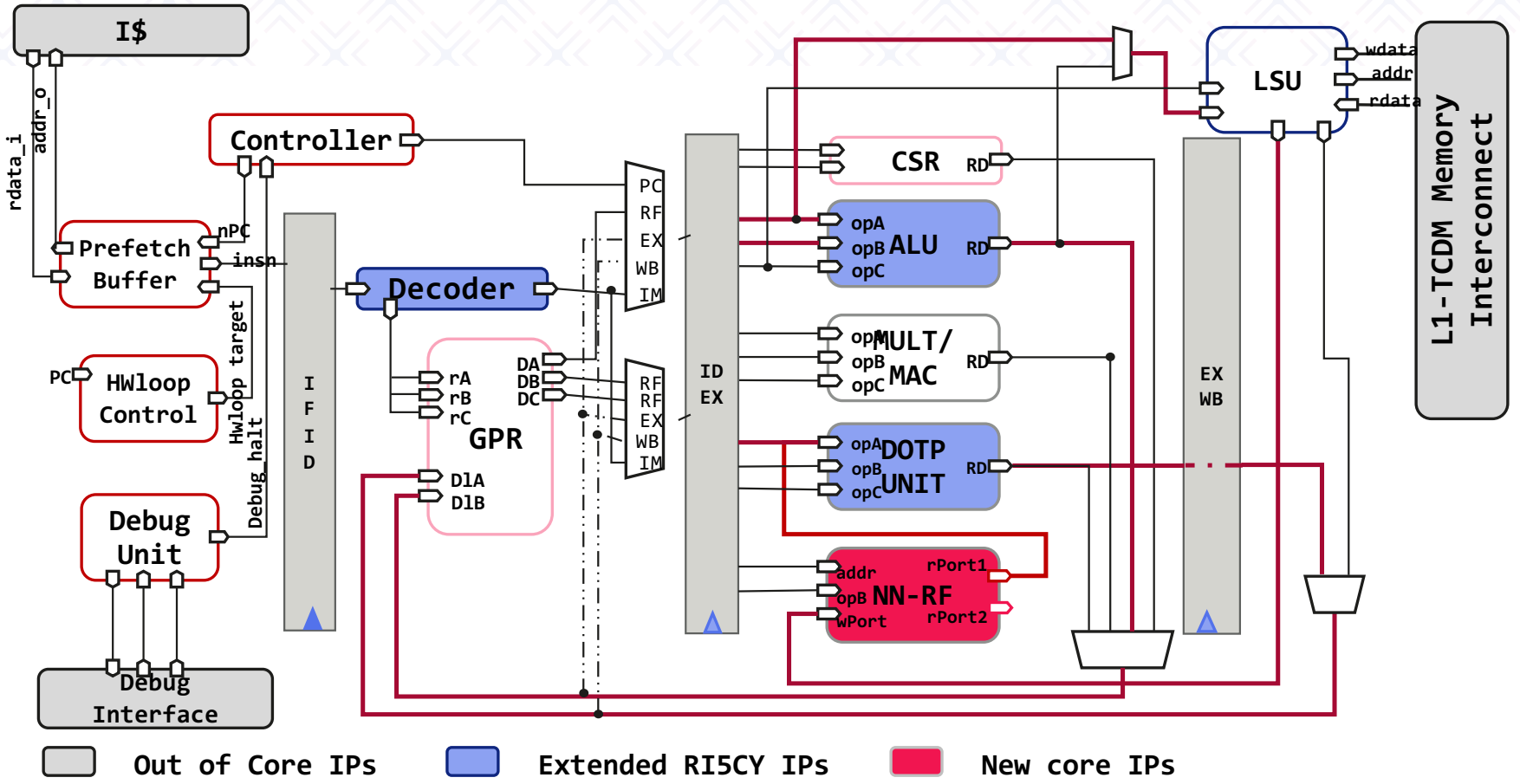
ALU SIMD Op.	Description for *nibble*
pv.add[.sc].{n, c}	rD[i] = rs1[i] + rs2[i]
pv.sub[.sc].{n, c}	rD[i] = rs1[i] - rs2[i]
pv.avg(u)[.sc].{n, c}	rD[i] = (rs1[i] + rs2[i])>>1
Vector Comparison Op.	
pv.max(u)[.sc].{n, c}	rD[i] = rs1[i] > rs2[i] ? rs1[i] : rs2[i]
pv.min(u)[.sc].{n, c}	rD[i] = rs1[i] <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

A. Garofalo, G. Tagliavini, F. Conti, L. Benini and D. Rossi, "XpulpNN: Enabling Energy Efficient and Flexible Inference of Quantized Neural Networks on RISC-V based IoT End Nodes," in *IEEE Transactions on Emerging Topics in Computing, 2021*

1) Multi-precision Dotp Unit



2) MAC/Load Extension



XPulpNN effect

Chips-IT

MatMul pseudo-assembly code:

Perf. Degradation due to loads in the innermost loop

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

XPulpNN effect

Chips-IT

INIT NN-RF

```
pv.nnsdotusp.h  zero, aw1,16
pv.nnsdotusp.h  zero, aw2,18
pv.nnsdotusp.h  zero, aw3,20
pv.nnsdotusp.h  zero, aw4,22
pv.nnsdotusp.h  zero, ax1,8
```

MatMul pseudo-assembly code:

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

XPulpNN effect

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

- Single-cycle MAC + load instruction (**Mac&Load**) down to 2-bit width
- **Only one explicit load inside the innermost loop**

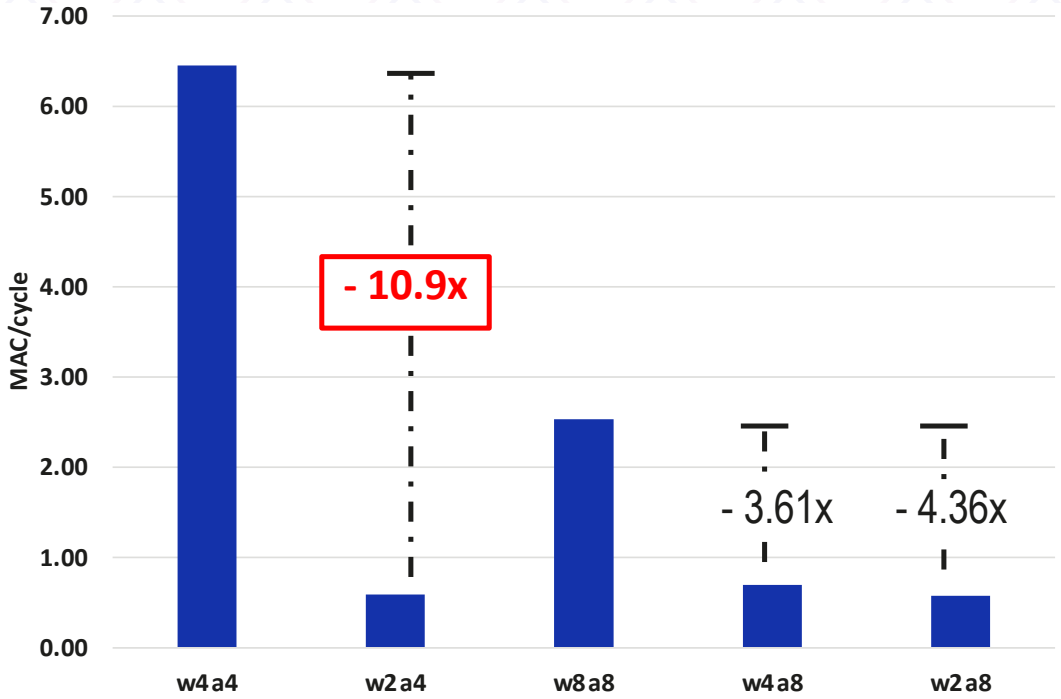
MatMul pseudo-assembly code:

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

XPulpNN effect

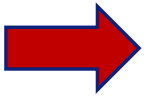
```
INIT    pv.nnsdotusp.h   zero, aw1,16
NN-RF   pv.nnsdotusp.h   zero, aw2,18
        pv.nnsdotusp.h   zero, aw3,20
        pv.nnsdotusp.h   zero, aw4,22
        pv.nnsdotusp.h   zero, ax1,8
        lp.setup         i1, i2, end
        pv.nnsdotup.h    zero,ax2,9
        pv.nnsdotusp.b   s1, aw2, 0
        pv.nnsdotusp.b   s2, aw4, 2
        pv.nnsdotusp.b   s3, aw3, 4
        pv.nnsdotusp.b   s4, ax1, 14
        pv.nnsdotusp.b   s5, aw2, 17
        pv.nnsdotusp.b   s6, aw4, 19
        pv.nnsdotusp.b   s7, aw3, 21
end:    pv.nnsdotusp.b   s8, aw1, 23
```

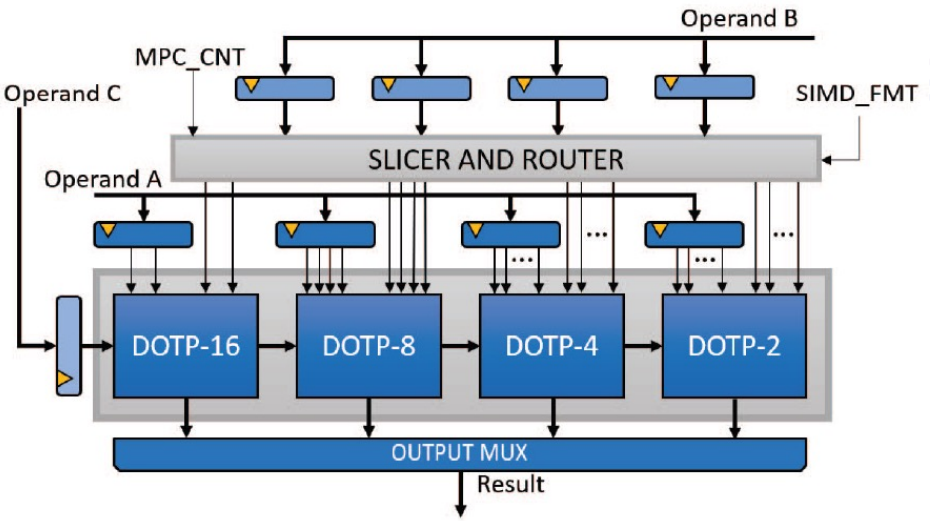
- Single-cycle MAC + load instruction (**Mac&Load**) down to 2-bit width
- **Only one explicit load inside the innermost loop**



