

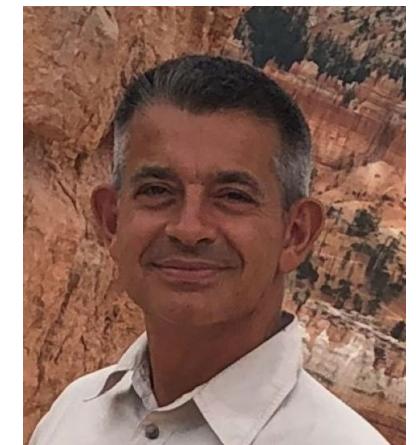
PULP PLATFORM

Open Source Hardware, the way it should be!

# Mempools: The Rise of Tightly-Coupled Processor Clusters

Luca Benini

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



European Research Council



EuroHPC  
Joint Undertaking



FNSNF

FONDS NATIONAL SUISSE  
SCHWEIZERISCHER NATIONALFONDS  
FONDO NAZIONALE SVIZZERO  
SWISS NATIONAL SCIENCE FOUNDATION

**ETH** zürich



<http://pulp-platform.org>



@pulp\_platform

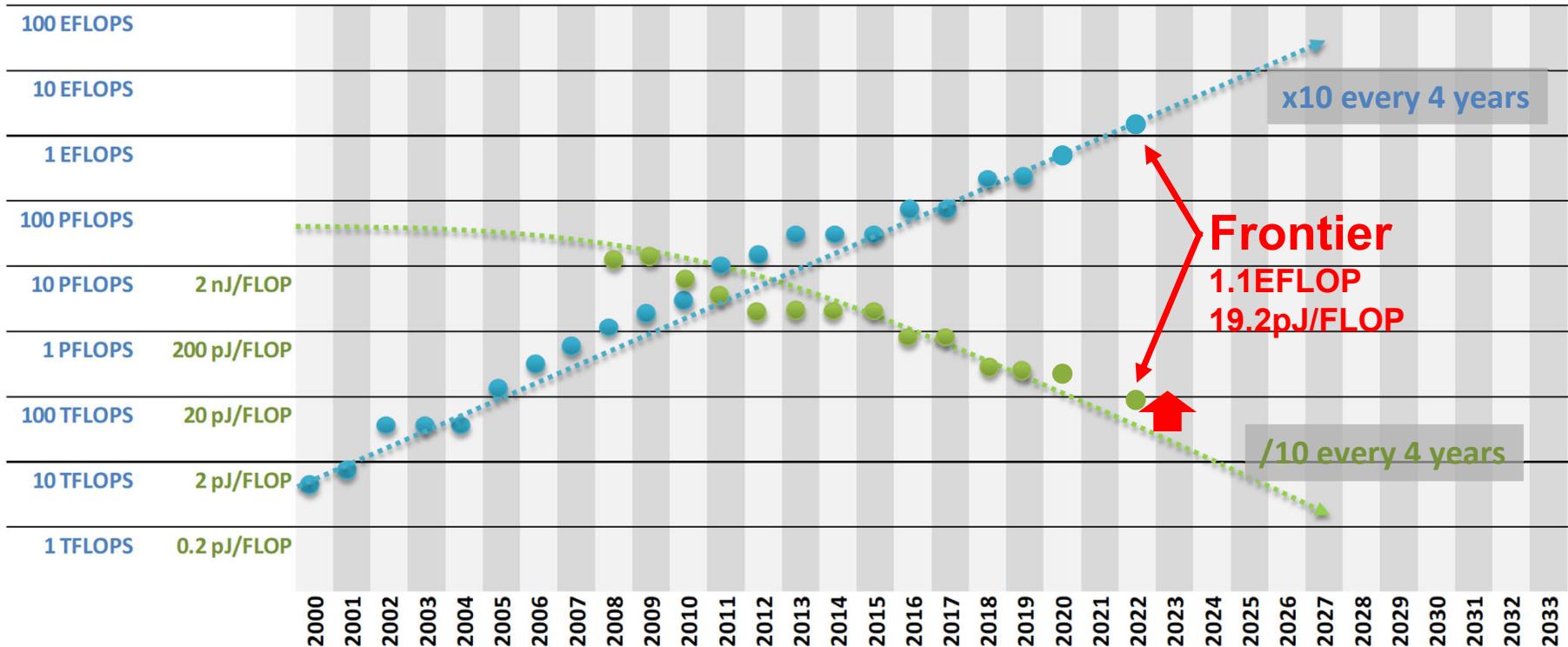


[https://www.youtube.com/pulp\\_platform](https://www.youtube.com/pulp_platform)

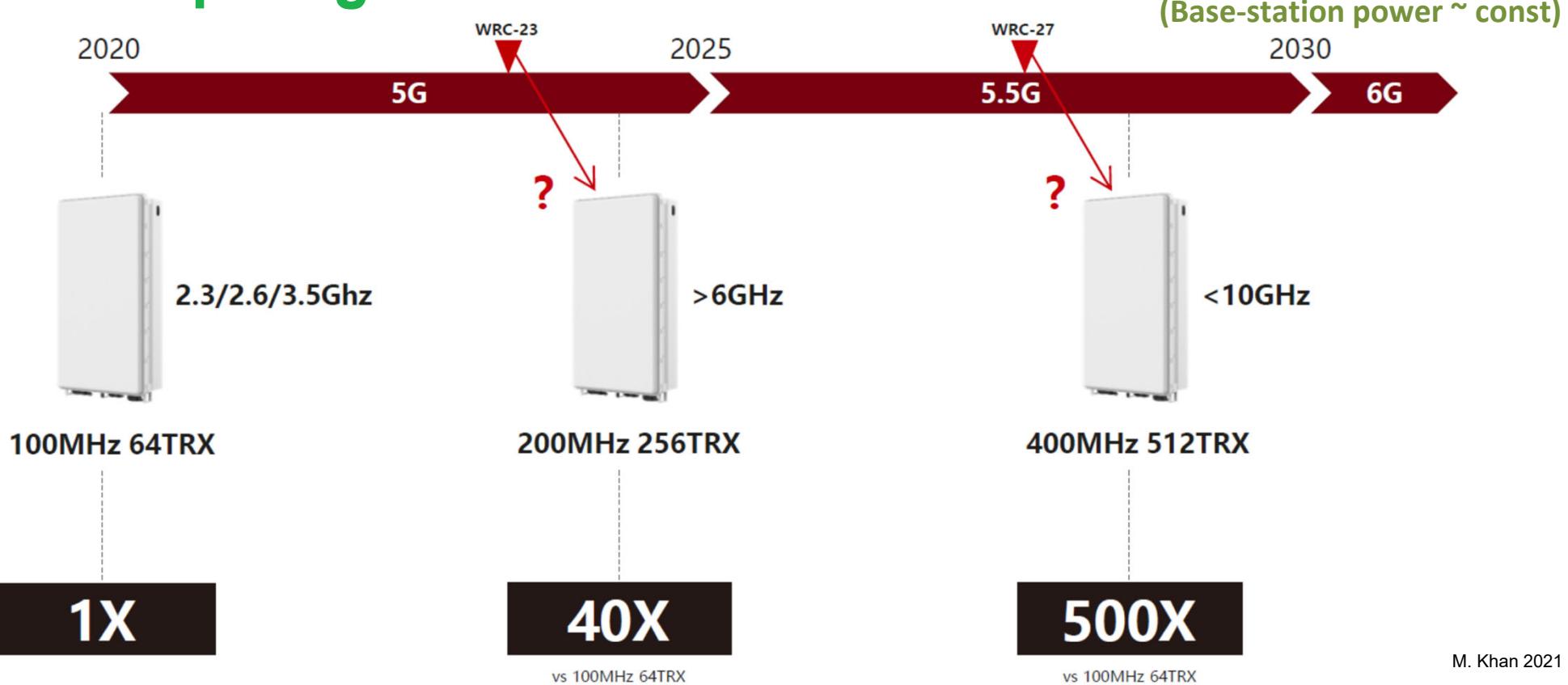
# Computing is Power Bound: HPC

HPC Performance

Copyright © European Processor Initiative 2019. EPI Tutorial/Barcelona/17-07-2019



# Computing is Power Bound: 5G/6G



Baseband station processing: 10x every 3 years!

# Computing is Power Bound: ML

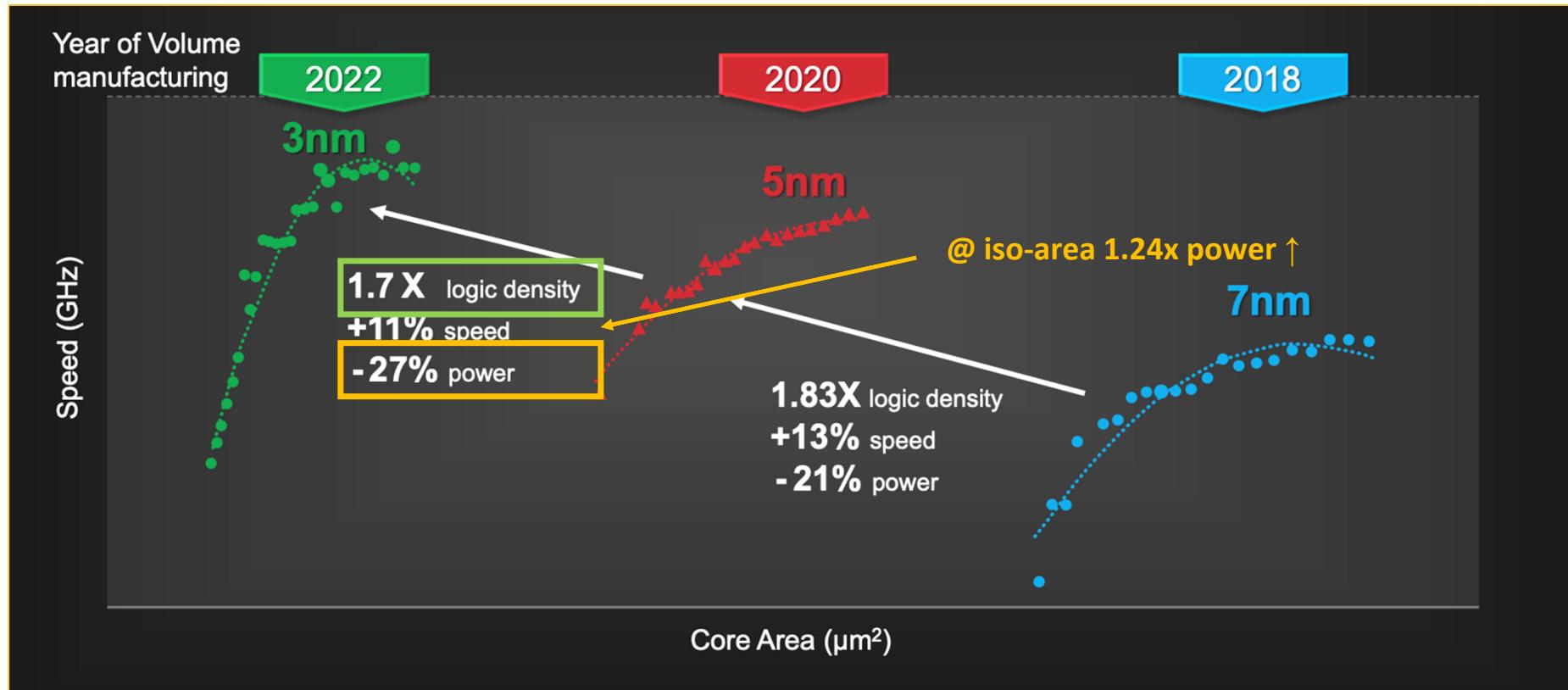


Sevilla 22: arXiv:2202.05924

Machine Learning (training): 10x every 2 years

# Technology Scaling?

TSMC, ISSCC21

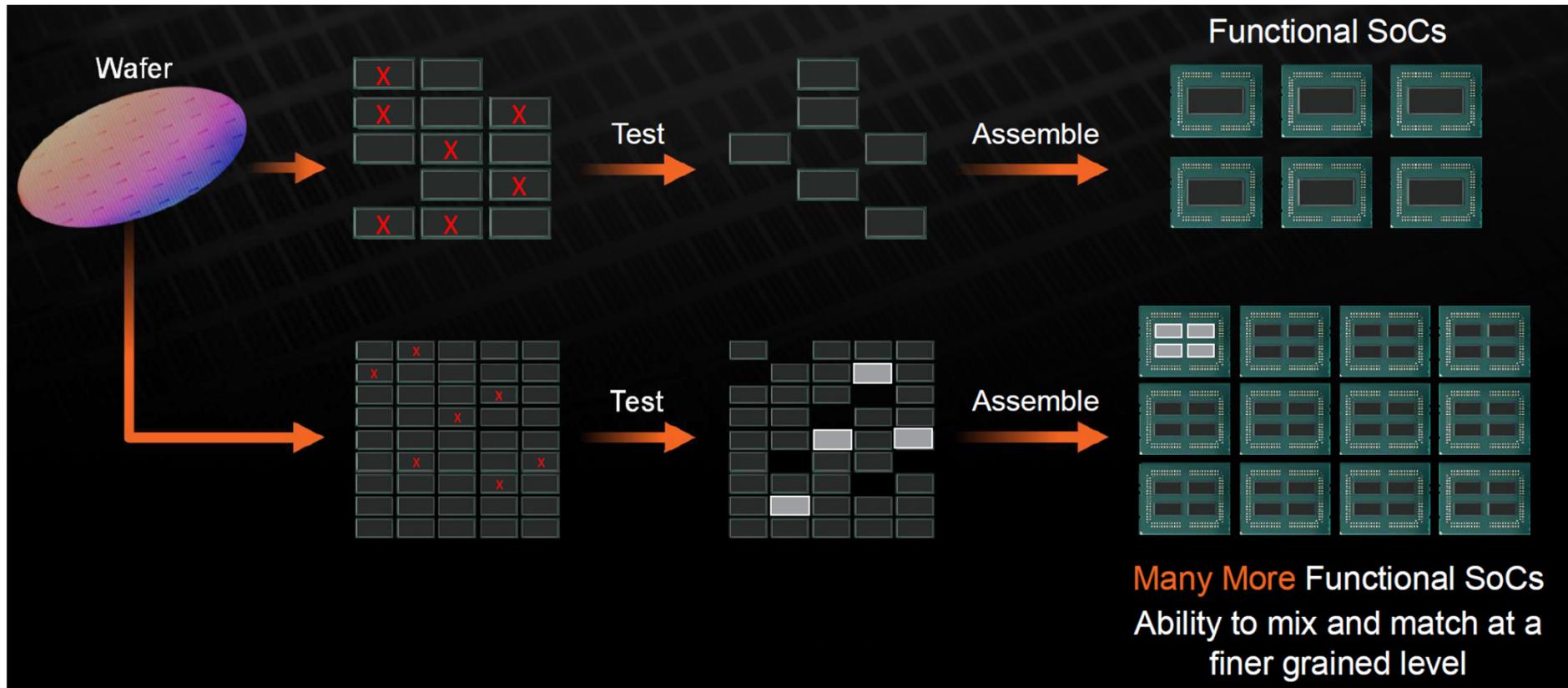


Energy Efficiency ( $\frac{1}{\text{Power}\cdot\text{Time}}$ ) → 10x every 12 years...



# Chiplets?

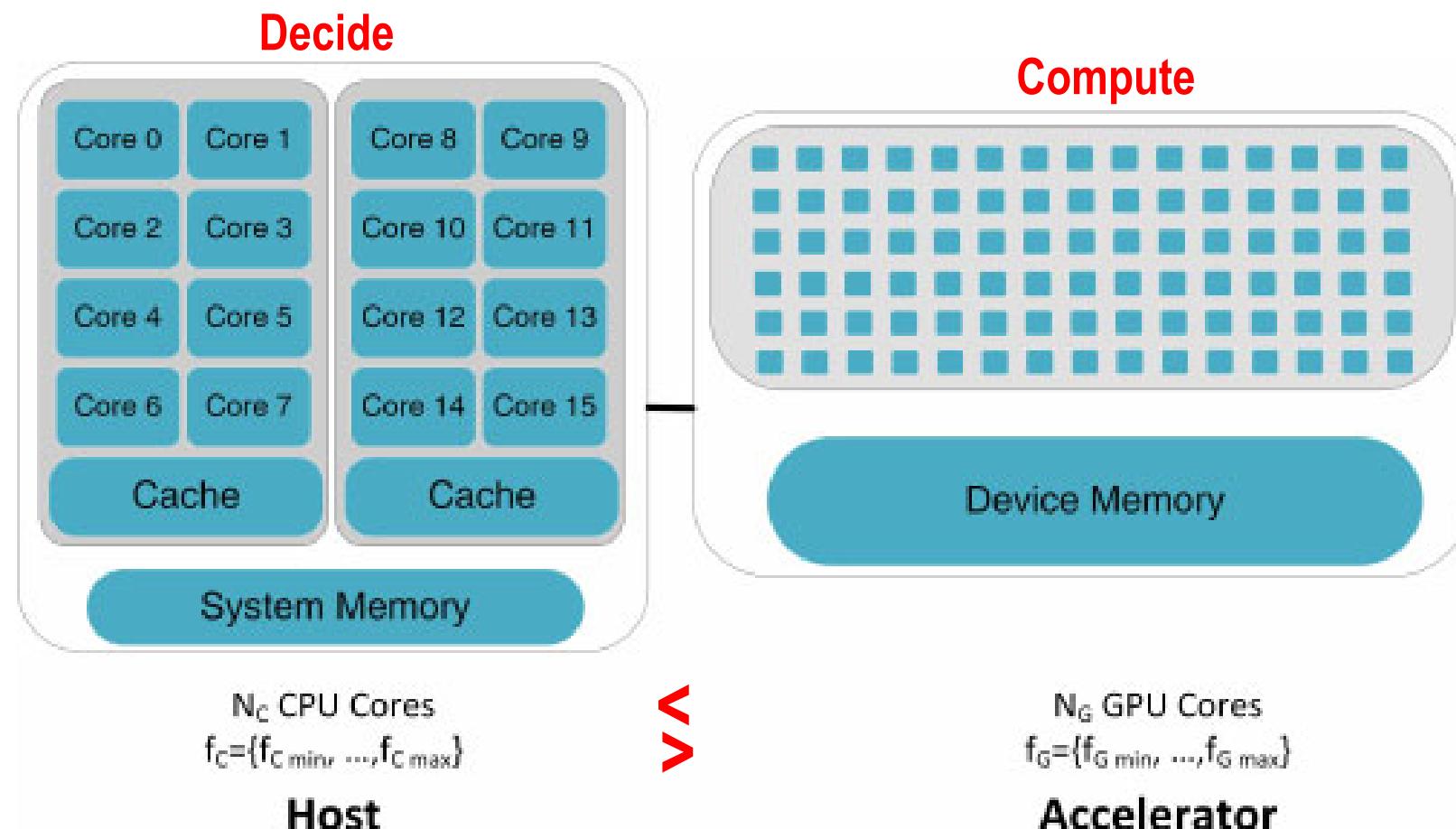
AMD, ISCAS22



Good news for Cost+Density, no silver bullet for Energy



# Efficient Architecture: Heterogeneous+Parallel



# Heterogeneous + Parallel... Why?

- Processors can do two kinds of useful work:

**Decide** (jump to different program part)

- Modulate flow of **instructions**
- Mostly sequential decisions:
  - Don't work too much
  - Be clever about the battles you pick (latency is king)
- Lots of decisions**  
Little number crunching

**Compute** (plough through numbers)

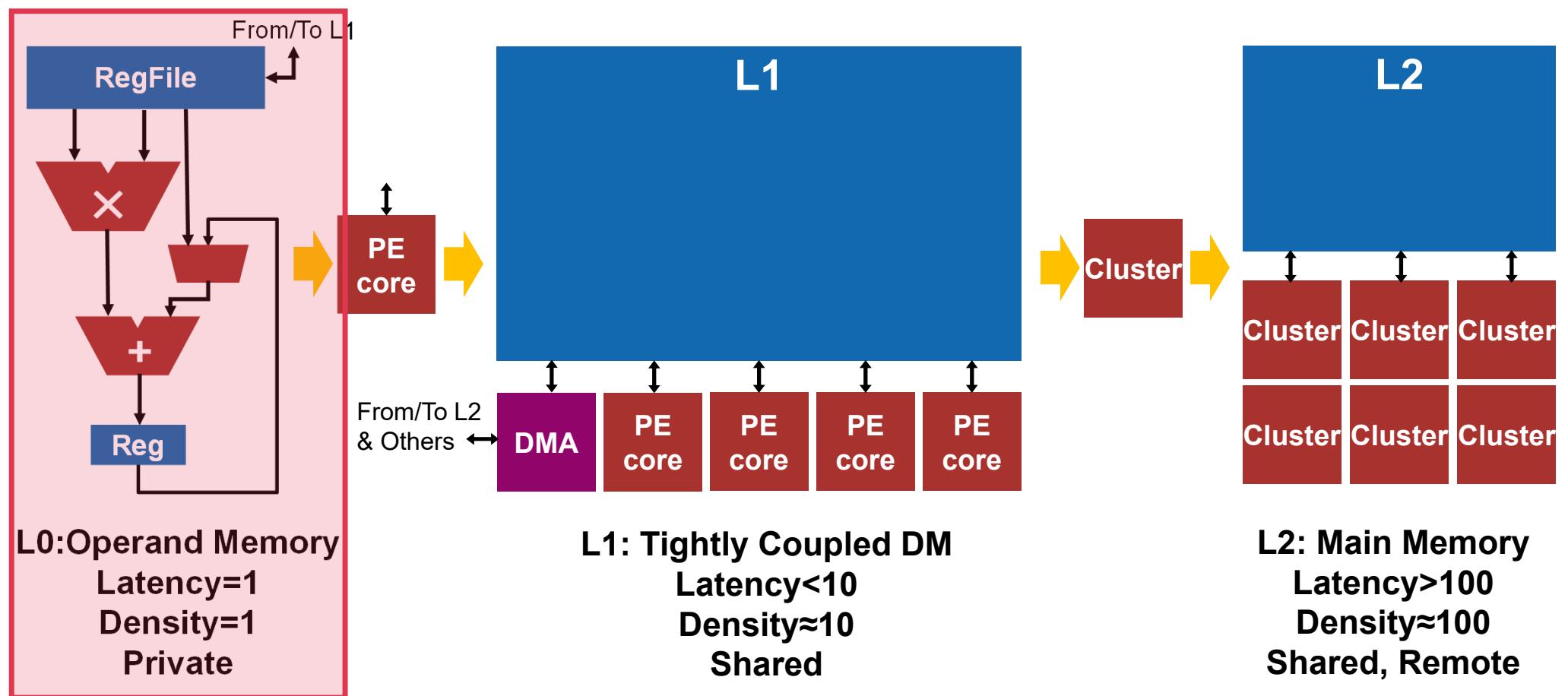
- Modulate flow of **data**
- Embarassing data parallel:
  - Don't think too much
  - Plough through the data (throughput is king)
- Few decisions**  
**Lots of number crunching**

- Today's workloads are dominated by “Compute”:

- Tons of data, few (as fast as possible) decisions based on the computed values,
- “Data-Oblivious Algorithms” (ML, or better DNNs are so!)
- Large data footprint + sparsity

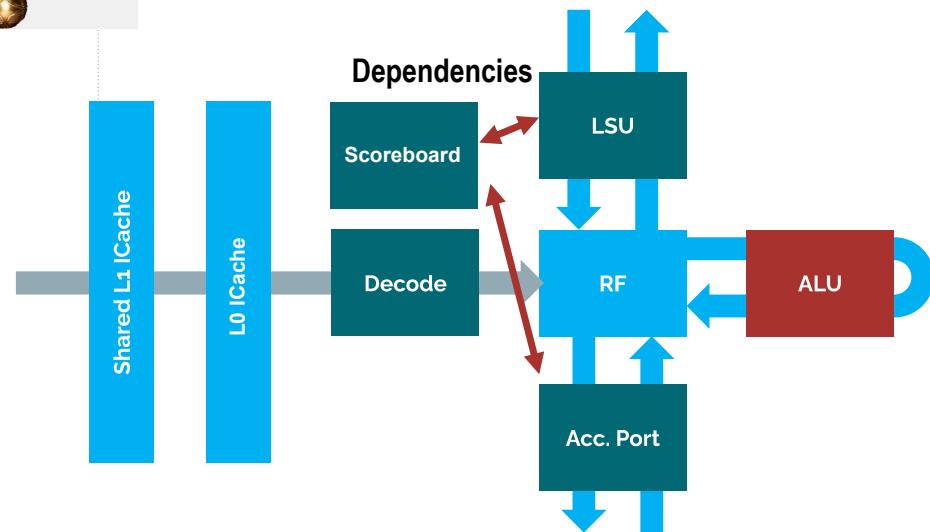
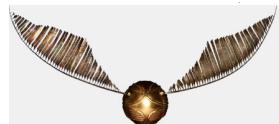
How to design an efficient “Compute” fabric?

# Compute Efficiency for PE, Cluster, SoC



# PE: Snitch, a Tiny RISC-V Control Core

A versatile building block



- **Simplest core: around 20KGE**
  - Speed via simplicity (1GHZ+)
  - L0 Icache/buffer for low energy fetch
  - Shared L1 for instruction reuse (SPMD)
- **Extensible → Accelerator port**
  - Minimal baseline ISA (RISC-V)
  - Extensibility: Performance through ISA extensions (via accelerator port)
- **Latency-tolerant → Scoreboard**
  - Tracks instruction dependencies
  - Much simpler than OOO support!

# Snitch PE: ISA Extension for efficient “Compute”

- How can we remove the Von Neumann Bottleneck?
- Targeting “compute” code

```
double sum = 0;  
for (int i = 0; i < N; ++i) {  
    sum += A[i] * B[i];  
}
```

fld	ft0, 0(a1)	70 pJ
fld	ft1, 0(a2)	70 pJ
addi	a1, a1, 8	50 pJ
addi	a2, a2, 8	50 pJ
fmadd.d	fa0, ft0, ft1, fa0	80 pJ
bne	a1, a3, -5	50 pJ

Memory access, operation, iteration control – can we do better?

Note: memory access (>1 cycle even for L1) → need latency tolerance for LD/ST

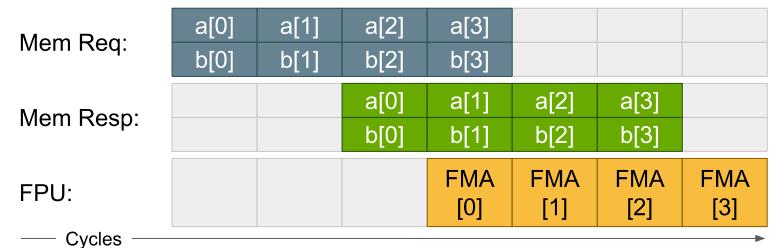
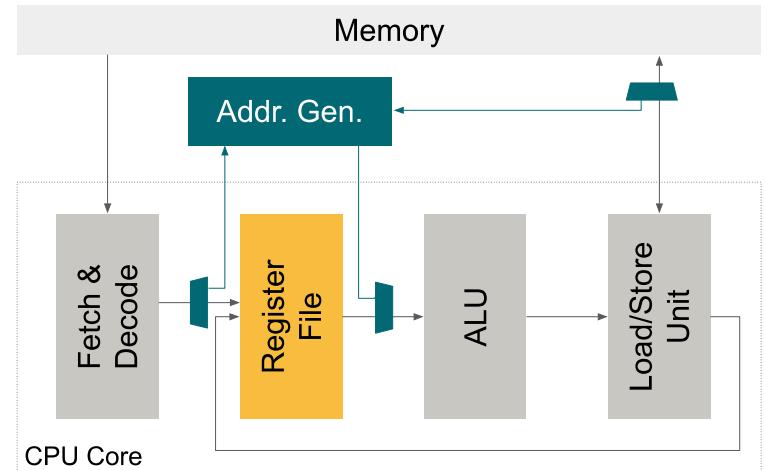
# Stream Semantic Registers

## LD/ST elision

- Intuition: High FPU utilization  $\approx$  high energy-efficiency
- Idea: Turn register read/writes into implicit memory loads/stores.
- Extension around the core's register file
- Address generation hardware

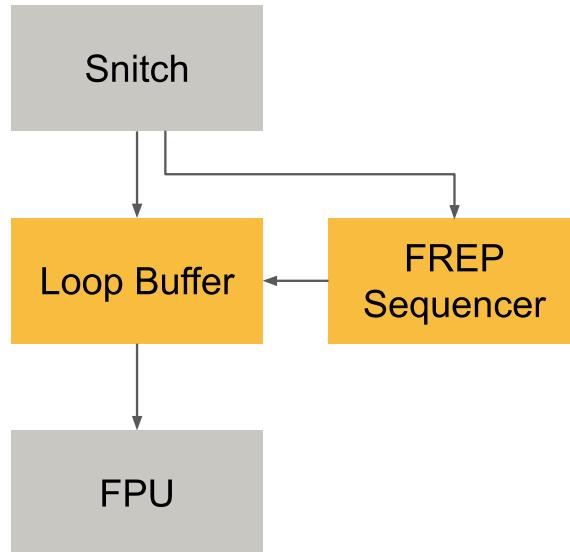
```
loop:  
  f1d r0, %[a]    → scfg 0, %[a], ldA  
  f1d r1, %[b]    → scfg 1, %[b], ldB  
  fmadd r2, r0, r1  
  fmadd r2, ssr0, ssr1
```

- Increase FPU/ALU utilization by  $\sim 3x$  up to 100%
- SSRs  $\neq$  memory operands
  - Perfect prefetching, latency-tolerant



# Floating-point Repetition Buffer

Remove control flow overhead in compute stream



- Programmable micro-loop buffer
- Sequencer steps through the buffer, independently of the FPU
- Integer core free to operate in parallel:  
*Pseudo-dual issue*
- High area- and energy-efficiency

```
mv r0, zero
loop:
  addi r0, 1
  fmadd r2, ssr0, ssr1
  bne r0, r1, loop
```

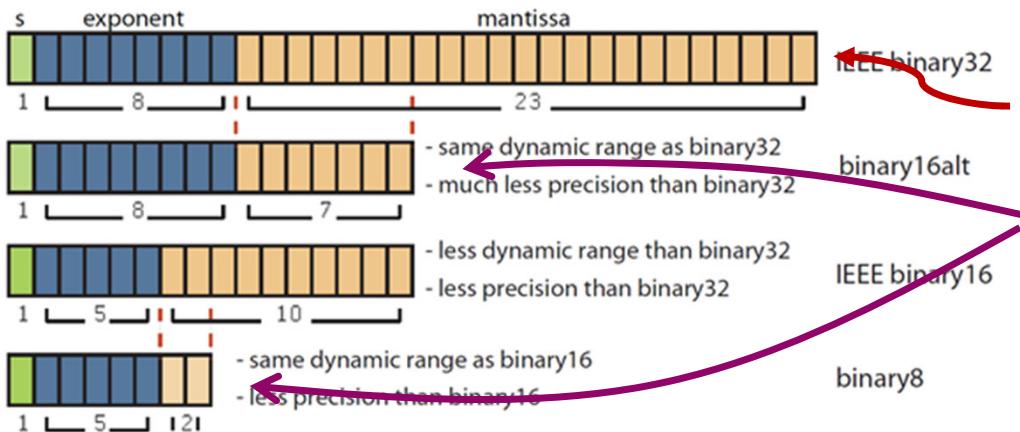
→

```
loop:
  frep r1, 1
  fmadd r2, ssr0, ssr1
```