MPIC

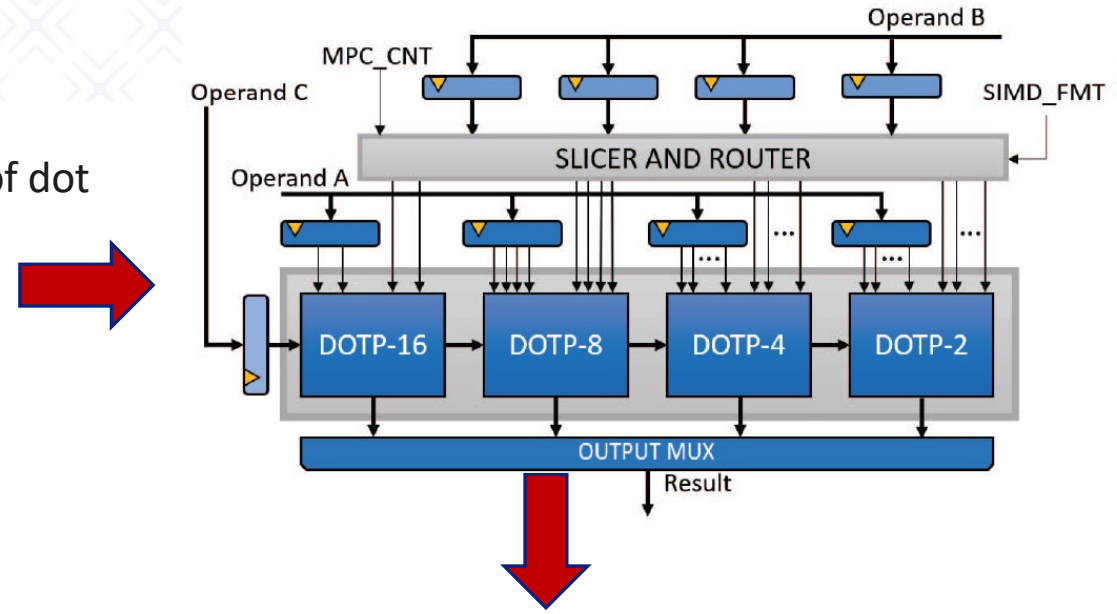
- HW sub-byte and mixed-precision sum of dot products (sdotp)





MPIC

- HW sub-byte and mixed-precision sum of dot products (sdotp)
- Virtual SIMD instructions
- Dynamic bit-scalable execution mode

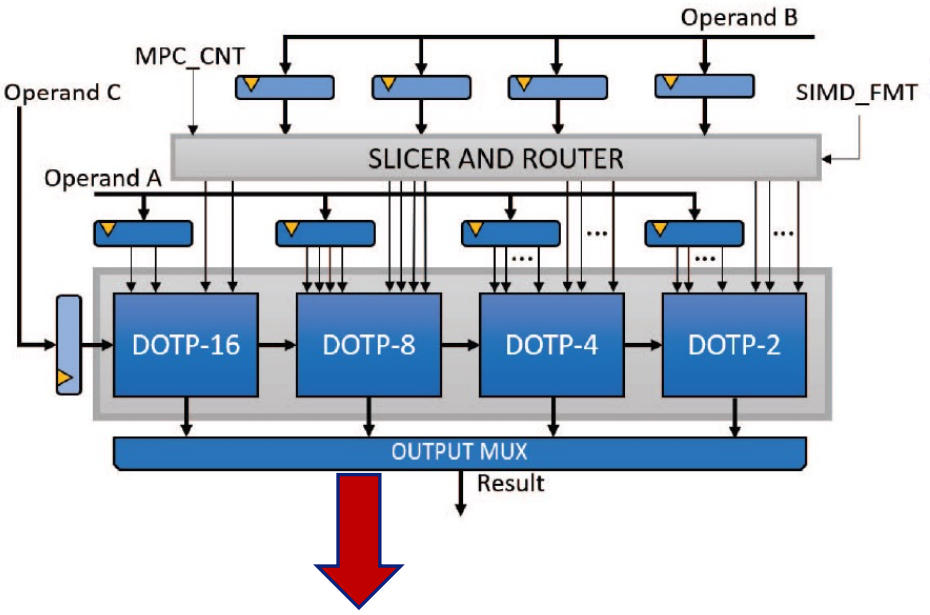


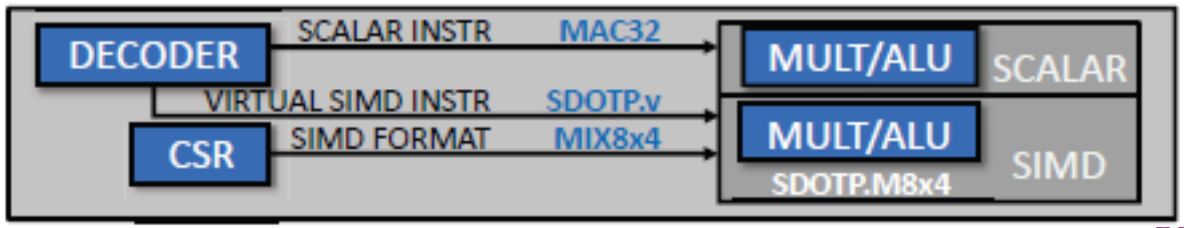

```
p.lw          x10,4(x4!)
p.lw          x11,4(x5!)
p.extract     x5,x11,4,0
p.extract     x6,x11,4,4
p.extract     x7,x11,4,8
p.extract     x8,x11,4,12
pv.packlo.b   x15,x5,x6
pv.packhi.b   x15,x7,x8
pv.sdotsp.b   x20,x15,x10
```

No more need for *packing* operations!

MPIC

- HW sub-byte and mixed-precision sum of dot products (sdotp)
- Virtual SIMD instructions
- Dynamic bit-scalable execution mode






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

No more need for *packing* operations!

Our proposal for energy-efficient inference of DNNs

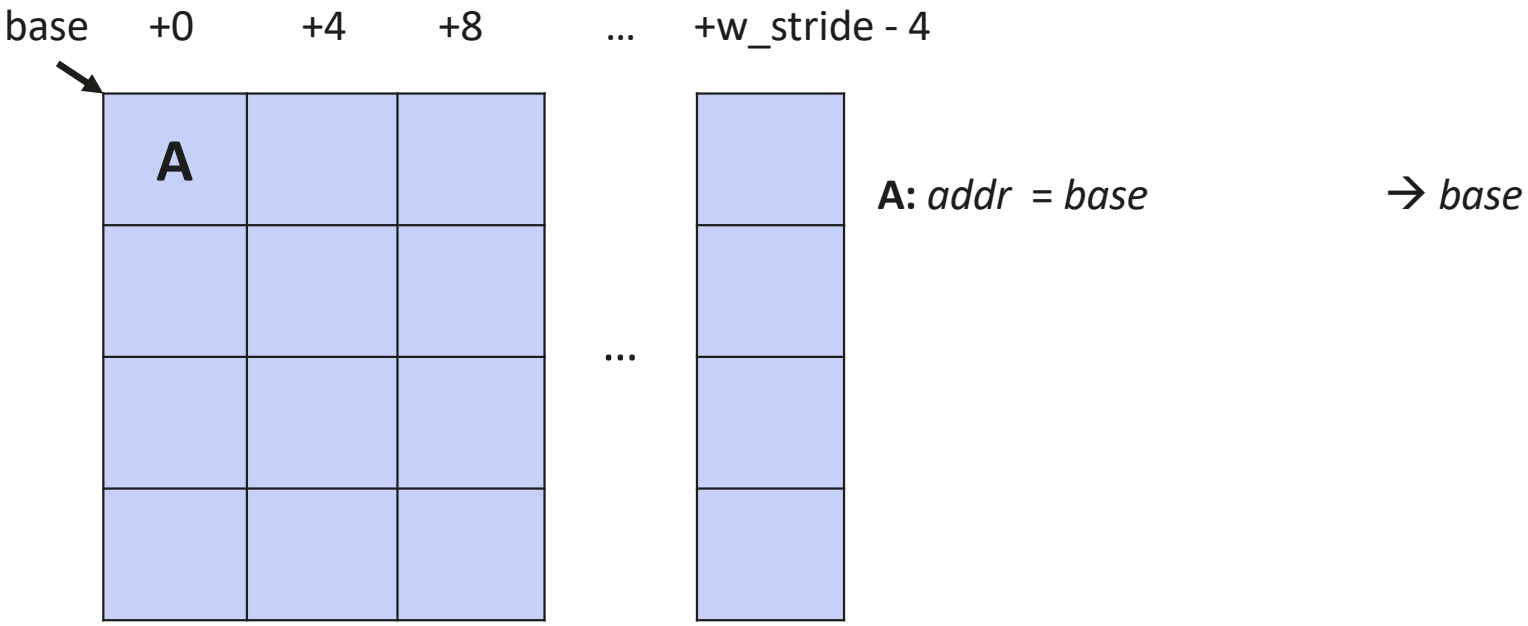
Chips-IT

- Flex-V core
- Performance of XpulpNN extensions
- Flexibility of MPIC
- Mixed-precision Mac&Load instructions
- Optimized SW library targeting well-known mixed-precision QNNs