# RISC-V ISA Extension for Target Workload

Mixed precision

Efficient DNN inference & training



Inference ≠ Training Quantization

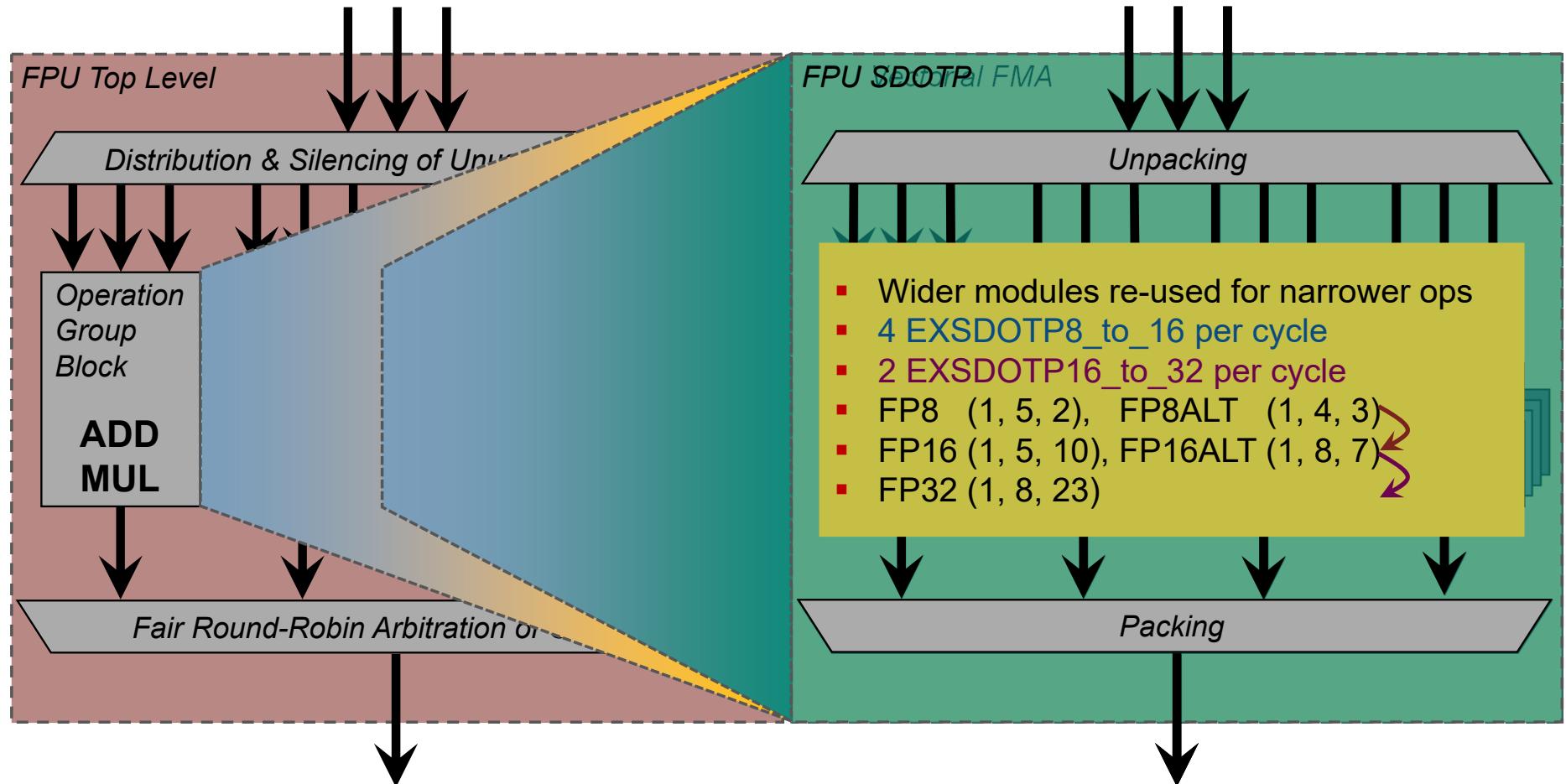
- Inference: INT8 quantization is SoA
- Training: High dynamic range needed for weights and weight updates

**fp32** is still standard for DNN training workloads. Low precision training with **bf16** and **fp8**

Support a wide variety of FP formats and instructions:

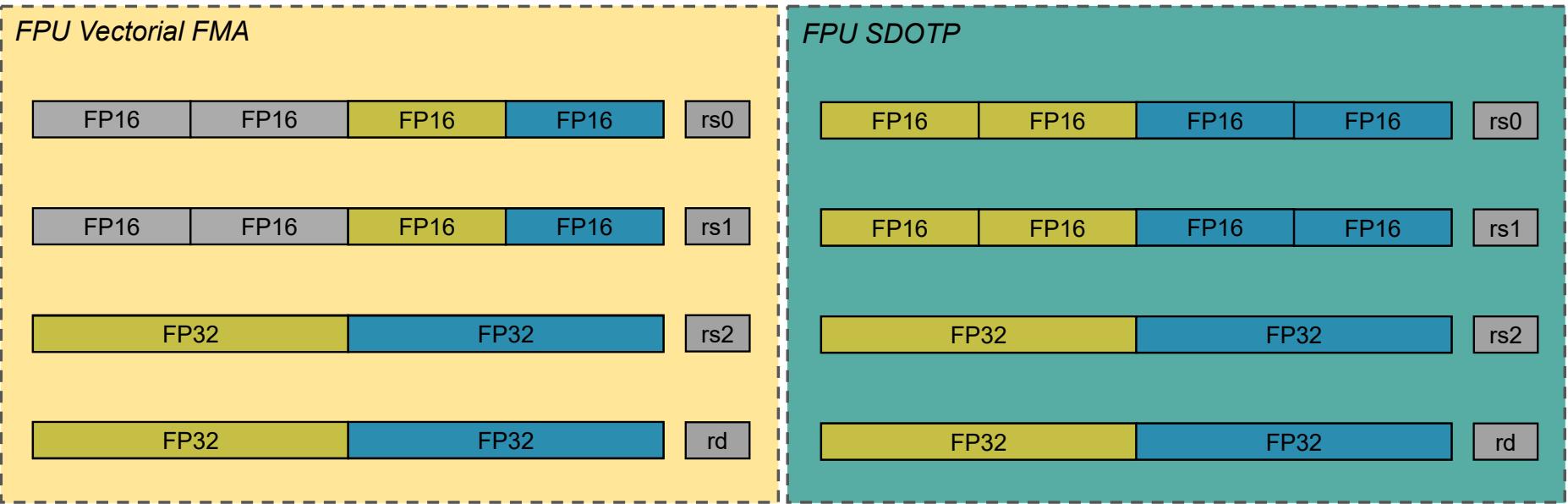
- Standard: **fp64**, **fp32**, **fp16**, **bf16**
- Low precision: **fp8**, **altfp8**
  - **fp8** (1-4-3): forward prop.
  - **altfp8** (1-5-2): backward prop.
  - Exp. ops: accumulation

# FPU and SDOTP unit



# EXFMA vs EXSDOTP

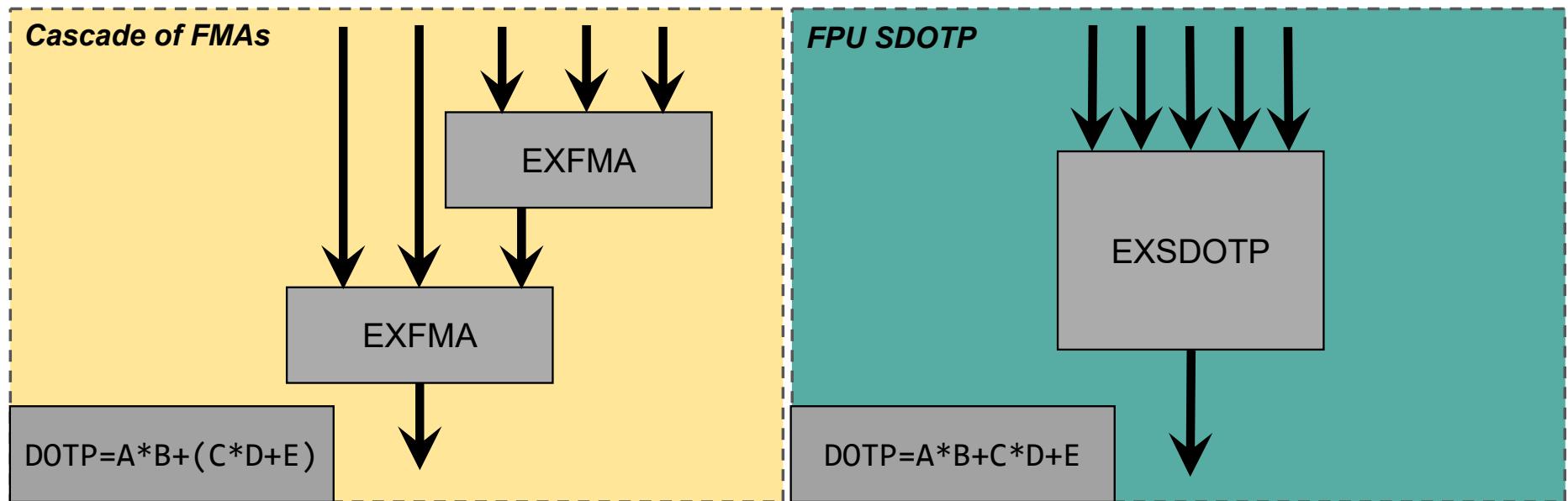
Expanding Dot Product w. accumulation



- $rd_{32} = rs0_{16} * rs1_{16} + rs2_{32}$
- Vectorial Expanding FMA: **Unbalanced**
- Consumes only half of rs0, rs1 and produces the whole rd

- $rd_{32} = rs0_{0,16} * rs1_{0,16} + rs0_{1,16} * rs1_{1,16} + rs2_{32}$
- Vectorial Expanding SDOTP: **Balanced**
- The whole rs0, rs1, rs2, rd are used

# Cascade of EXFMAs vs EXSDOTP



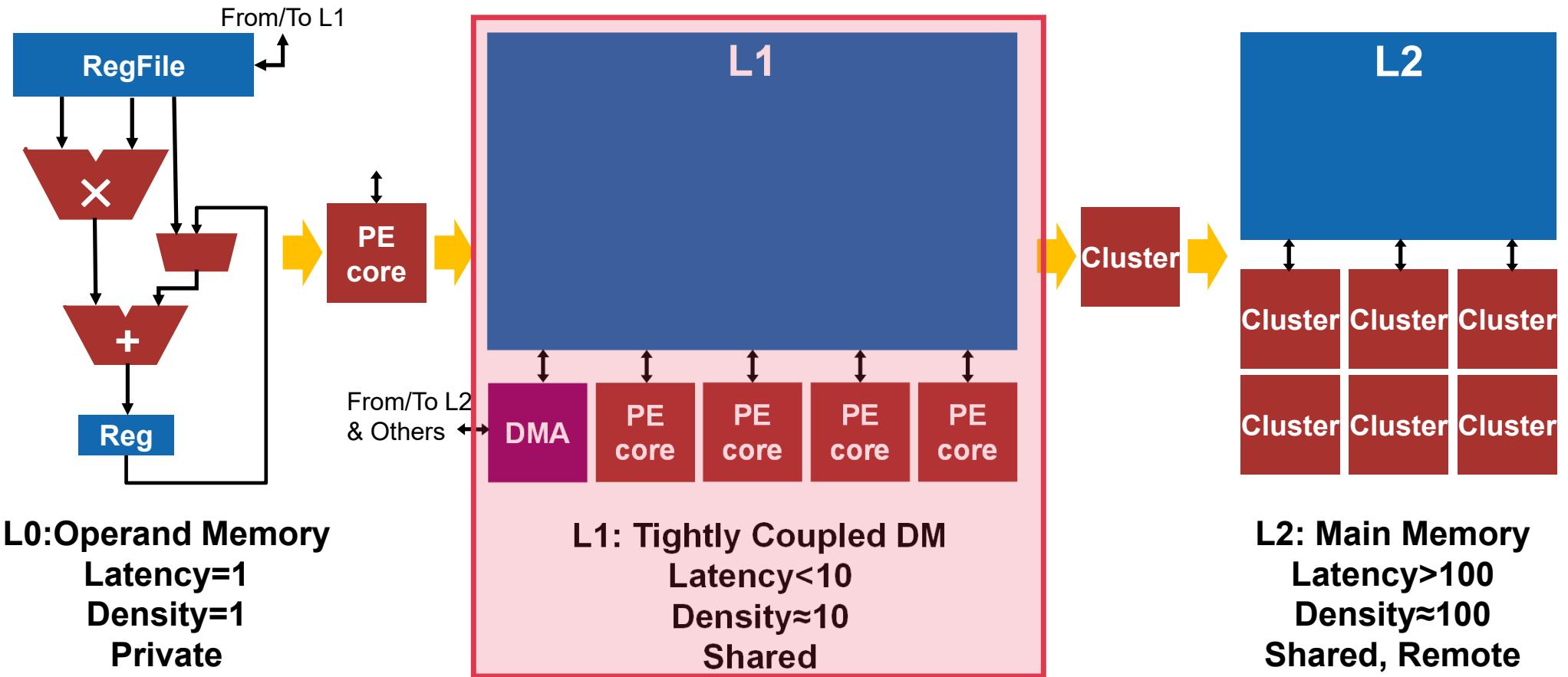
Non-distributive FP addition → Precision Loss

- **Fused** EXSDOTP (i.e. lossless)
- Single normalization and rounding step
- **Smaller** area and **shorter critical path**
- Product by-pass to compute **fused three-term addition** (vector inner sum)

# Efficient PE architecture in perspective

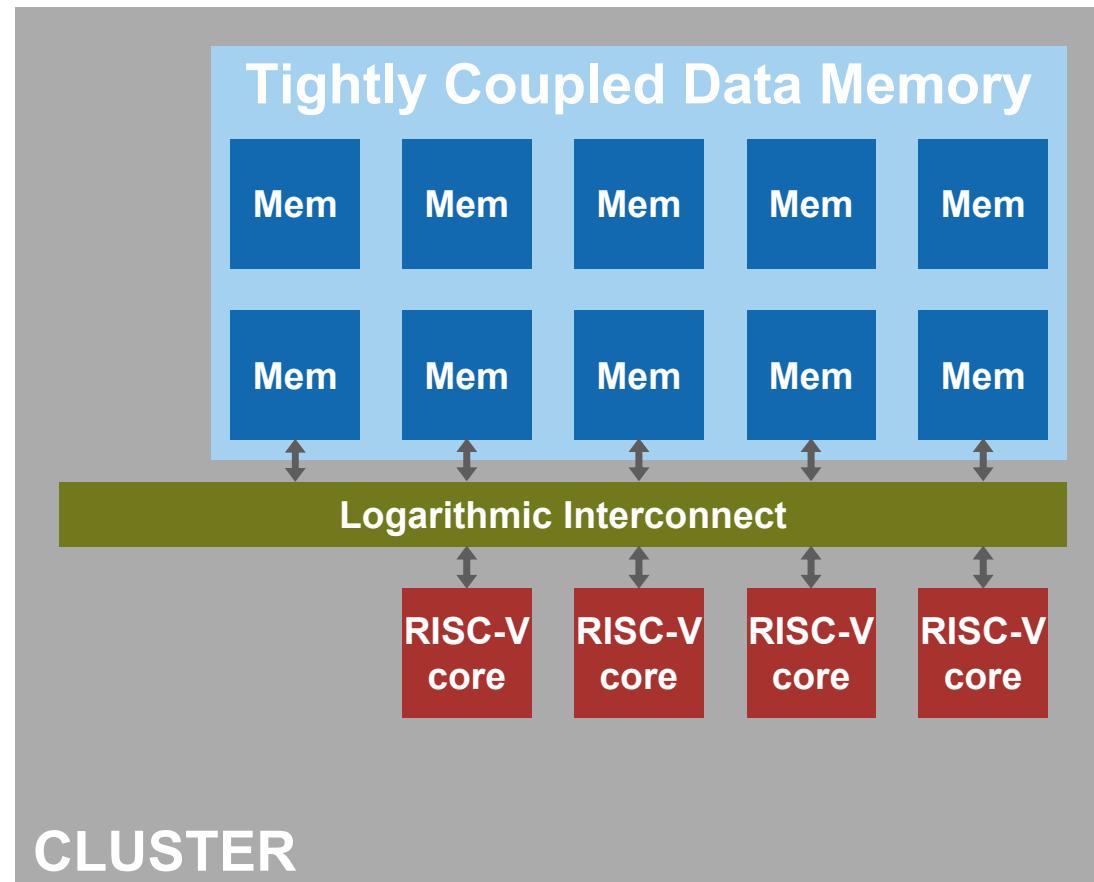
1. Minimize control overhead → Simple, shallow pipelines
2. Reduce VNB → amortize IF: SSR-FREP + SIMD (Vector processing)
3. Hide memory latency → non-blocking LD/ST+dependency tracking
4. Highly expressive, domain-specific instructions with SPUs

# Compute Efficiency for PE, Cluster, SoC

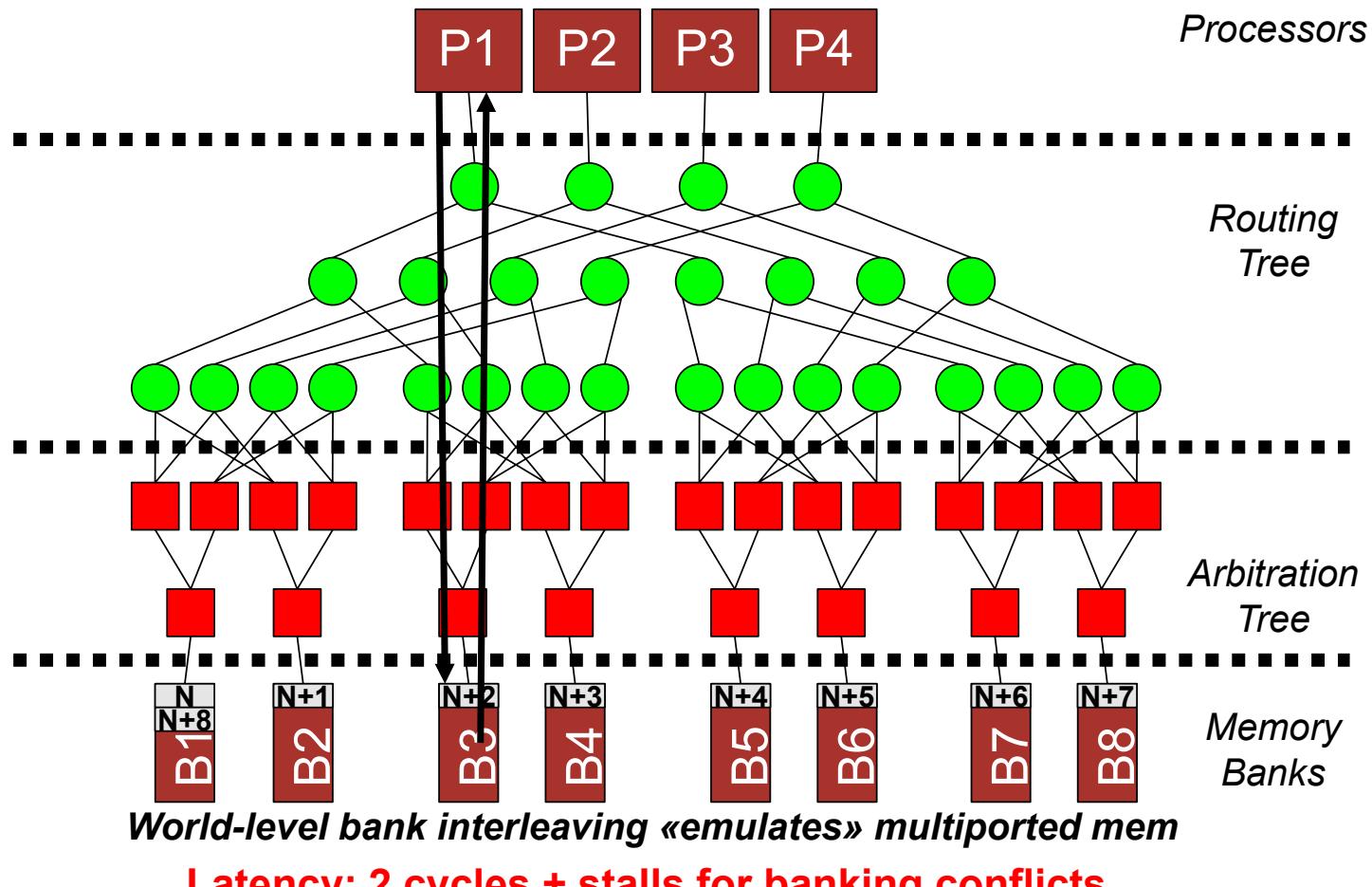


# The Cluster: Design Challenges

- **Low latency access TCDM**
  - Multi-banked architecture
  - Fast logarithmic interconnect
- **Fast synchronization**
  - Atomics
  - Barriers
- **Efficient PE**
  - Hide TCDM “residual” latency
  - Remove Von Neumann Bottleneck

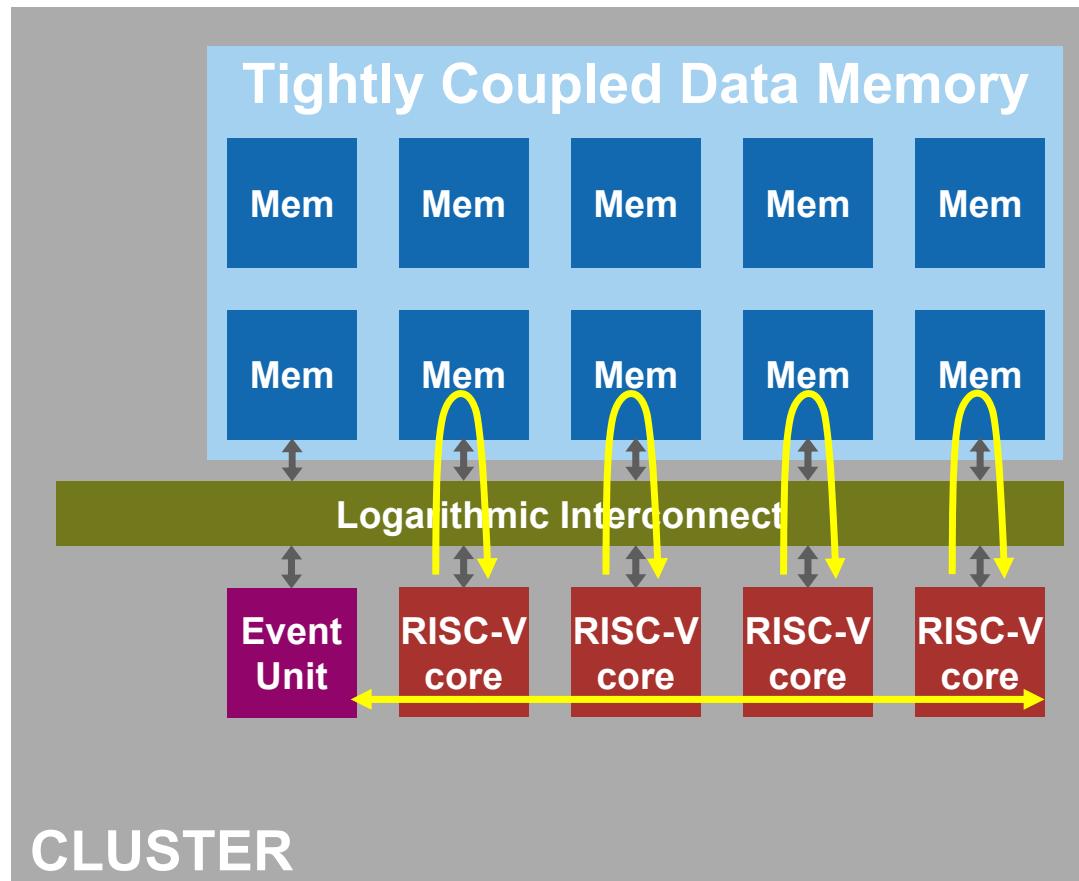


# High speed logarithmic interconnect

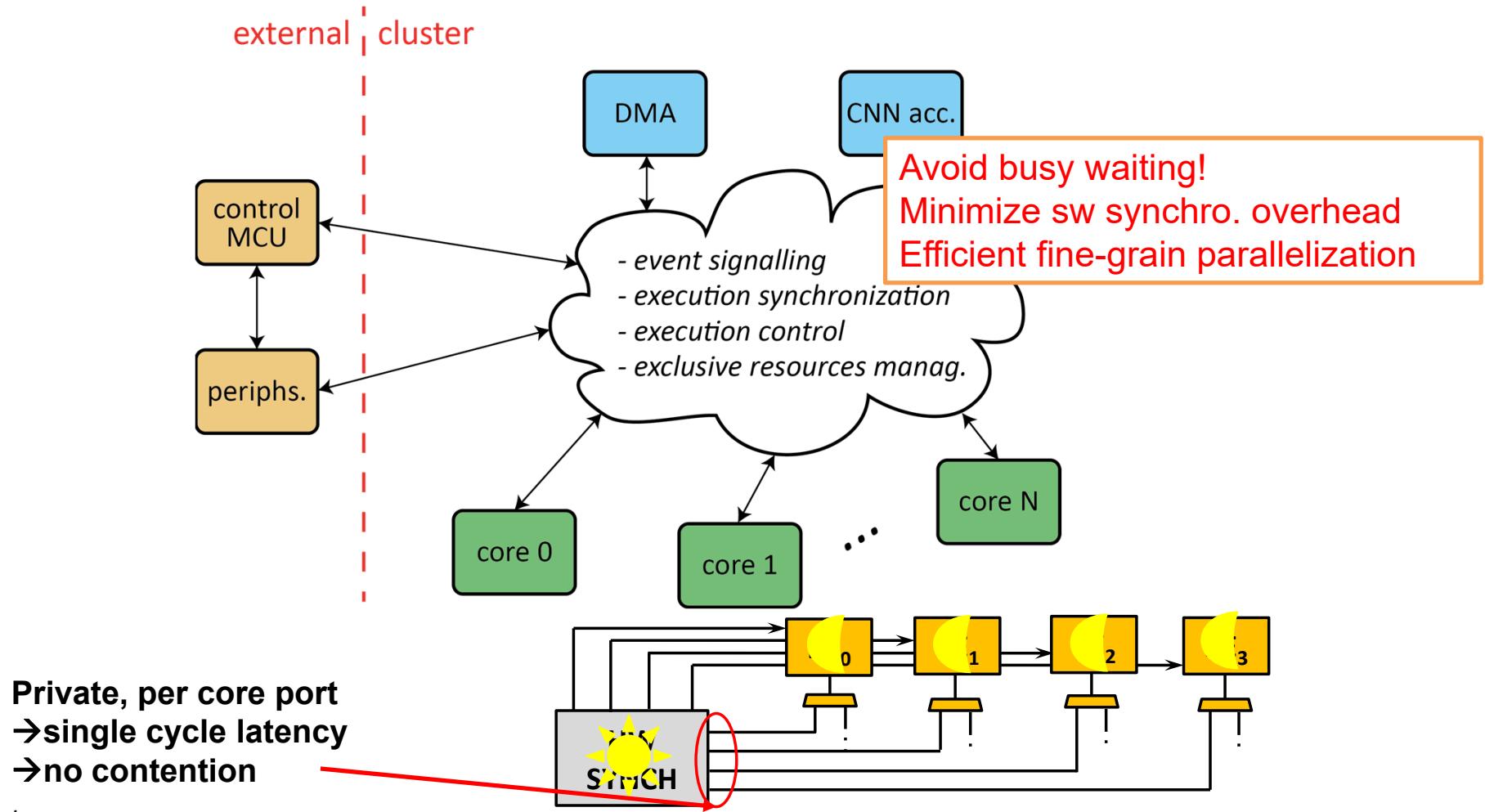


# Fast synchronization and Atomics

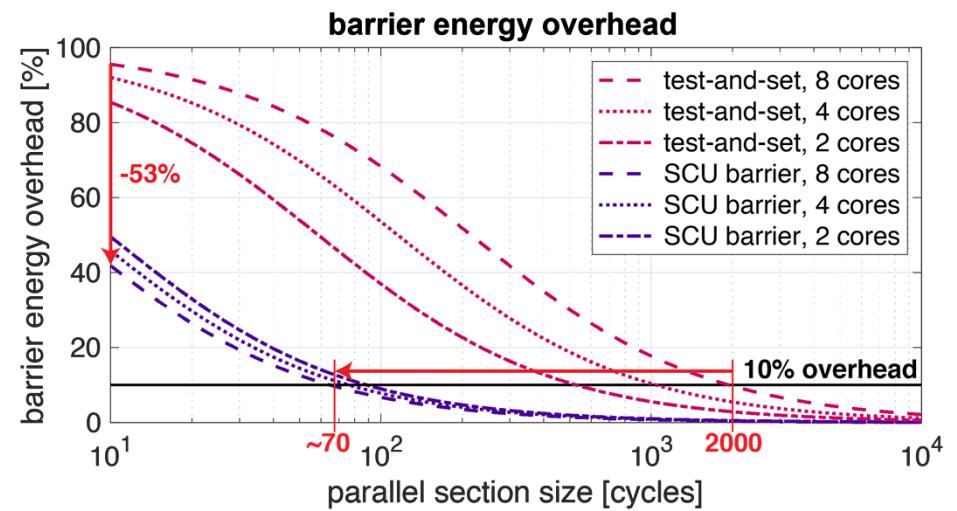
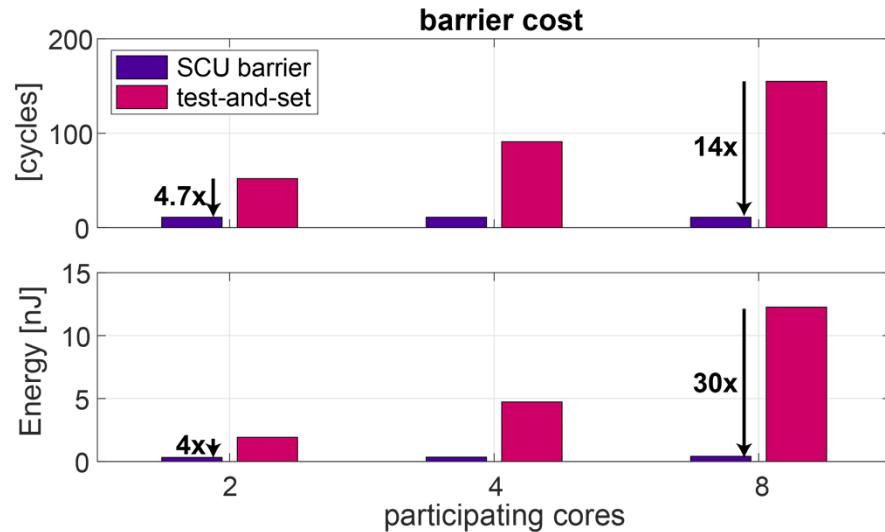
- **Atomic instructions**
  - Leverage fast memory access
  - Based on per-bank atomic adapters
- **Fast synchronization**
  - Atomics for generic primitives
  - Accelerating barriers in HW (make the common case fast)
  - Minimize idle power while waiting