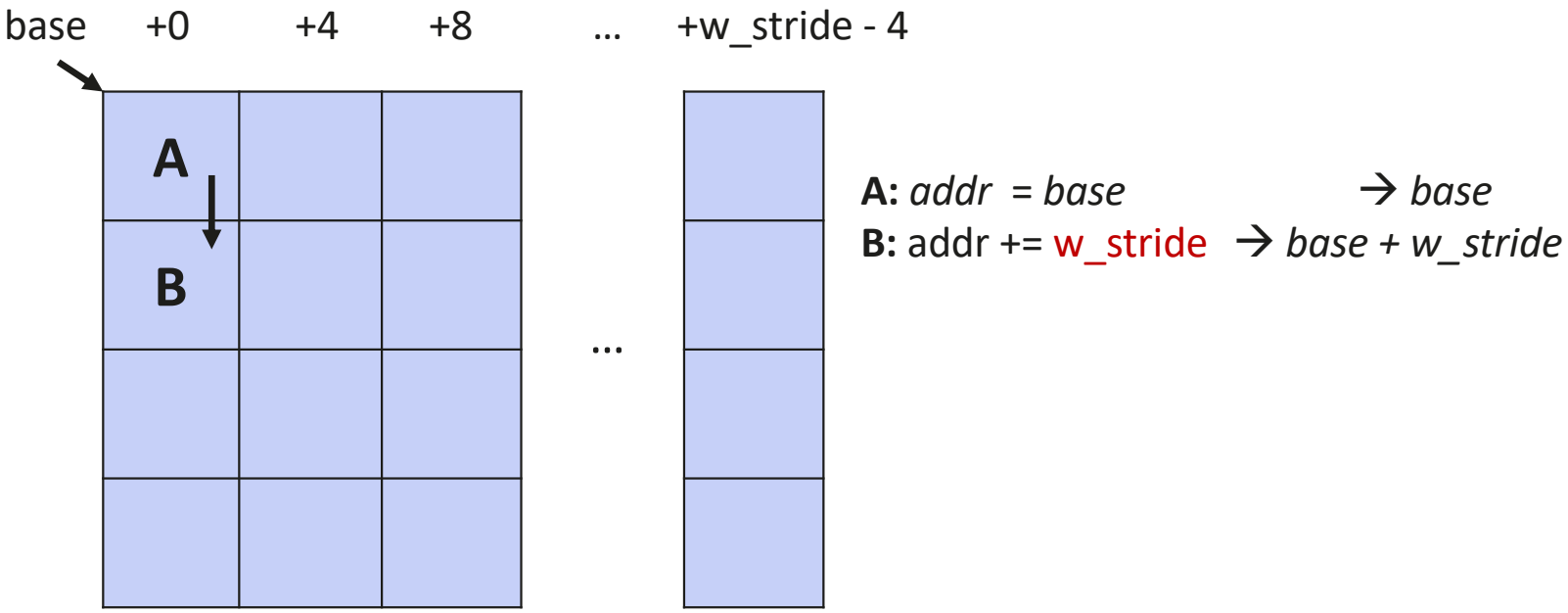
Optimized SW Library

HW support for mixed-precision

MatMul – Memory access pattern



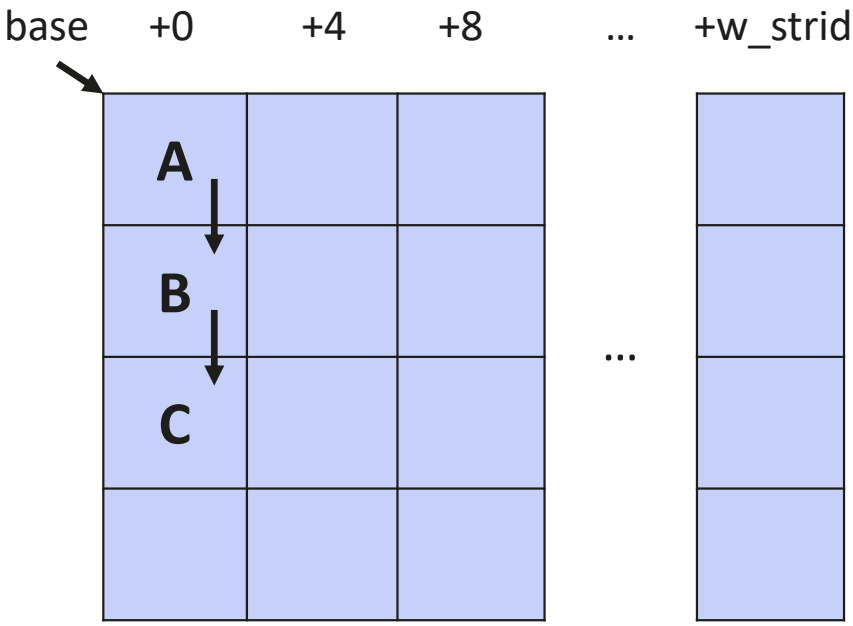
MatMul – Memory access pattern



A: *addr = base* → *base*

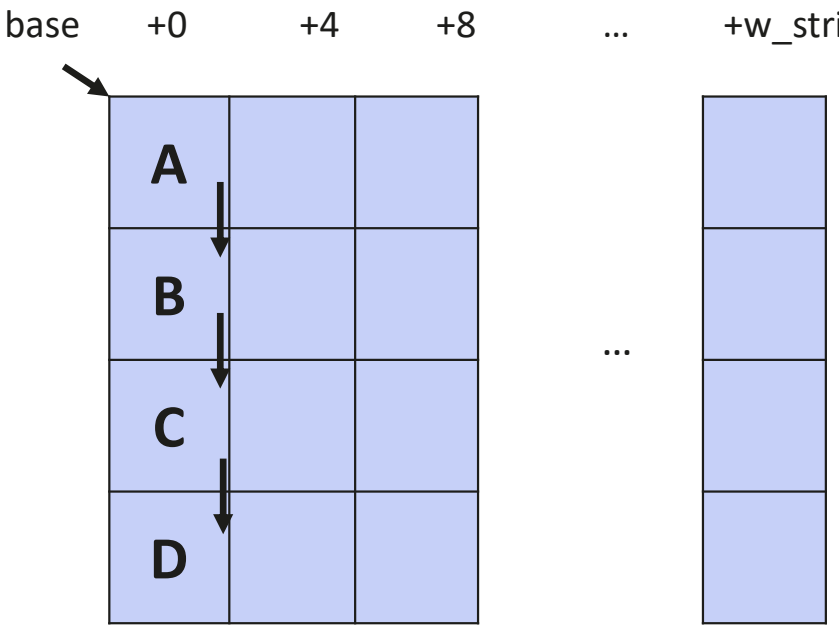
B: addr += w_stride → *base + w_stride*

MatMul – Memory access pattern



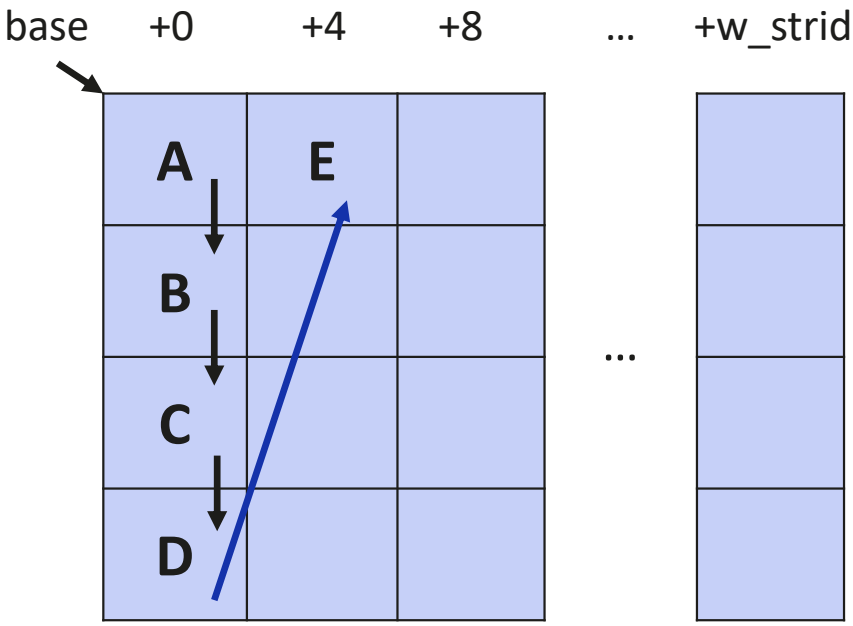
A: *addr = base* → *base*
B: addr += w_stride → *base + w_stride*
C: *addr += w_stride* → *base + 2*w_stride*

MatMul – Memory access pattern



A: *addr = base* → *base*
B: addr += w_stride → *base + w_stride*
C: *addr += w_stride* → *base + 2*w_stride*
D: *addr += w_stride* → *base + 3*w_stride*

MatMul – Memory access pattern



A: *addr = base* → *base*
B: addr += w_stride → *base + w_stride*
C: *addr += w_stride* → *base + 2*w_stride*
D: *addr += w_stride* → *base + 3*w_stride*
E: addr += *w_rollback* → *base + 4*

Automatic address generation

Chips-IT

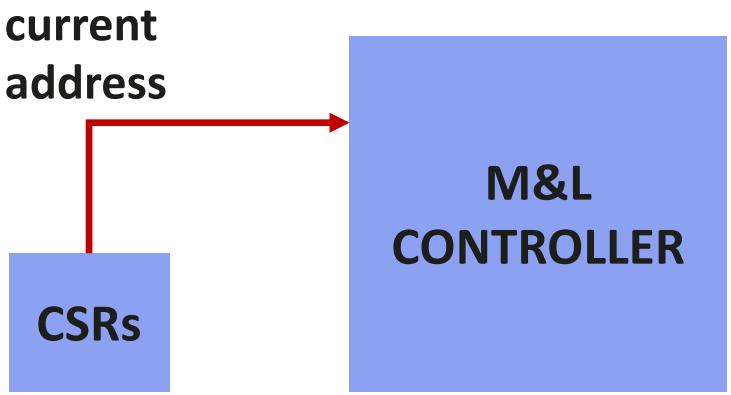
- Based on static information related to the kernel
- Needed invariant parameters are stored in CSRs
- Only one pointer for activations and one for weights
- Extensible to all 2D strided patterns, not only MatMuls!!

Automatic address generation

- Based on static information related to the kernel
- Needed invariant parameters are stored in CSRs
- Only one pointer for activations and one for weights
- Extensible to all 2D strided patterns, not only MatMuls!!

How does it work?

1. Read current address

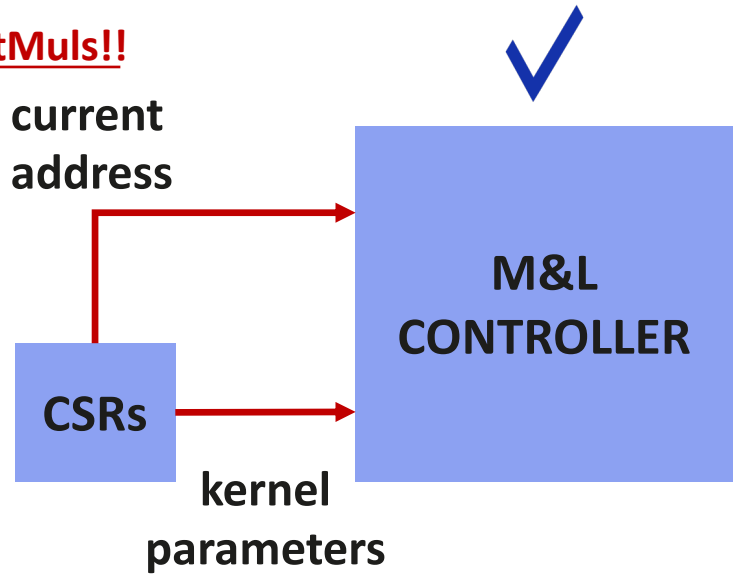


Automatic address generation

- Based on static information related to the kernel
- Needed invariant parameters are stored in CSRs
- Only one pointer for activations and one for weights
- Extensible to all 2D strided patterns, not only MatMuls!!