# Hardware-Accelerated, Event-Based Barrier



# Barrier: Results



- 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

# Efficiently Global Data Mover

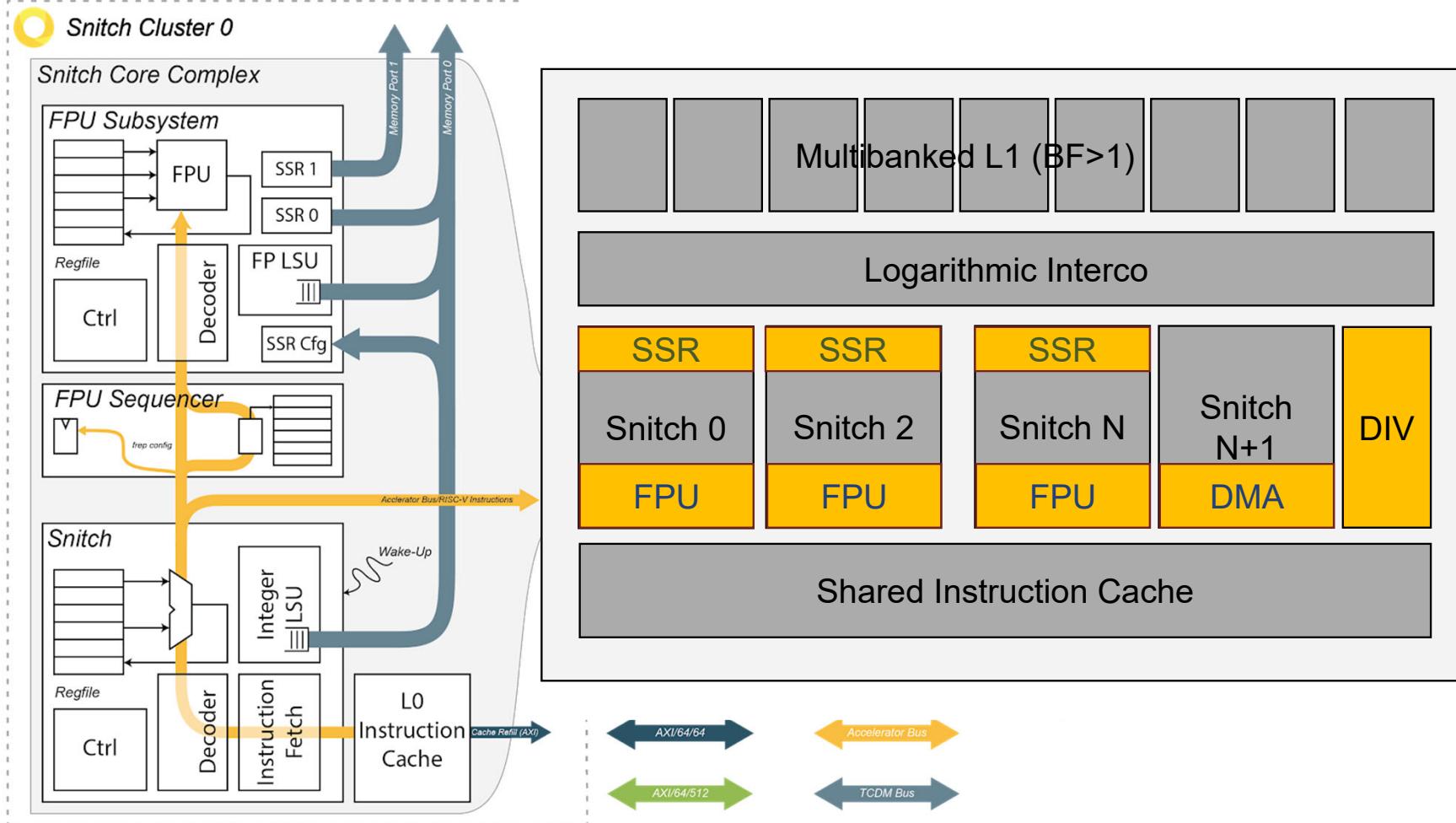
hide L2main memory latency



- 64-bit AXI DMA – explicit double-buffered transfers – **better than D\$**
- Tightly coupled with Snitch (<10 cycles configuration)
- Operates on wide 512-bit data-bus
- Hardware support to copy 2D shapes
- Higher-dimensionality handled by SW
- Intrinsics/library for easy programming

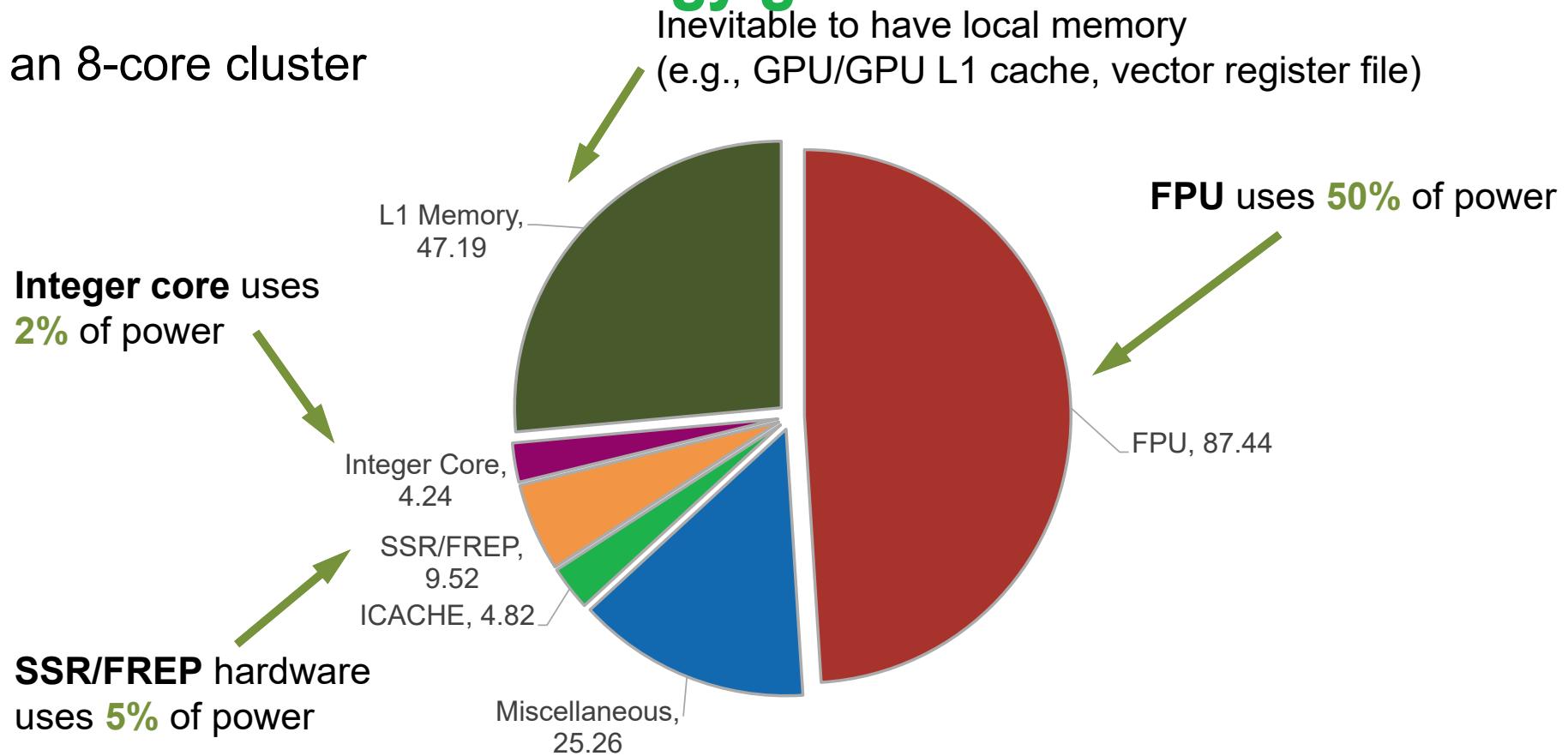
```
// setup and start a 1D transfer, return transfer ID
uint32_t __builtin_sdma_start_oned(
    uint64_t src, uint64_t dst, uint32_t size, uint32_t cfg);
// setup and start a 2D transfer, return transfer ID
uint32_t __builtin_sdma_start_twod(
    uint64_t src, uint64_t dst, uint32_t size,
    uint32_t sstrd, uint32_t dstrd, uint32_t nreps, uint32_t cfg);
// return status of transfer ID tid
uint32_t __builtin_sdma_stat(uint32_t tid);
// wait for DMA to be idle (no transfers ongoing)
void __builtin_sdma_wait_for_idle(void);
```

# Snitch Cluster Architecture



# Where does the Energy go?

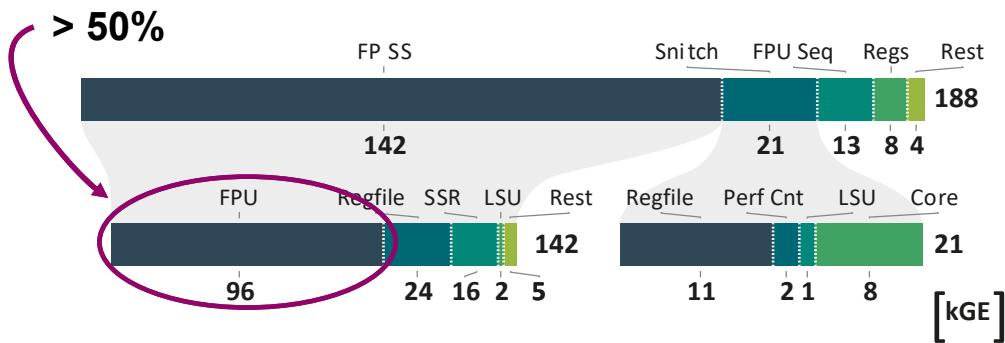
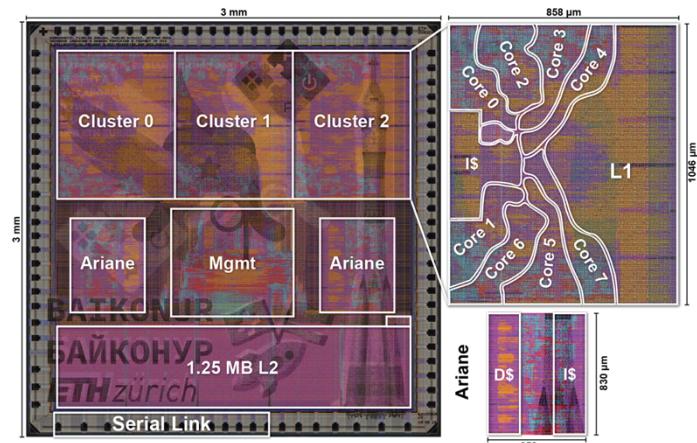
In an 8-core cluster



Spending energy where it contributes to the result → High Efficiency

# Baikonur

The first Snitch-based test chip.



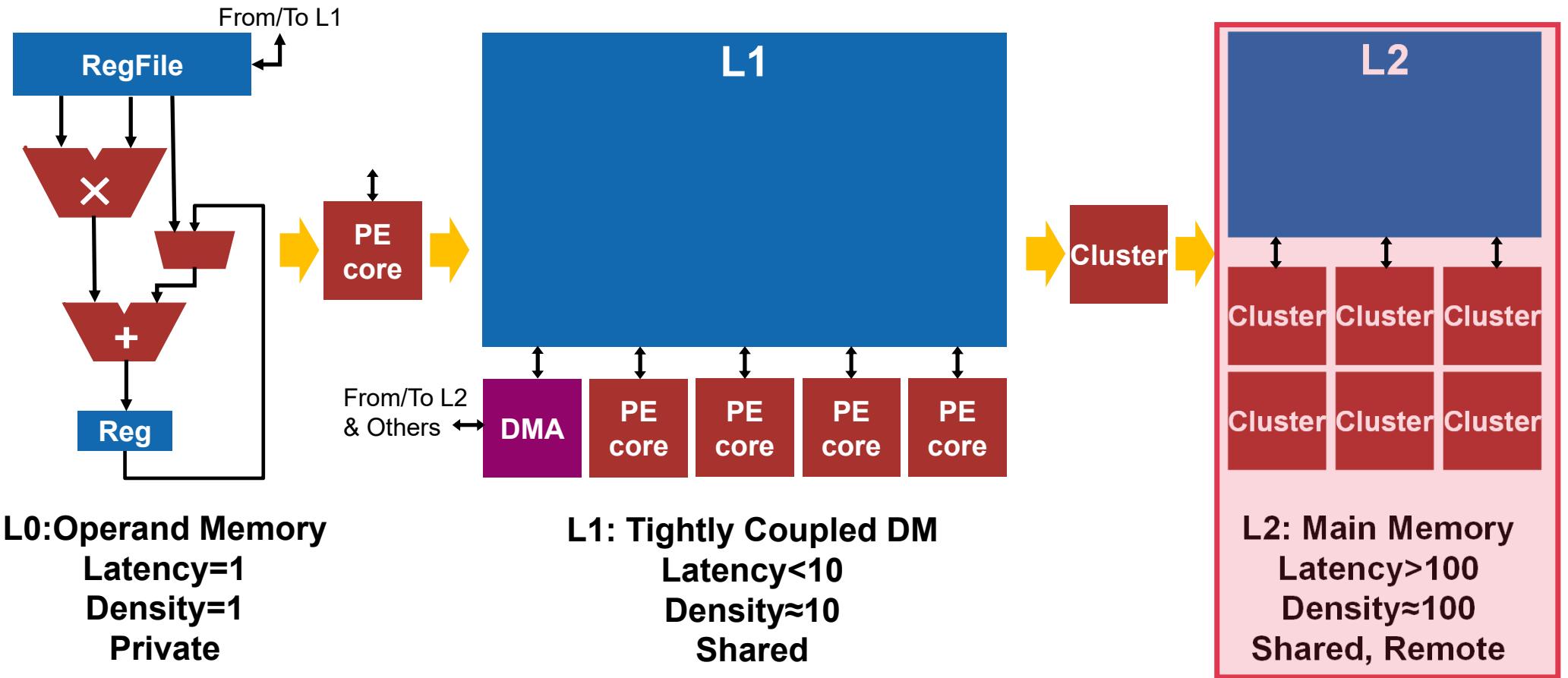
- **Snitch compute cluster:**
  - 8x RV32G Snitch cores
  - 8x large FPUs
  - > 40% energy spent on FPU
  - High energy-efficiency of ~80 Gdpflop/s/W
  - 104 Gspflop/s/W similar to accelerators
- **9 mm<sup>2</sup> prototype in GF 22FDX**
- **Testbed for key architectural components**
  - Snitch octa-core cluster
  - Ariane cores

Very efficient, versatile, compute cluster! How do we scale?

# Efficient Cluster architecture in perspective

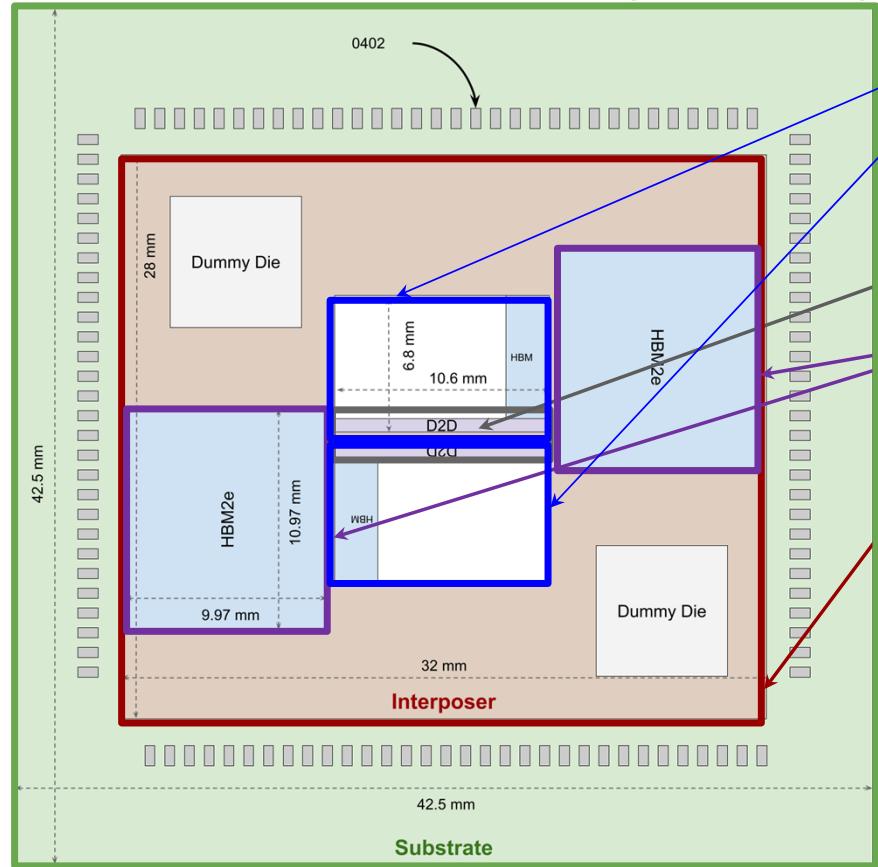
1. Memory pool – efficient sharing of L1 memory
2. Fast and parsimonious synchronization
3. Data Mover + Double buffering – explicitly managed block transfers at the boundary
4. **More cores and more memory per cluster... that would be nice!**

# Compute Efficiency for PE, Cluster, SoC



# Occamy: a 12LPP+ AI & Stream Computing Chiplet

Silicon implementation of all key IPs (core, Phys, chiplet)