How does it work?

1. Read current address
2. Check number of performed updates

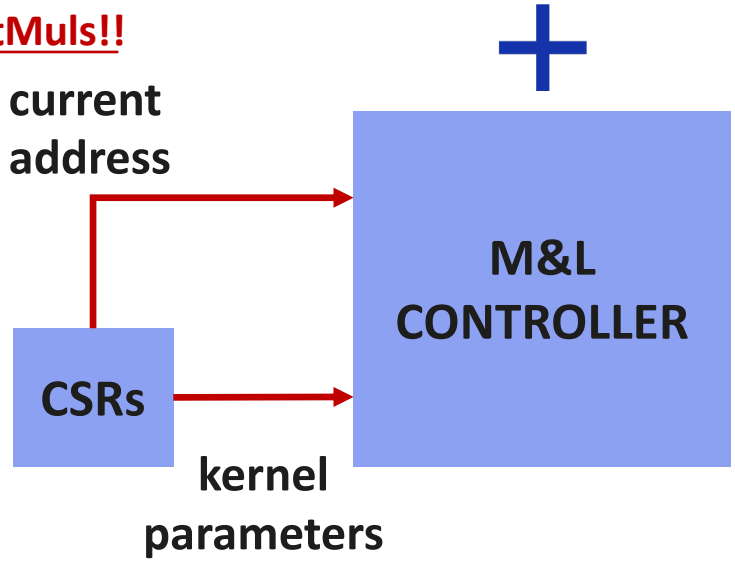


Automatic address generation

- Based on static information related to the kernel
- Needed invariant parameters are stored in CSRs
- Only one pointer for activations and one for weights
- Extensible to all 2D strided patterns, not only MatMuls!!

How does it work?

1. Read current address
2. Check number of performed updates
3. Adds proper increment

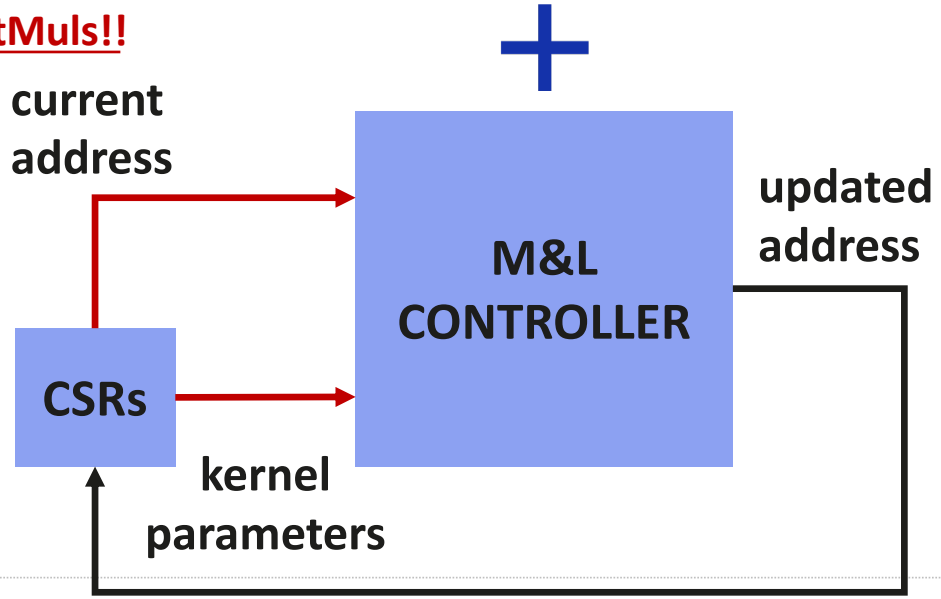


Automatic address generation

- Based on static information related to the kernel
- Needed invariant parameters are stored in CSRs
- Only one pointer for activations and one for weights
- Extensible to all 2D strided patterns, not only MatMuls!!

How does it work?

1. Read current address
2. Check number of performed updates
3. Adds proper increment
4. New address stored back in related CSR



Mixed-precision + *M&L* + Automatic Address Generation


```
csrwi sb_legacy,  0          ┐
csrwi simd_fmt,   8          │
csrwi mix_skip,   16         │  CSRs CONF. OUTSIDE THE KERNEL
csrw  a_stride,   A_STRIDE   │
csrw  w_stride,   W_STRIDE   │
csrw  a_rollback, A_ROLLB    │
csrw  w_rollback, W_ROLLB    ┘
csrw  a_csr, A_BASE_ADDR     ┐  CSRs CONF. INSIDE THE KERNEL
csrw  w_csr, W_BASE_ADDR     ┘
pv.mlsdotsp.h zero, aw, 16   ┐
pv.mlsdotsp.h zero, aw, 18   │  INIT THE NN-RF
pv.mlsdotsp.h zero, aw, 20   │
pv.mlsdotsp.h zero, aw, 22   │
pv.mlsdotsp.h zero, ax, 8    ┘
lp.setup l1, l2, end
```

```
                    pv.mlsdotsp.h zero, ax, 9
                    pv.mlsdotusp.b s1,  aw, 0
                    pv.mlsdotusp.b s2,  aw, 2
                    pv.mlsdotusp.b s3,  aw, 4
                    pv.mlsdotusp.b s4,  ax, 14
                    …
                    pv.mlsdotusp.b s13, aw, 1
                    pv.mlsdotusp.b s14, aw, 3
                    pv.mlsdotusp.b s15, aw, 5
                    pv.mlsdotusp.b s16, ax, 15
                    pv.mlsdotusp.b s1,  aw, 0
                    …
                    pv.mlsdotusp.b s13, aw, 17
                    pv.mlsdotusp.b s14, aw, 19
                    pv.mlsdotusp.b s15, aw, 21
(end):              pv.mlsdotusp.b s16, aw, 23
```

Mixed-precision + *M&L* + Automatic Address Generation

- NO extraction/packing operation within the innermost loop of MatMuls
- Masked load operations
- Extension of the MatMul unrolling factor

```
csrwi sb_legacy,   0
csrwi simd_fmt,    8
csrwi mix_skip,    16
csrw  a_stride,    A_STRIDE
csrw  w_stride,    W_STRIDE
csrw  a_rollback,  A_ROLLB
csrw  w_rollback,  W_ROLLB
csrw  a_csr, A_BASE_ADDR
csrw  w_csr, W_BASE_ADDR
pv.mlsdotsp.h zero, aw, 16
pv.mlsdotsp.h zero, aw, 18
pv.mlsdotsp.h zero, aw, 20
pv.mlsdotsp.h zero, aw, 22
pv.mlsdotsp.h zero, ax, 8
lp.setup l1, l2, end
```

CSRs CONF. OUTSIDE THE KERNEL

CSRs CONF. INSIDE THE KERNEL

INIT THE NN-RF

```
                    pv.mlsdotsp.h zero, ax, 9
                    pv.mlsdotusp.b s1,  aw, 0
                    pv.mlsdotusp.b s2,  aw, 2
                    pv.mlsdotusp.b s3,  aw, 4
                    pv.mlsdotusp.b s4,  ax, 14
                    …
                    pv.mlsdotusp.b s13, aw, 1
                    pv.mlsdotusp.b s14, aw, 3
                    pv.mlsdotusp.b s15, aw, 5
                    pv.mlsdotusp.b s16, ax, 15
                    pv.mlsdotusp.b s1,  aw, 0
                    …
                    pv.mlsdotusp.b s13, aw, 17
                    pv.mlsdotusp.b s14, aw, 19
                    pv.mlsdotusp.b s15, aw, 21
(end):              pv.mlsdotusp.b s16, aw, 23
```

Mixed-precision + *M&L* + Automatic Address Generation

- NO extraction/packing operation within the innermost loop of MatMuls
- Masked load operations
- Extension of the MatMul unrolling factor
- At the cost of simple writings to the CSRs outside the body of the loop

```
csrwi sb_legacy,   0
csrwi simd_fmt,    8
csrwi mix_skip,    16
csrw  a_stride,    A_STRIDE
csrw  w_stride,    W_STRIDE
csrw  a_rollback,  A_ROLLB
csrw  w_rollback,  W_ROLLB
csrw  a_csr, A_BASE_ADDR
csrw  w_csr, W_BASE_ADDR
pv.mlsdotsp.h zero, aw, 16
pv.mlsdotsp.h zero, aw, 18
pv.mlsdotsp.h zero, aw, 20
pv.mlsdotsp.h zero, aw, 22
pv.mlsdotsp.h zero, ax, 8
lp.setup l1, l2, end
```