**Dual-chiplet configuration in 12LP+**

- Area: ~70mm<sup>2</sup>

**Chip2chip link**

**HBM2e memories**

**Interposer**

- 65nm tech, passive

**Substrate**



**GlobalFoundries™**  
**synopsys®**  
**Rambus**

# From Snitch Cluster

## 8 Snitch compute cores

- Single-stage, small Integer control core

## 9<sup>th</sup> Core: DMA

- 512 bit data interface
- Efficient data movement

## 128 kB TCDM

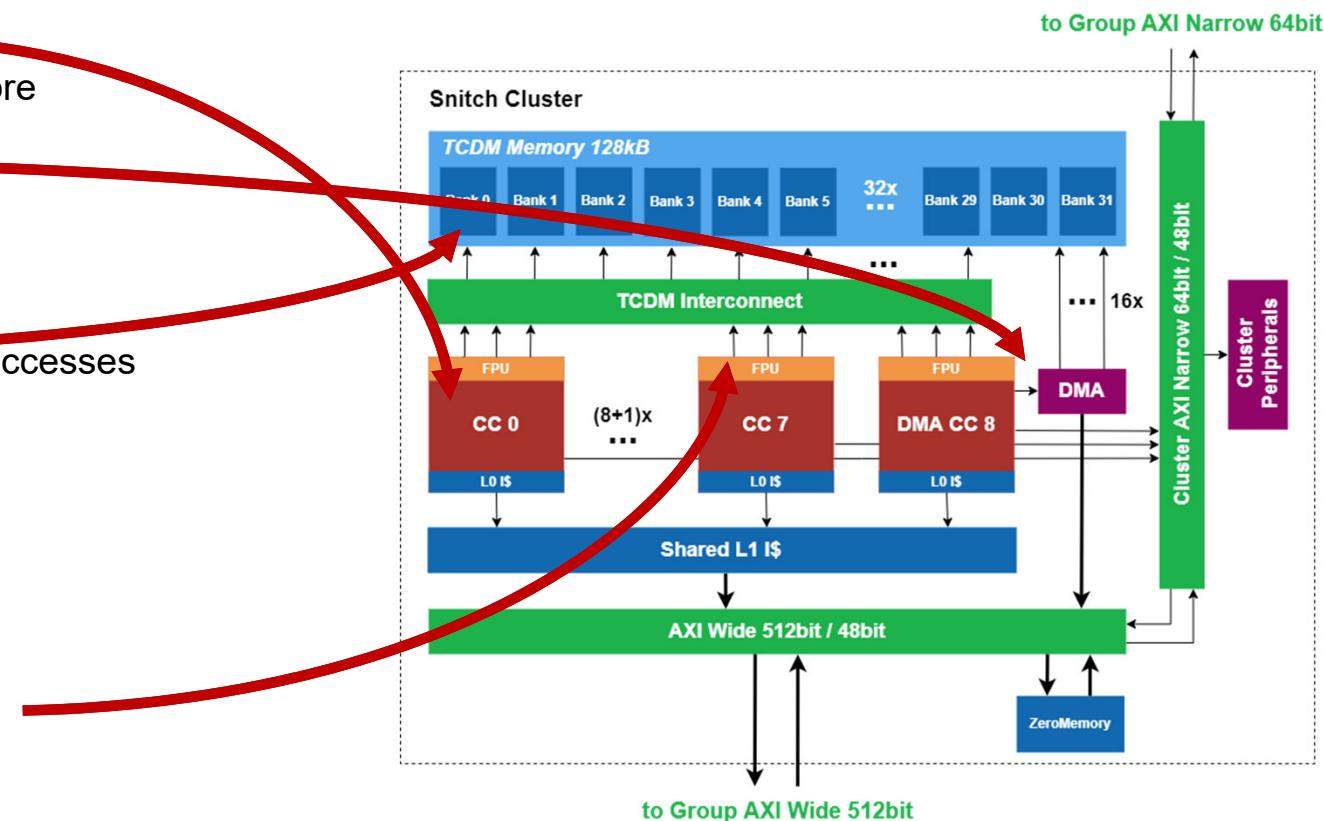
- Scratchpad for predictable memory accesses
- 32 Banks

## Custom ISA extensions

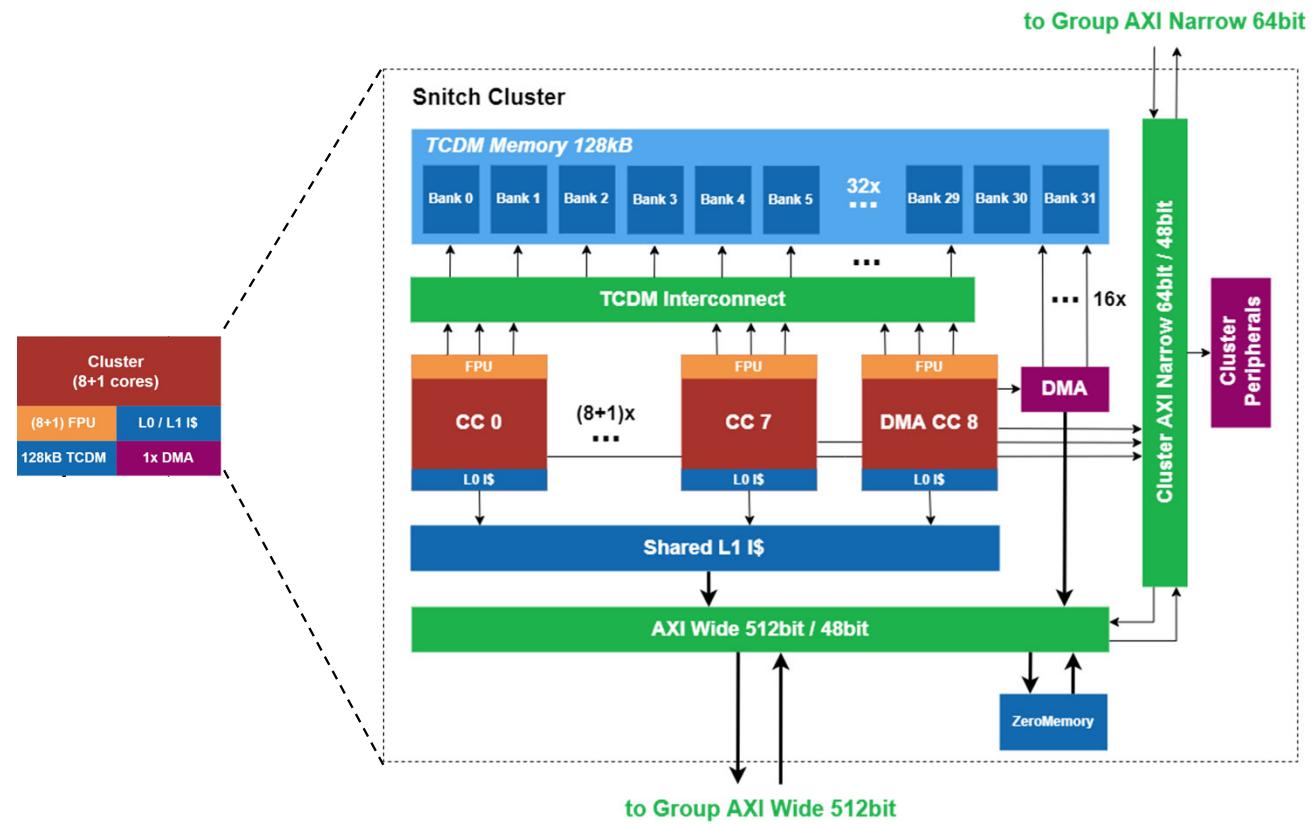
- Xfrep, XSSR
- **New: XISSR sparsity support**

## 1 FPU per Snitch core

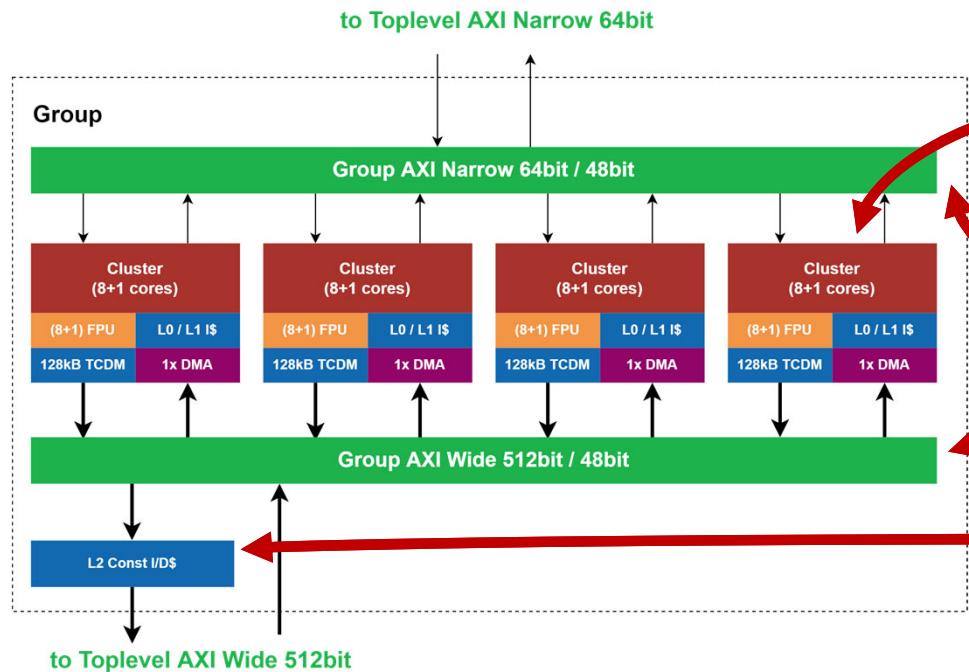
- Decoupled and heavily pipelined
- Multi-format FPU (+SIMD)
- **New: Minifloat support + SDOTP**



# To Snitch Group



# Snitch Group



## 4 Clusters per Group

- Single-stage, small Integer control core

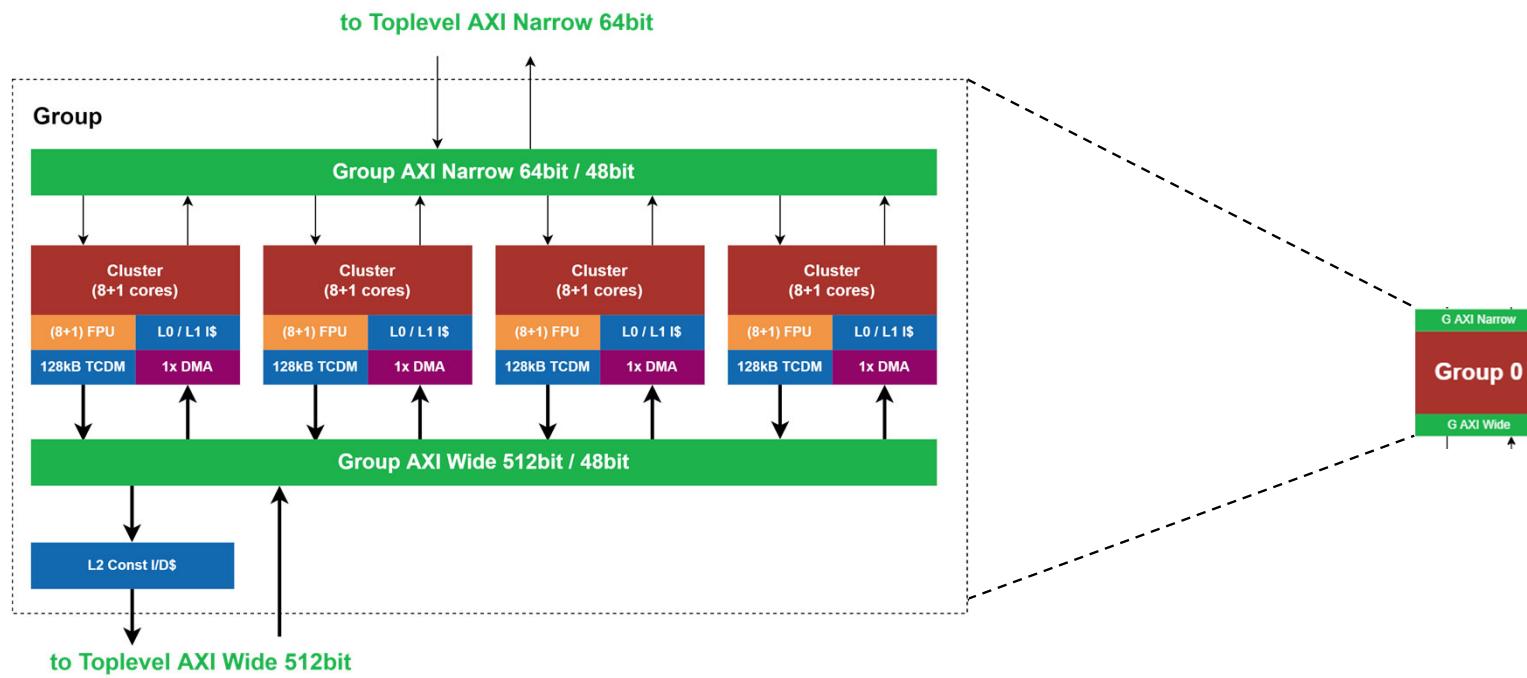
## 2 AXI Busses

- 64-bit narrow interface: config
- 512-bit wide interface: DMA

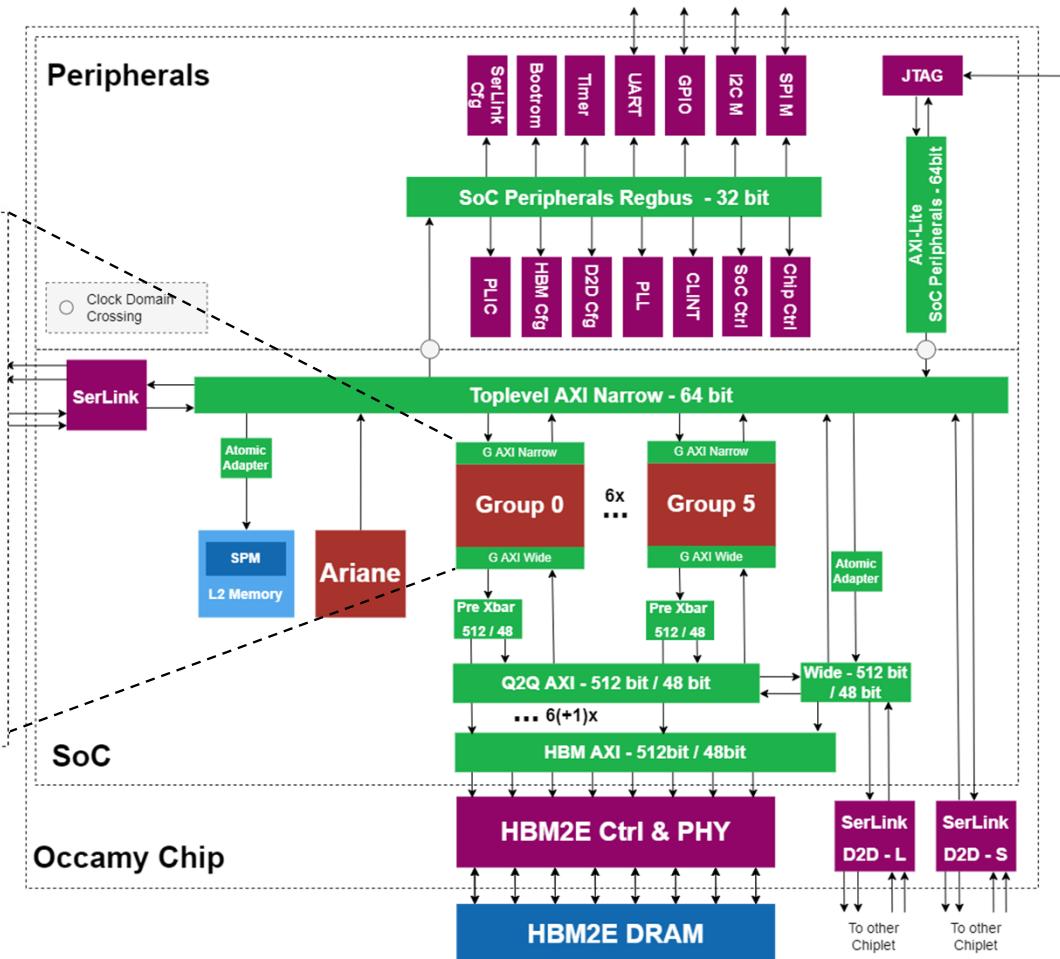
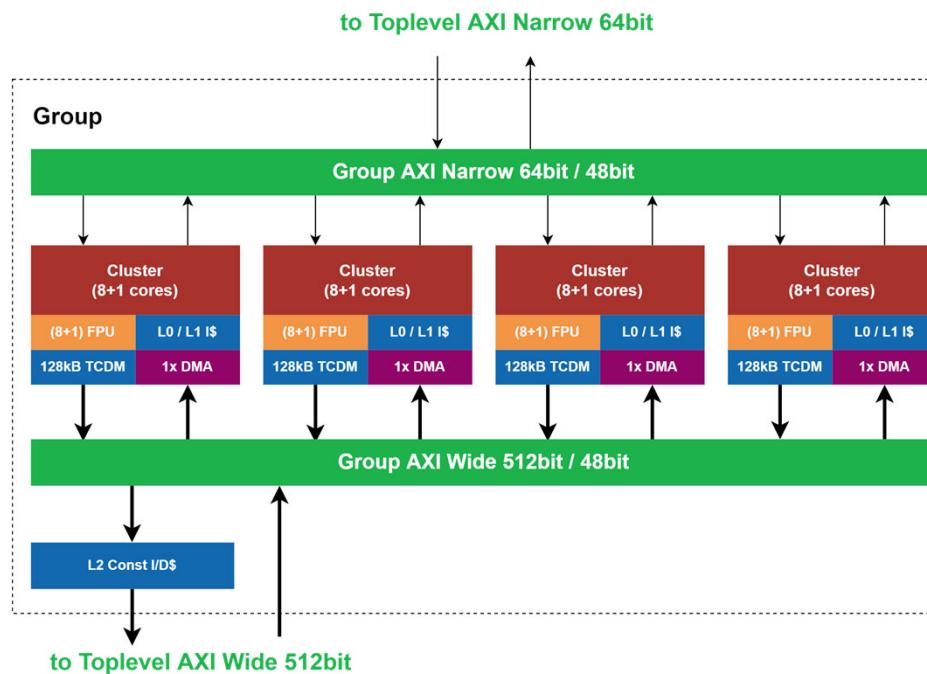
## Constant Cache

- D/I-Cache hierarchy

# From Snitch Group



# To Multi-Group



# Occamy – Chiplet Architecture

## 2 AXI Busses

- 64-bit narrow interface: config
- 512-bit wide interface: DMA

## Peripherals

- Complex address space management

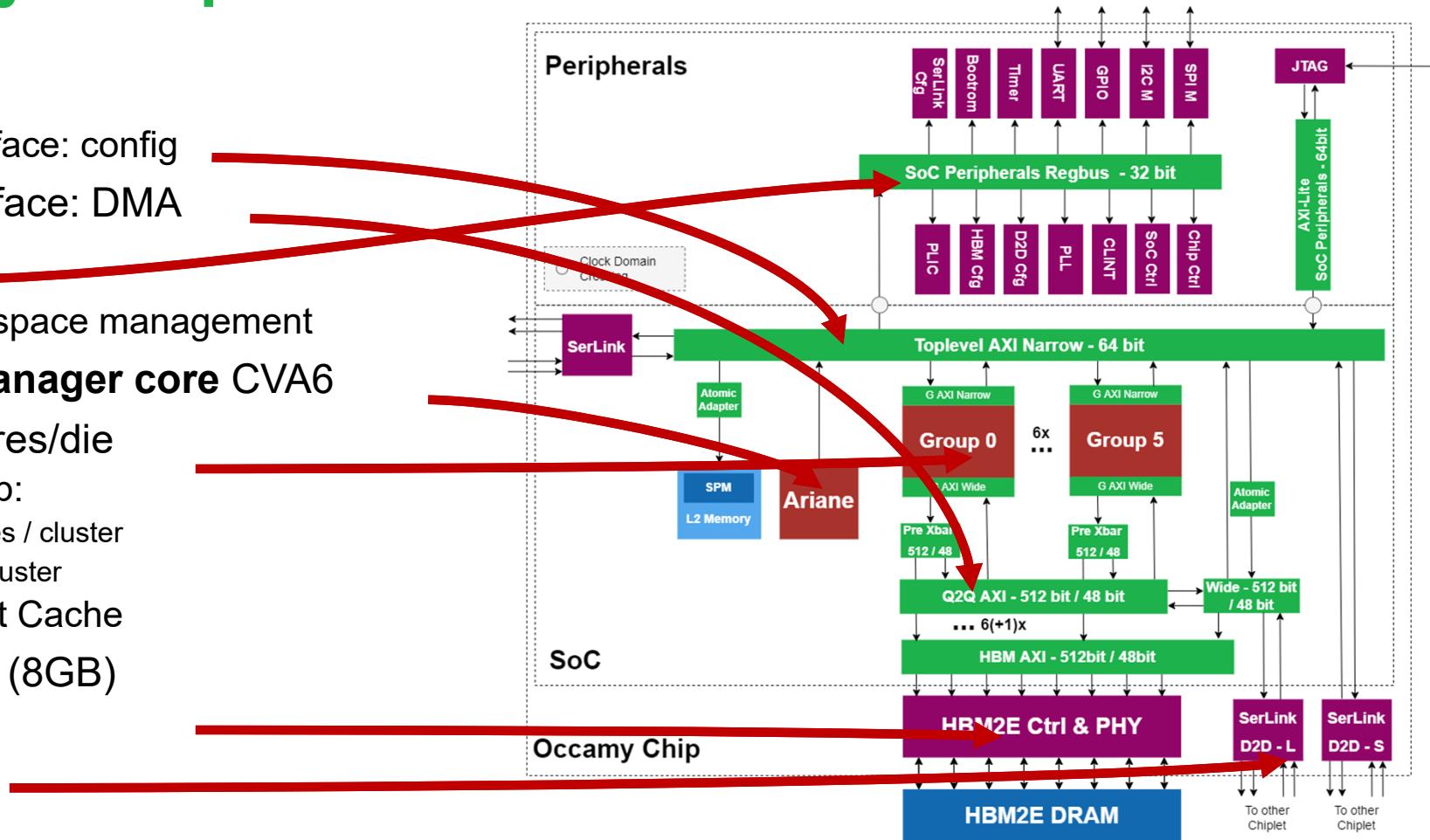
## Linux-capable manager core CVA6

## 6 Groups: 216 cores/die

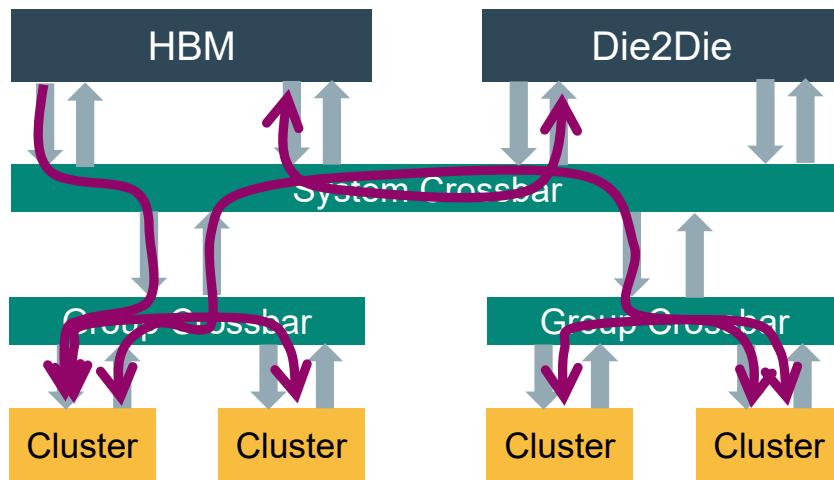
- 4 cluster / group:
  - 8 compute cores / cluster
  - 1 DMA core / cluster
- 512bit Constant Cache

## 8-channel HBM2e (8GB)

## D2D serial link



# Occamy NoC: Efficient and Flexible Data Movement



[1] Burrello, Alessio, et al. "Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus." IEEE Transactions on Computers 70.8 (2021): 1253-1268.

**Problem:** HBM Accesses are critical in terms of

- Access energy
- Congestion
- High latency

Instead reuse data on lower levels of the memory hierarchy

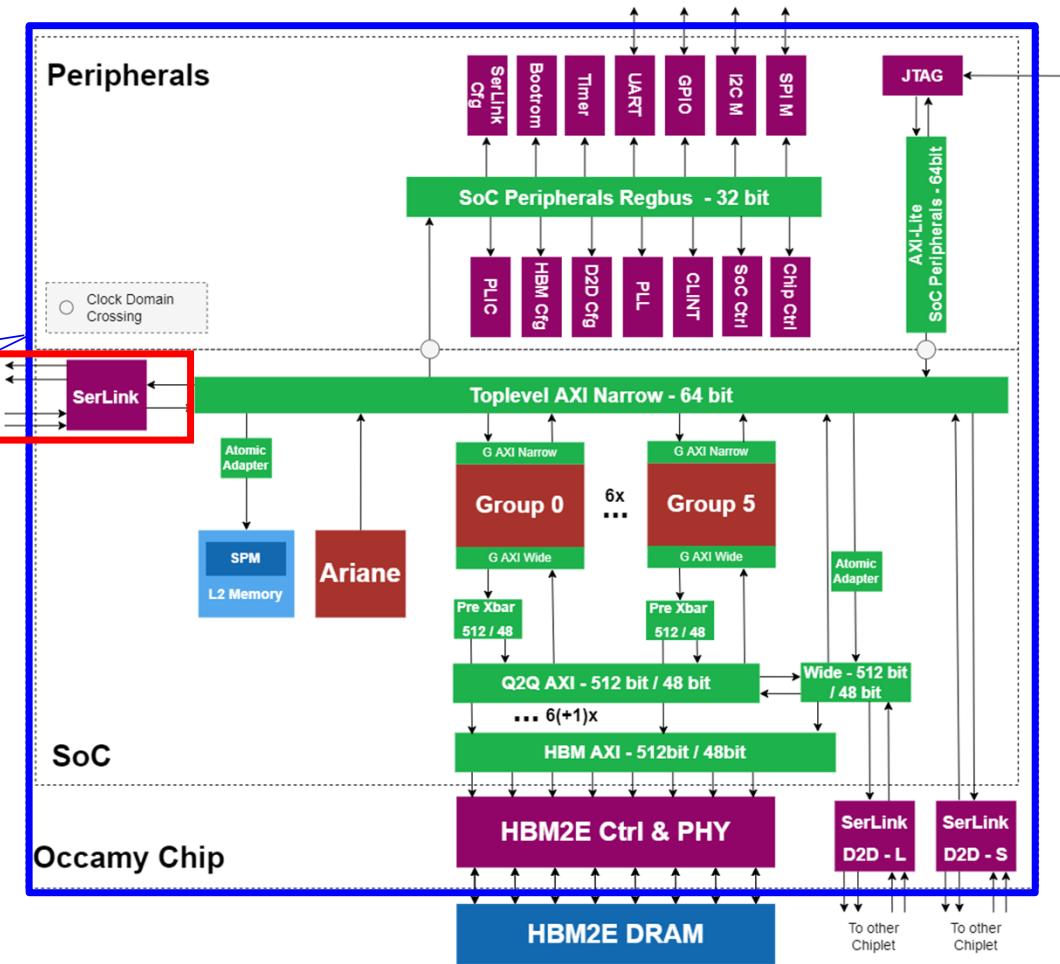
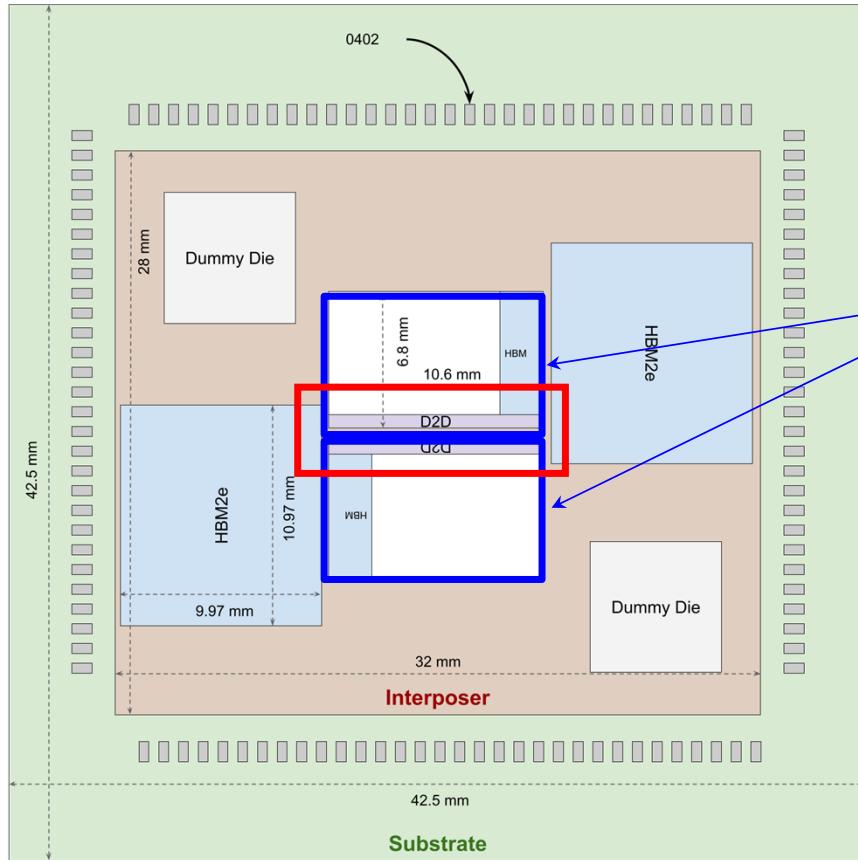
- Between **clusters**
- Across **groups**

Smartly distribute workload

- **Clusters:** *DORY* framework for tiling strategy [1]
- **Chiplets:** E.g. Layer pipelining

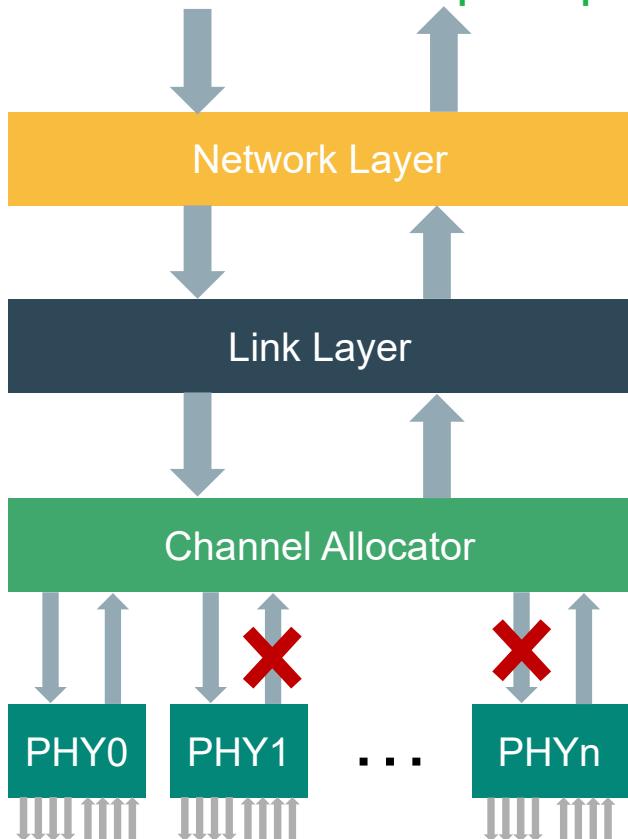
Big trend today!