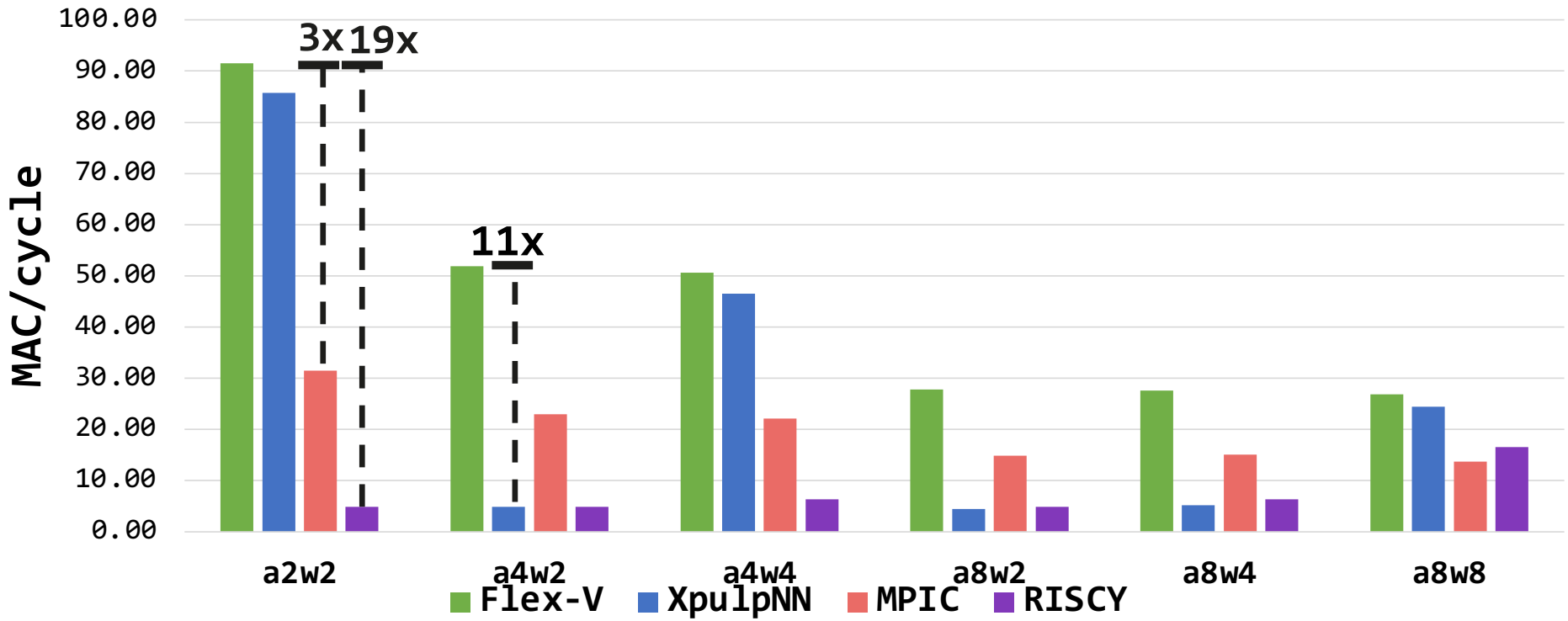
CSRs CONF. OUTSIDE THE KERNEL

CSRs CONF. INSIDE THE KERNEL

INIT THE NN-RF

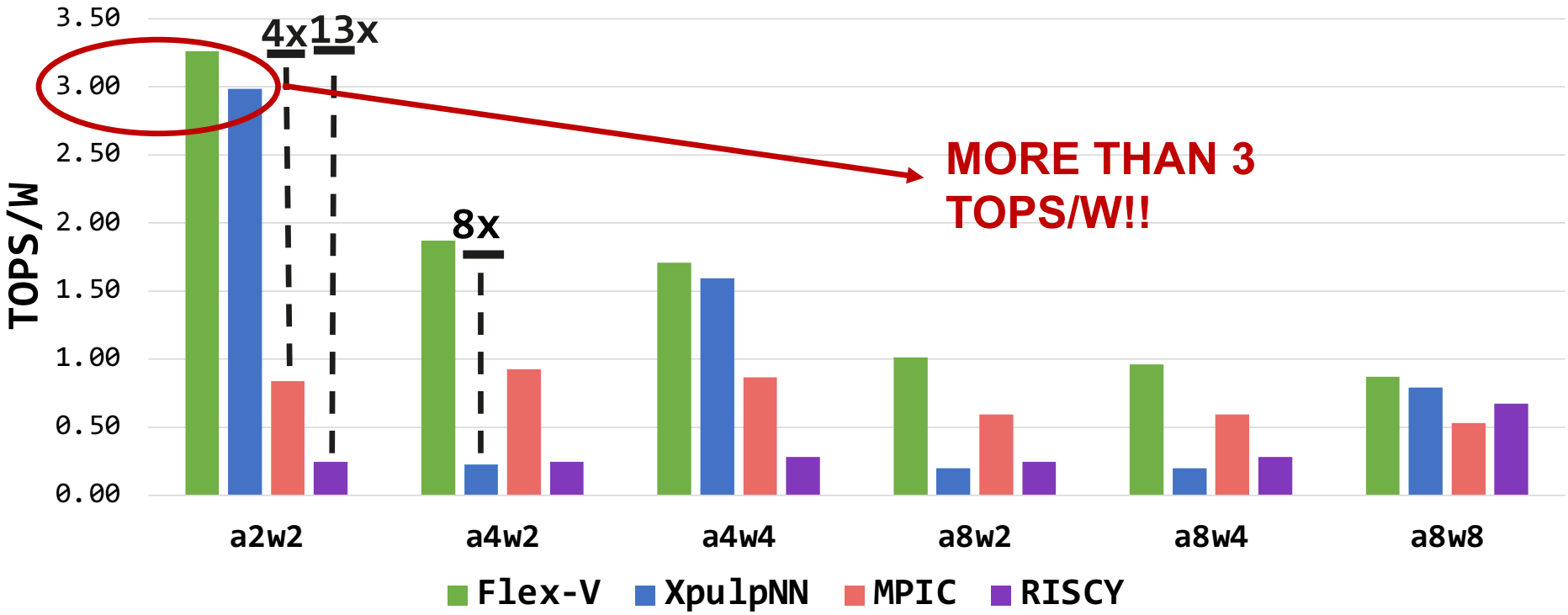
```
                    pv.mlsdotsp.h zero, ax, 9
                    pv.mlsdotusp.b s1,  aw, 0
                    pv.mlsdotusp.b s2,  aw, 2
                    pv.mlsdotusp.b s3,  aw, 4
                    pv.mlsdotusp.b s4,  ax, 14
                    …
                    pv.mlsdotusp.b s13, aw, 1
                    pv.mlsdotusp.b s14, aw, 3
                    pv.mlsdotusp.b s15, aw, 5
                    pv.mlsdotusp.b s16, ax, 15
                    pv.mlsdotusp.b s1,  aw, 0
                    …
                    pv.mlsdotusp.b s13, aw, 17
                    pv.mlsdotusp.b s14, aw, 19
                    pv.mlsdotusp.b s15, aw, 21
(end):              pv.mlsdotusp.b s16, aw, 23
```

Results – Single kernels Performance



Results – Single kernels Energy Efficiency

Physical implementation
in GF-22nm technology



Results – Full network

Network	MobileNetV1 (8b)	MobileNetV1 (8b4b)	ResNet-20 (4b2b)
Top-1 Accuracy	69.3 %	66.0 %	90.2 % [1]
Deg. W.r.t. 8b	-	3.3 %	0.15 %
Model size	1.9 MB	997 kB	142 kB
Memory saved	-	47 %	63 %
Performance (MAC/cycle)			
STM32H7	0.33	0.30	-
XpulpV2	5.6	3.2	4.8
XpulpNN	6.0	2.7	4.4
Flex-V	6.0	5.8	11.2

Results – Full network

Network	MobileNetV1 (8b)	MobileNetV1 (8b4b)	ResNet-20 (4b2b)
Top-1 Accuracy	69.3 %	66.0 %	90.2 % [1]
Deg. W.r.t. 8b	-	3.3 %	0.15 %
Model size	1.9 MB	997 kB	142 kB
Memory saved	-	47 %	63 %
Performance (MAC/cycle)			
STM32H7	0.33	0.30	-
XpulpV2	5.6	3.2	4.8
XpulpNN	6.0	2.7	4.4
Flex-V	6.0	5.8	11.2

Results – Full network

Chips-IT

Network	MobileNetV1 (8b)	MobileNetV1 (8b4b)	ResNet-20 (4b2b)
Top-1 Accuracy	69.3 %	66.0 %	90.2 % [1]
Deg. W.r.t. 8b	-	3.3 %	0.15 %
Model size	1.9 MB	997 kB	142 kB
Memory saved	-	47 %	63 %
Performance (MAC/cycle)			
STM32H7	0.33	0.30	-
XpulpV2	5.6	3.2	4.8
XpulpNN	6.0	2.7	4.4
Flex-V	6.0	5.8	11.2

~ 20x

Results – Full network

Network	MobileNetV1 (8b)	MobileNetV1 (8b4b)	ResNet-20 (4b2b)
Top-1 Accuracy	69.3 %	66.0 %	90.2 % [1]
Deg. W.r.t. 8b	-	3.3 %	0.15 %
Model size	1.9 MB	997 kB	142 kB
Memory saved	-	47 %	63 %
Performance (MAC/cycle)			
STM32H7	0.33	0.30	-
XpulpV2	5.6	3.2	4.8
XpulpNN	6.0	2.7	4.4
Flex-V	6.0	5.8	11.2

~ 20x

~ 2x

Results – Full network

Chips-IT

Network	MobileNetV1 (8b)	MobileNetV1 (8b4b)	ResNet-20 (4b2b)
Top-1 Accuracy	69.3 %	66.0 %	90.2 % [1]
Deg. W.r.t. 8b	-	3.3 %	0.15 %
Model size	1.9 MB	997 kB	142 kB
Memory saved	-	47 %	63 %
Performance (MAC/cycle)			
STM32H7	0.33	0.30	-
XpulpV2	5.6	3.2	4.8
XpulpNN	6.0	2.7	4.4
Flex-V	6.0	5.8	11.2

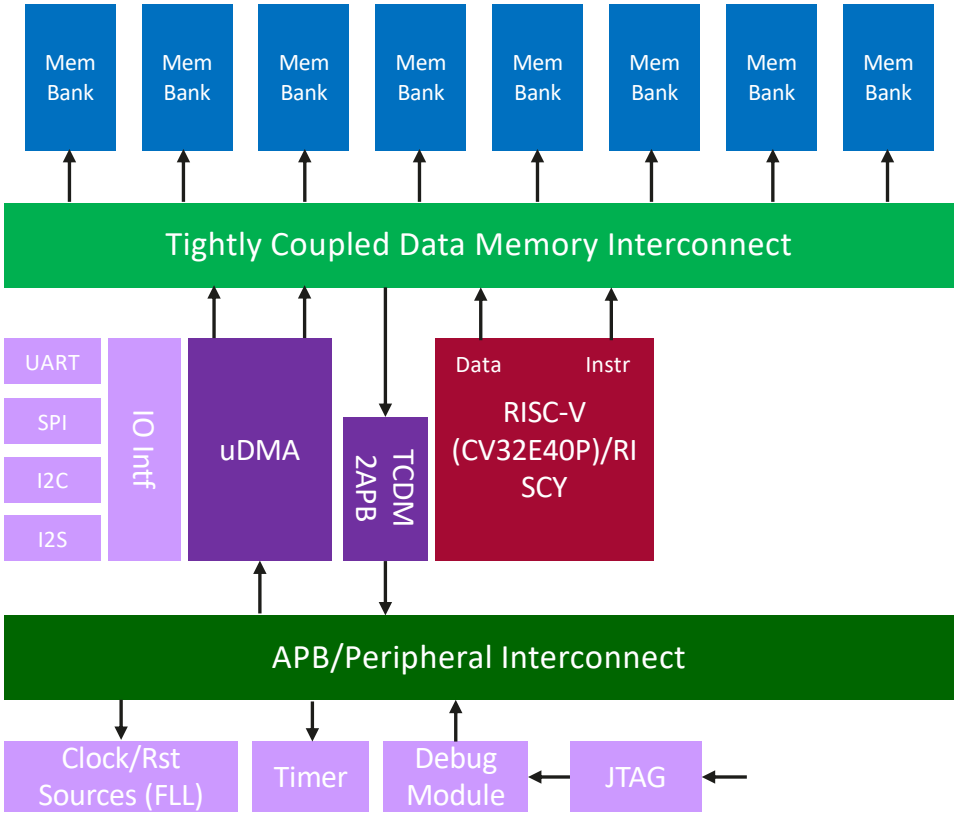
~ 20x

~ 2x

~ 2.5x

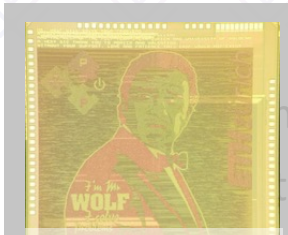
PULPissimo: embedded IoT controller

- Compact and tiny micro controller
- Extreme ultra-low power
- Simple control tasks and peripherals management

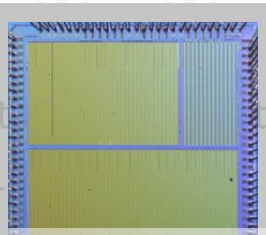


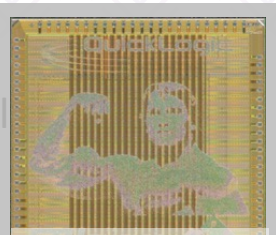
PULPissimo: embedded IoT controller

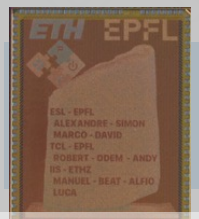
Chips-IT



Mr. Wolf 40nm

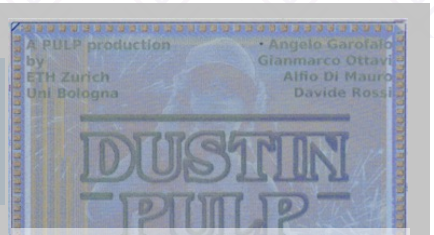

Poseidon 22nm


Arnold 22nm

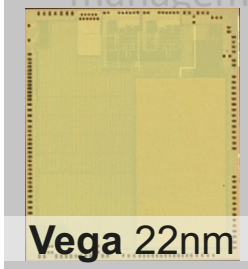


Rosetta 65nm

Xavier 65nm



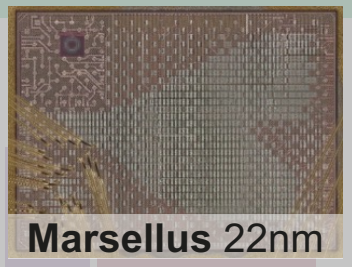
Dustin 65nm


Vega 22nm

Darkside 65nm

Echoes 65nm

Kraken 22nm

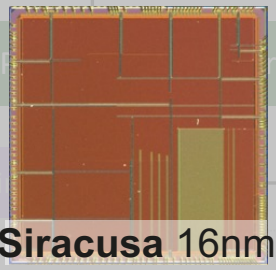

Marsellus 22nm

Cerberus 65nm

Eclipse 65nm

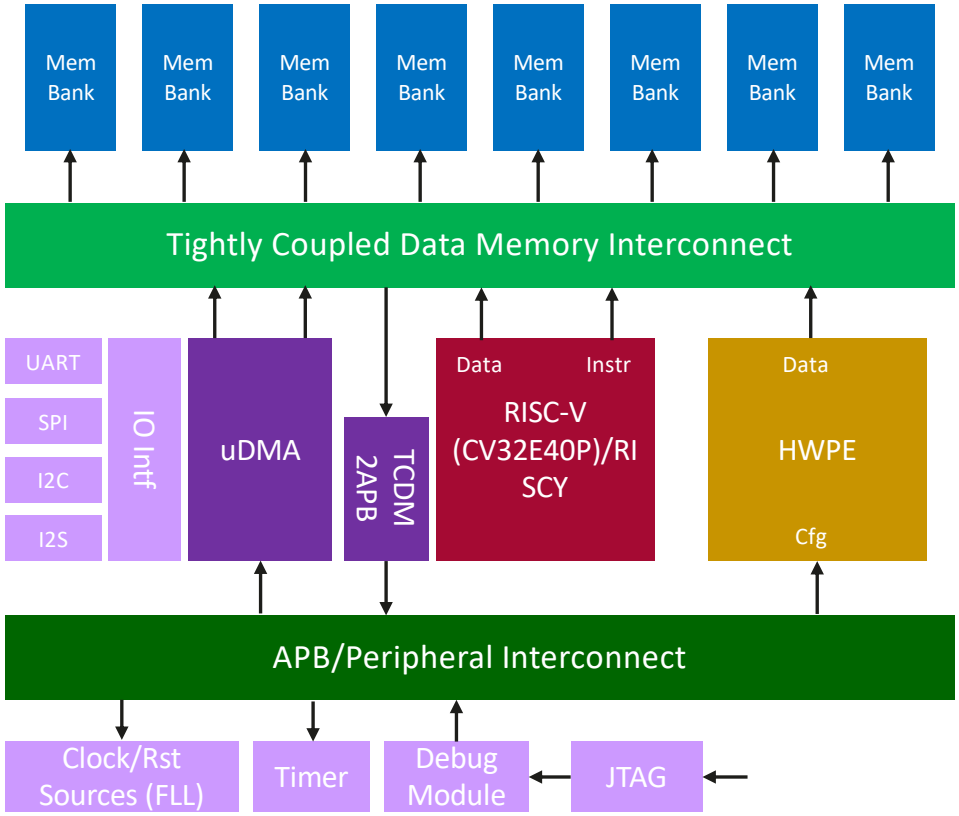
Kairos 65nm

Trikarenos 28nm


Siracusa 16nm

...

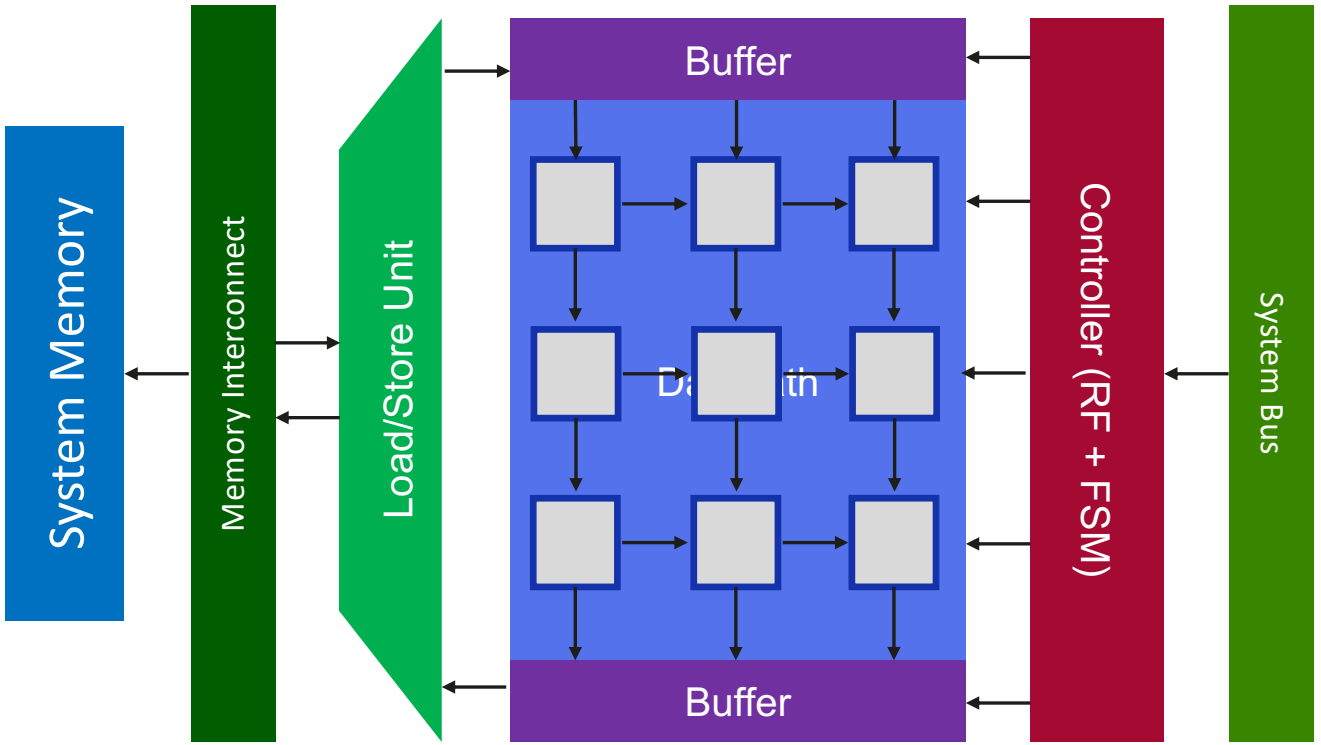
Are instructions extensions enough?



Cooperative Hardware Processing Engines (HWPE)

HWPE efficiency vs. optimized RISC-V core

- Specialized datapath (e.g. systolic MAC) & internal storage (e.g. linebuffer, accum-regs)
- Dedicated control (no I-fetch) with shadow registers (overlapped config-exec)
- Specialized high-BW interco into L1 (on data-plane)



Reconfigurable Binary Engine (RBE)

- Partially bit-serial dataflow
- "Bit" dimension is decomposed, in **inputs**, **weights**, **outputs**

$$y(k_{out}) = quant\left(\sum_{k_{in}}\left(W(k_{out}, k_{in}) \otimes x(k_{in})\right)\right)$$

32b accumulator

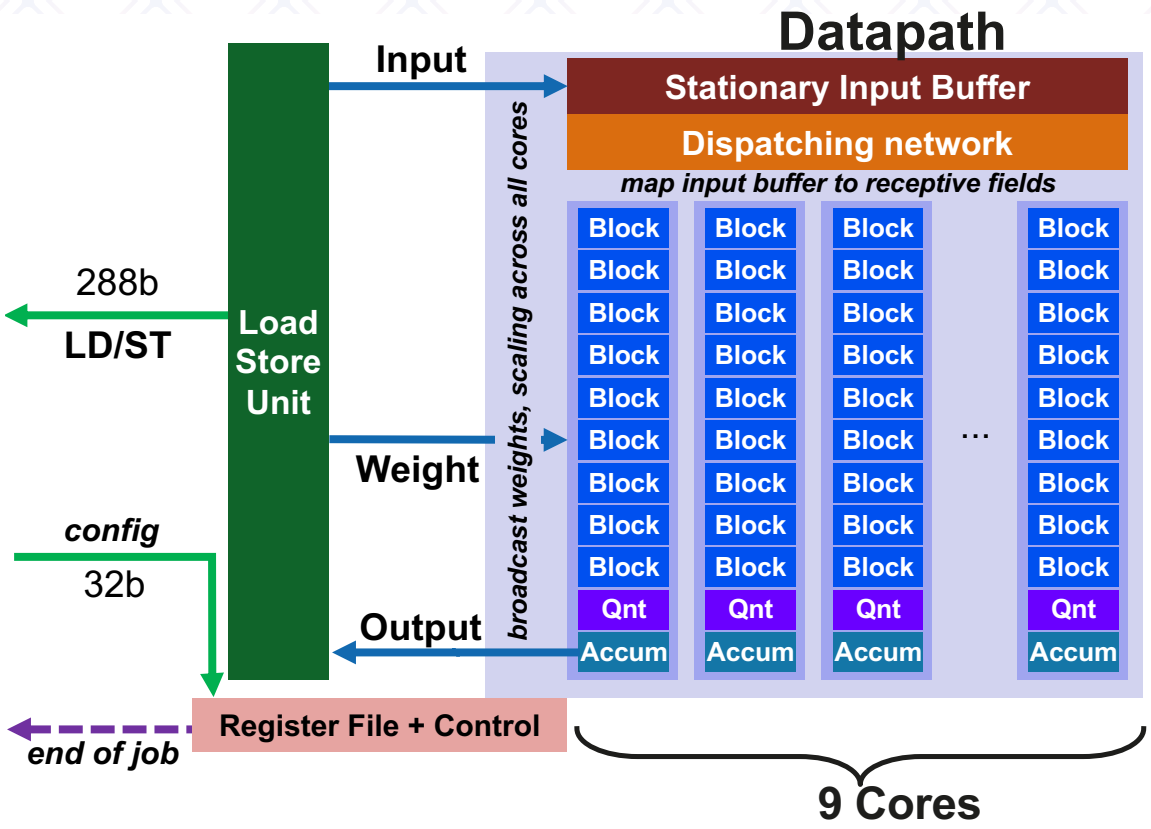
O-b output

W-b weights

I-b input

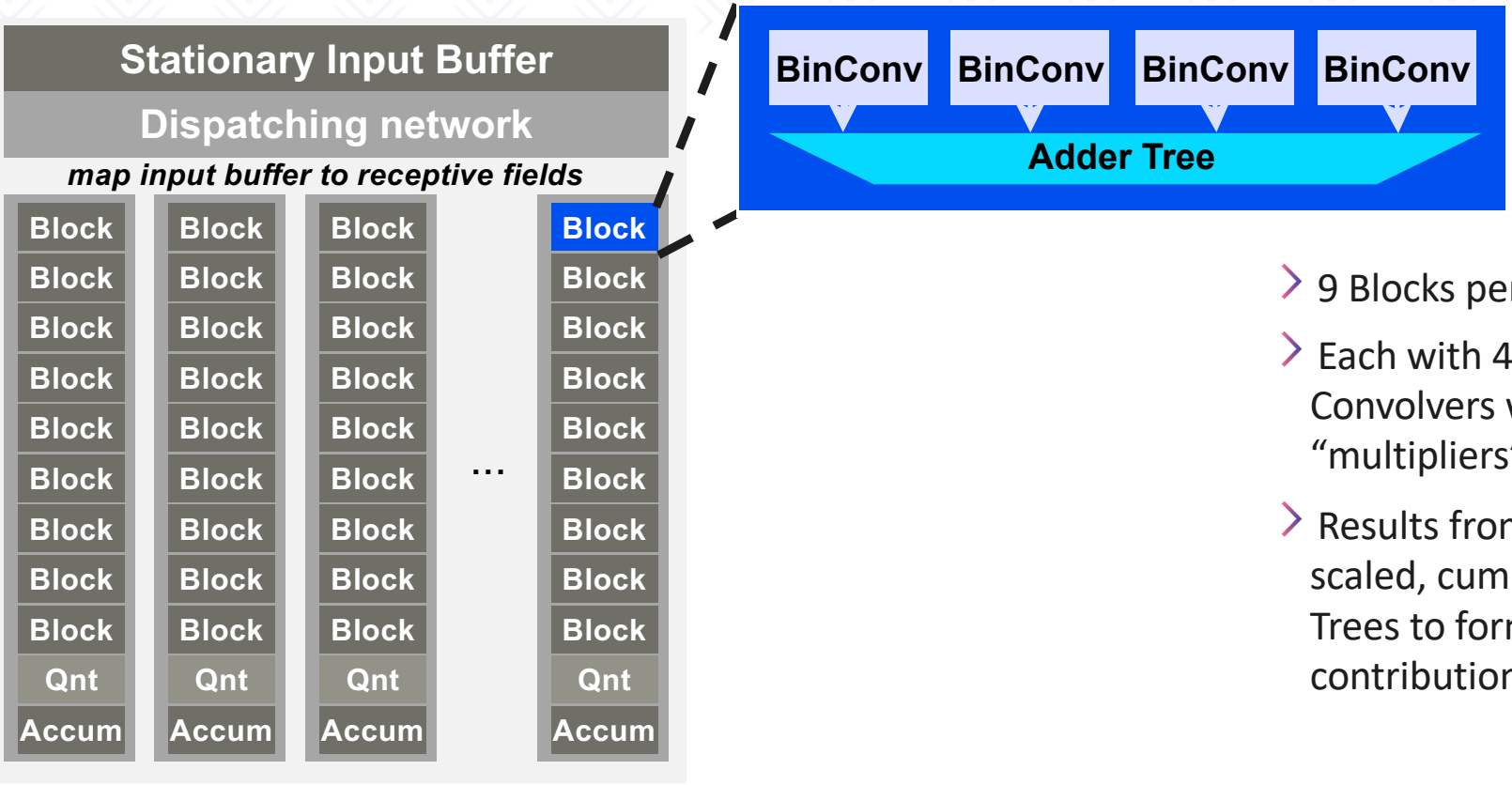
- DNN layer operating at WxI-b decomposed in WxI 1x1b MAC ops
- RBE tensors use a special layout in memory to expose 1x1b-ops
- RBE is integrated in cluster with
- 288b LD/ST initiator port toward CLUSTER interconnect
- 32b config port + "end of job" event

Reconfigurable Binary Engine (RBE)



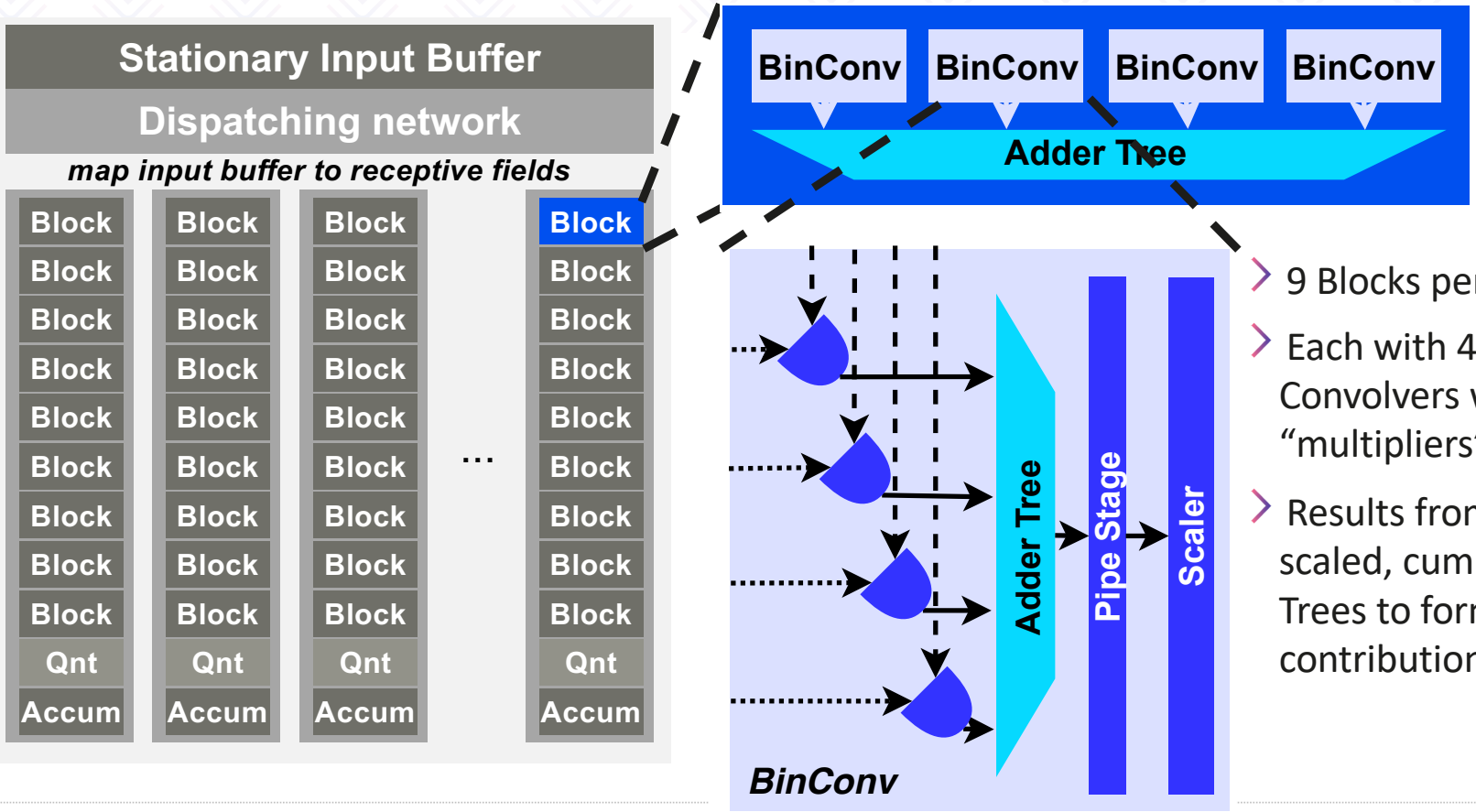
- RBE is 652 kGE divided in
- Register File + Control (6%)
- 288b Load/Store Unit (2%)
- Datapath (92%)
- Datapath comprise 9 Cores
- 1 Core = receptive field of one output in space across 32 Kout
- Stationary input statically multi-casted to each Core receptive field
- Output stored in 32x 32b accumulators per Core + Quantized
- Weights renovated each cycle and broadcasted on all Cores

Reconfigurable Binary Engine (RBE)



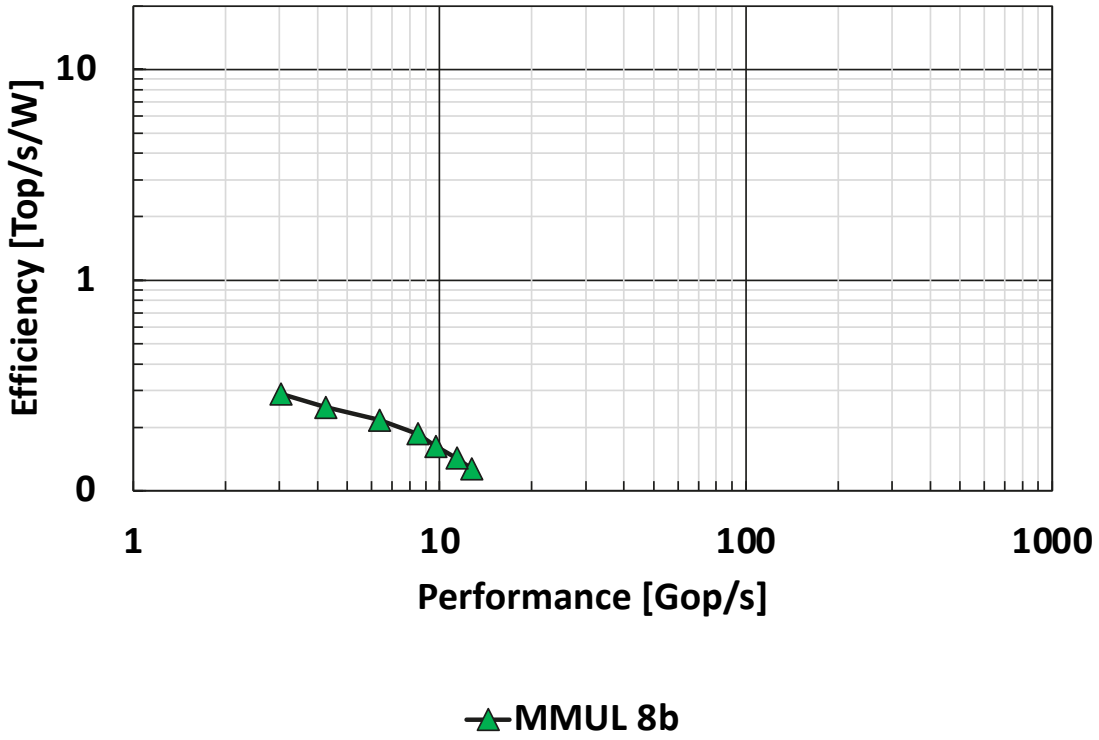
- 9 Blocks per Core
- Each with 4 Binary Convolvers with 32 1x1b “multipliers” (i.e., AND gates)
- Results from all BinConvs are scaled, cumulated with Adder Trees to form a complete contribution

Reconfigurable Binary Engine (RBE)



- 9 Blocks per Core
- Each with 4 Binary Convolvers with 32 1x1b “multipliers” (i.e., AND gates)
- Results from all BinConvs are scaled, cumulated with Adder Trees to form a complete contribution

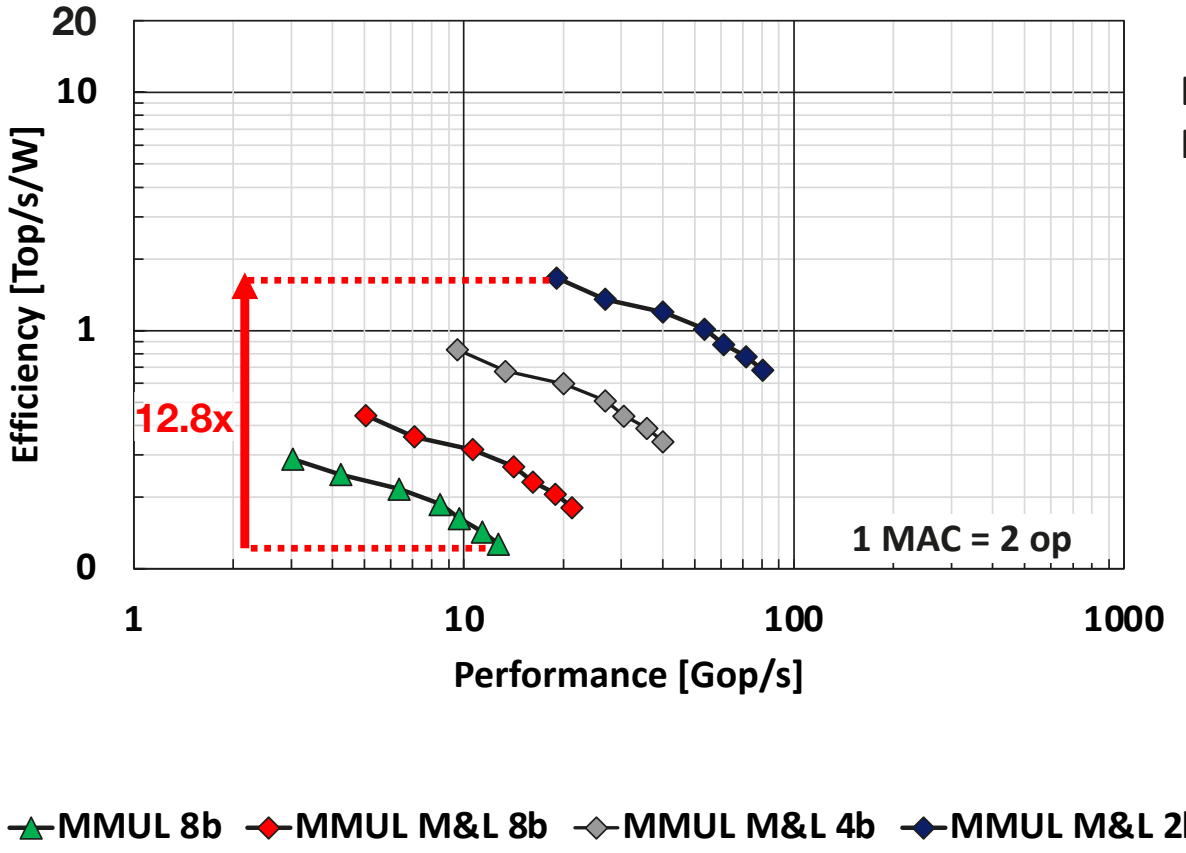
Energy Efficiency Comparison



Baseline efficiency RISC-V cores with MAC&Load extension

- 130 Gop/s/W @ 0.8V -> 290 Gop/s/W @ 0.5V

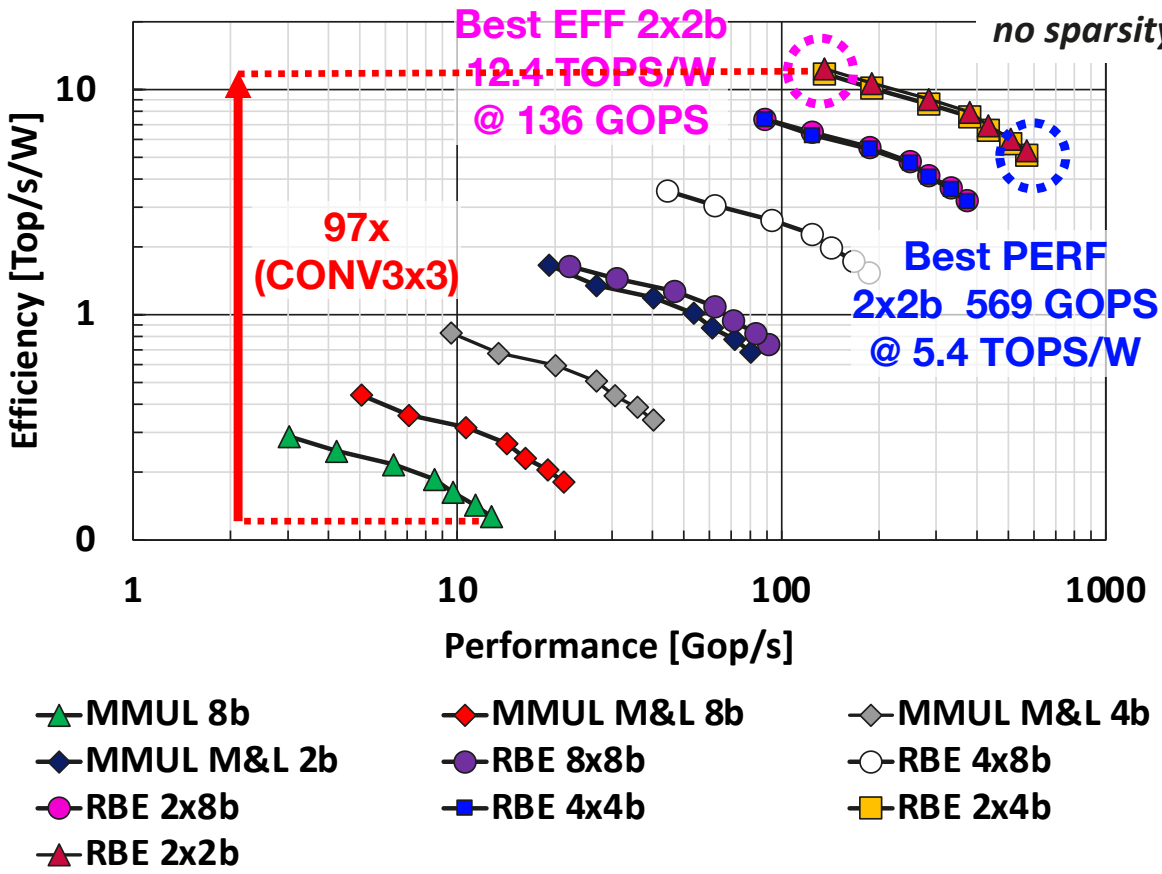
Energy Efficiency Comparison



Baseline efficiency RISC-V cores with MAC&Load extension

- 130 Gop/s/W @ 0.8V -> 290 Gop/s/W @ 0.5V
- M&L and quantization down to 2b bring up to 12.8x improvement in efficiency

Energy Efficiency Comparison



HW Workload: Conv3x3
32x16x3x3
no sparsity

Baseline efficiency RISC-V cores with MAC&Load extension

- 130 Gop/s/W @ 0.8V -> 290 Gop/s/W @ 0.5V
- M&L and quantization down to 2b bring up to 12.8x improvement in efficiency
- Employing RBE on CONV3x3 kernels efficiency can be improved by a further 7.6x, up to 12.4Top/s/W

Thank you

Chips-IT