# Occamy's C2C Link



# C2C Link

Scalable and fault tolerant Chiplet2Chiplet Link



## Network Layer

- Full AXI4 interface
- AXI4 to AXI stream converter

## Data Link Layer

- Credit-based flow control
- RX synchronisation

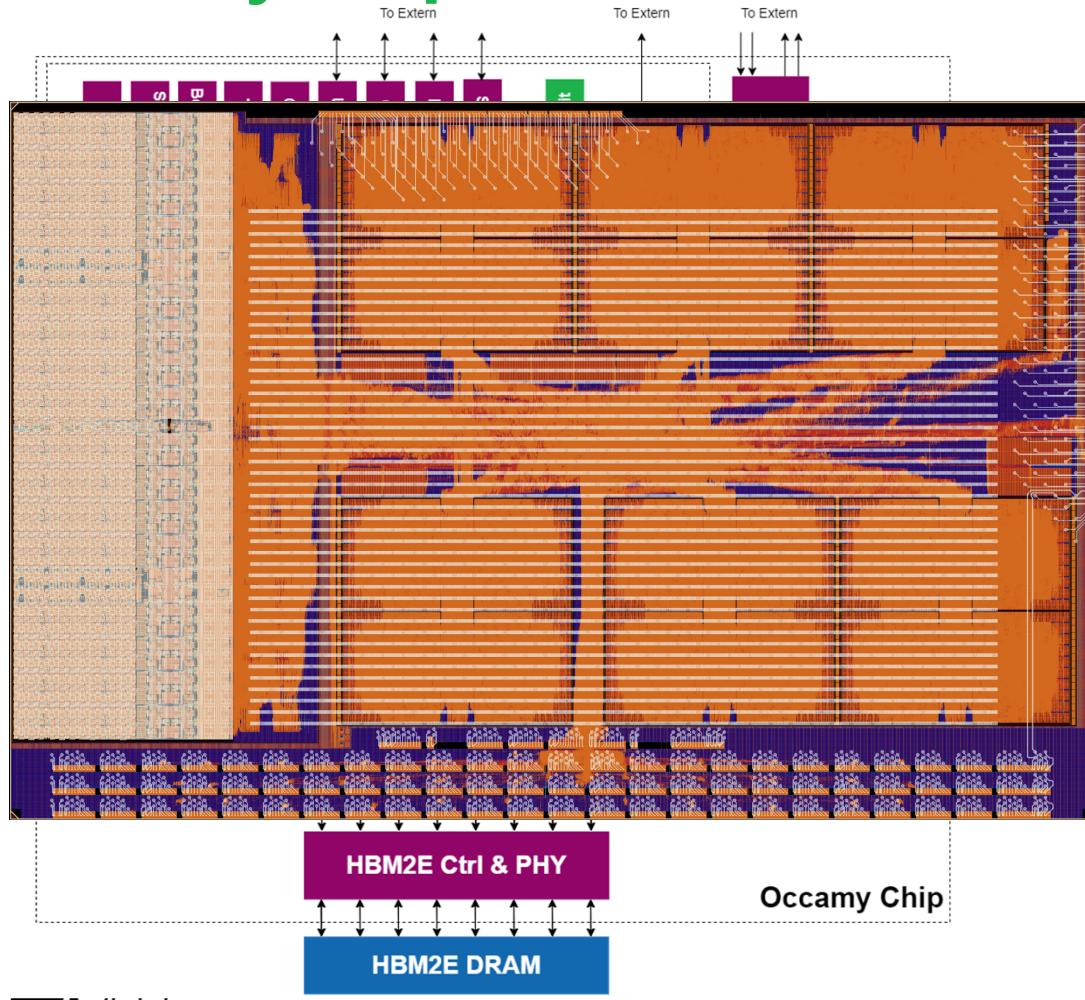
## Channel Allocator

- Chops and reshuffles payload
- **Fault tolerance** mechanisms

## Physical layer (decoupled)

- Source-synchronous DDR sampling
- **Scalable** number of channels  
(38 x 8-bit full-duplex DDR channels in Occamy)
- Compatible with fast serial phy (e.g. HBI)

# Occamy Chiplet



- GF12, target **1GHz** (typ)
- 2 AXI NoCs (multi-hierarchy)
  - 64-bit
  - 512-bit with “interleaved” mode
- Peripherals
- Linux-capable manager core CVA6
- 6 Quadrants: 216 cores/chiplet
  - 4 cluster / quadrant:
    - 8 compute +1 DMA core / cluster
    - 1 multi-format FPU / core (FP64,x2 32, x4 16/alt, x8 8/alt)
- 8-channel HBM2e (8GB) **512GB/s**
- D2D link (Wide, Narrow) **70+2GB/s**
- System-level DMA
- SPM (2MB wide, 512KB narrow)

**Peak 384 GDPflop/s per chiplet – Taped-out in July**

# Efficient SoC architecture in Perspective

1. Multi-cluster single-die scaling → strong latency tolerance, modularity
2. NoC for flexible Clus2Clus, Clus2Mem traffic → reduce pressure to Main memory
3. Full chiplet architecture: HBM2e, NoC-wrapped C2C, multi-chiplet ready

# Back to the cluster... Can we make it Bigger?

- **Why?**
  - Better global latency tolerance if  $L1 > 2 * \text{Latency} * \text{Bandwidth}$
  - Easier to program (data-parallel, functional pipeline...)
  - Smaller data partitioning overhead
- **An efficient many-core cluster with low-latency shared L1**
  - 256+ cores
  - 1+ MiB of shared L1 data memory
  - $\leq 5$  cycles latency (without contention)
- **Physical-aware design**
  - WC Frequency  $> 500$  Mhz
  - Targeting iso-frequency with small cluster



# Hierarchical Physical Architecture

- **Tile**

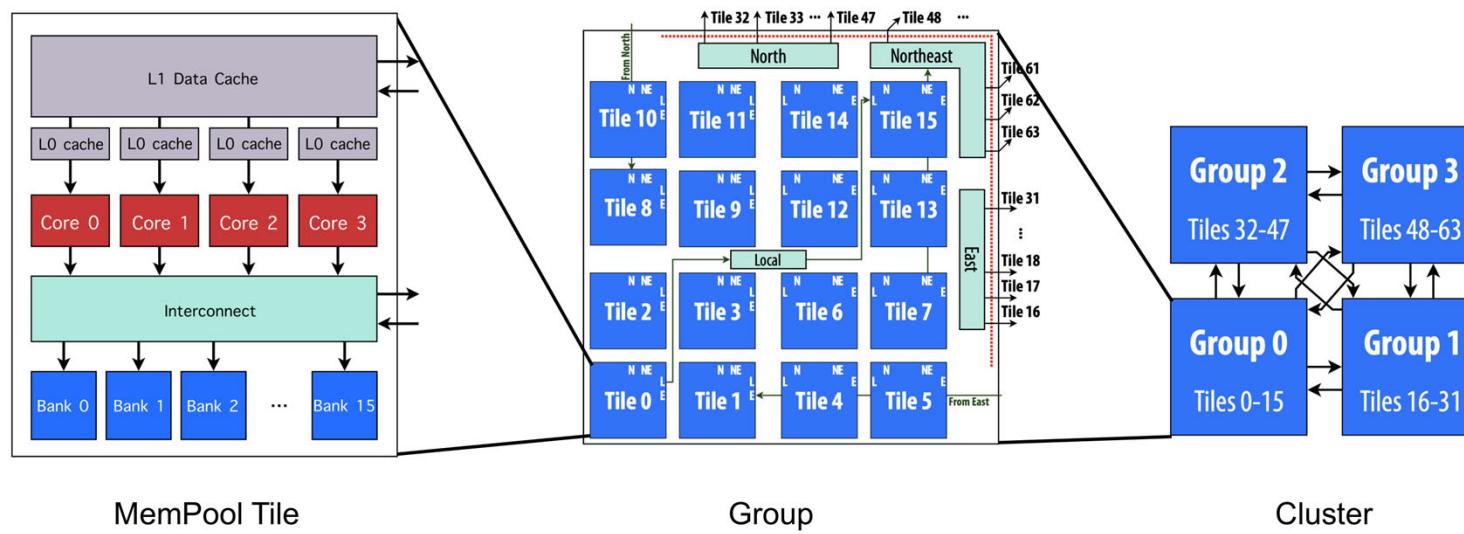
- 4 32-bit cores
- 16 banks
- **Single cycle** memory access

- **Group**

- 64 cores
- 256 banks
- **3-cycles latency**

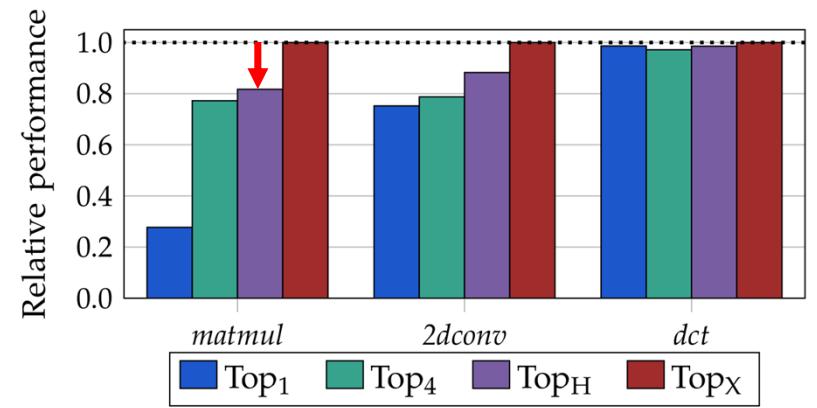
- **Cluster**

- 256 cores
- 1 MiB of memory (1024 banks)
- **5-cycles latency**



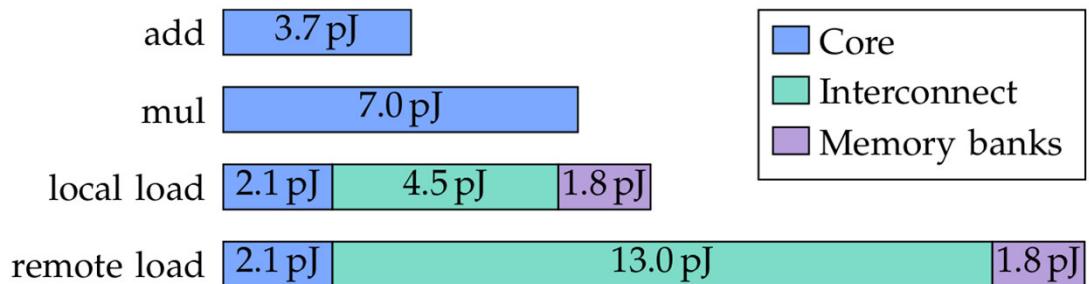
# Benchmarks: can we compete with the impossible?

- Baseline: idealized Top<sub>X</sub>
  - Fully connected logarithmic crossbar between 256 cores and 1024 banks
- Cycle-accurate RTL simulation
  - *Matmul*
    - Multiplication of two  $64 \times 64$  matrices
  - *2dconv*
    - 2D Convolution with a  $3 \times 3$  kernel
  - *dct*
    - 2D Discrete Cosine Transform on  $8 \times 8$  blocks in local memory
- Top<sub>H</sub> has a performance penalty of at most 20%, on all kernels

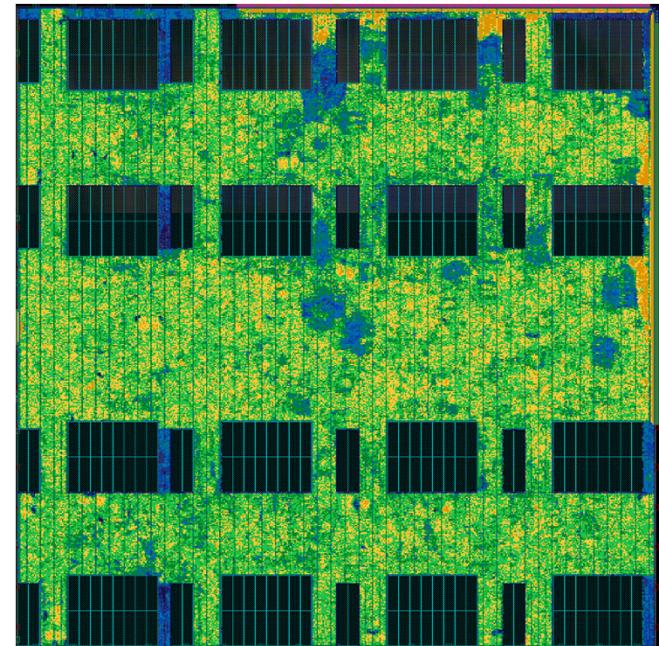


# Energy Analysis (500 MHz, TT, 0.80 V, 25 °C)

- Breakdown of the energy consumption per instruction:

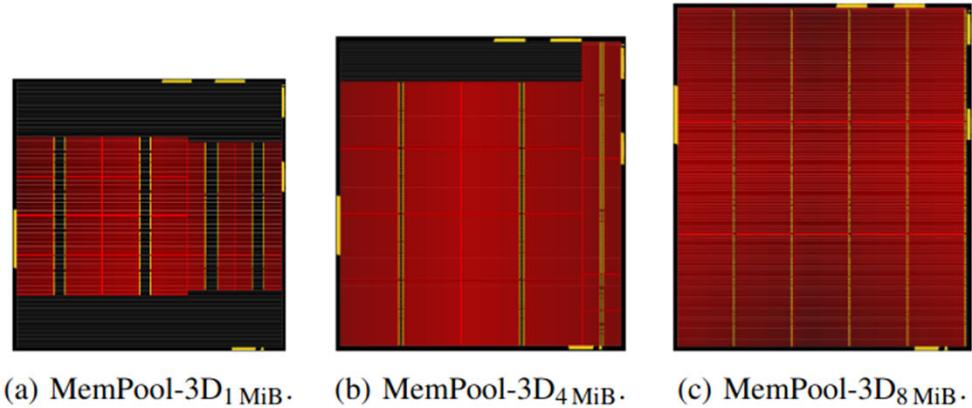
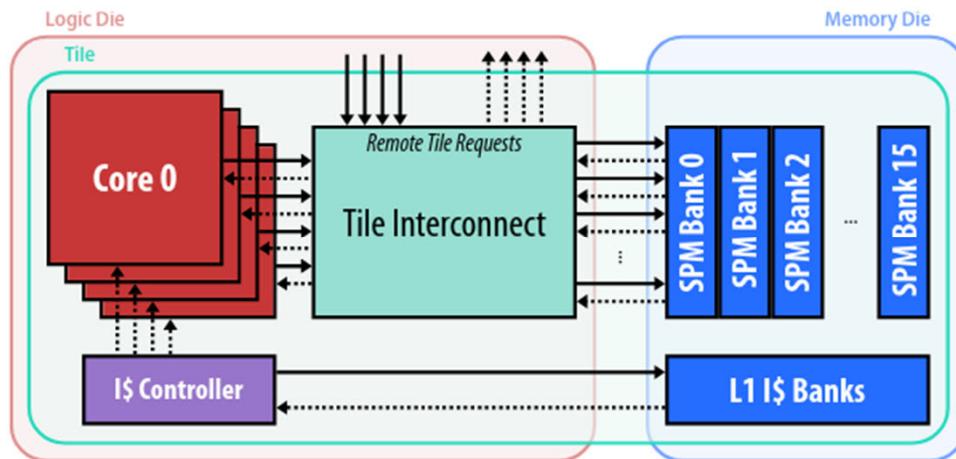


- Local loads consume about as much energy as a *mul*
  - About half of it, 4.5pJ, by the interconnect
- Remote loads: twice the energy of a local load
  - Despite crossing the whole cluster, twice!



# MemPool-3D

- Memory-on-Logic implementation of MemPool
- Implementation of MemPool with 1, 2, 4, 8 MiB of L1
- Leading to a higher utilization of the memory die



# Groups: MemPool-2D vs. MemPool-3D

## MemPool-2D

128 KiB    Footprint: 1.000  
↓  
*7% larger*

256 KiB    Footprint: 1.074

512 KiB    Footprint: 1.299

1 MiB      Footprint: 1.572

## MemPool-3D

Footprint: 0.665  
↓  
*Same footprint*

Footprint: 0.665

Footprint: 0.737

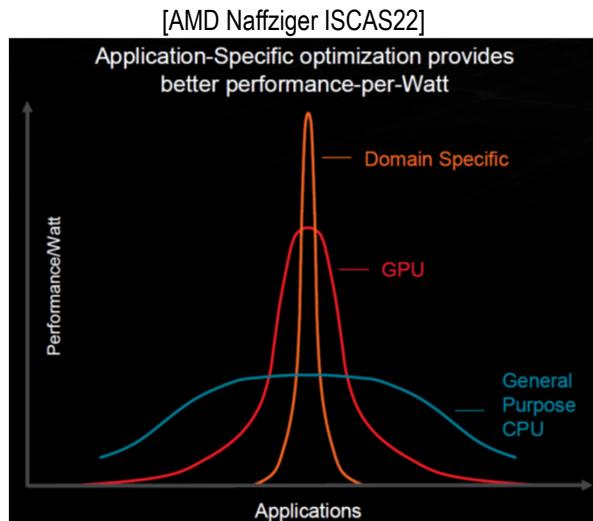
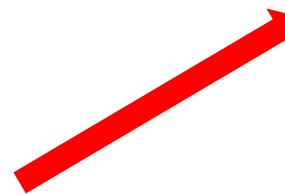
*14% smaller, despite 8x the L1 capacity!*

Footprint: 0.857

**Similar trends for Wire length and #Buffers**

# Conclusion

- Energy efficiency quest: PE, Cluster, SoC



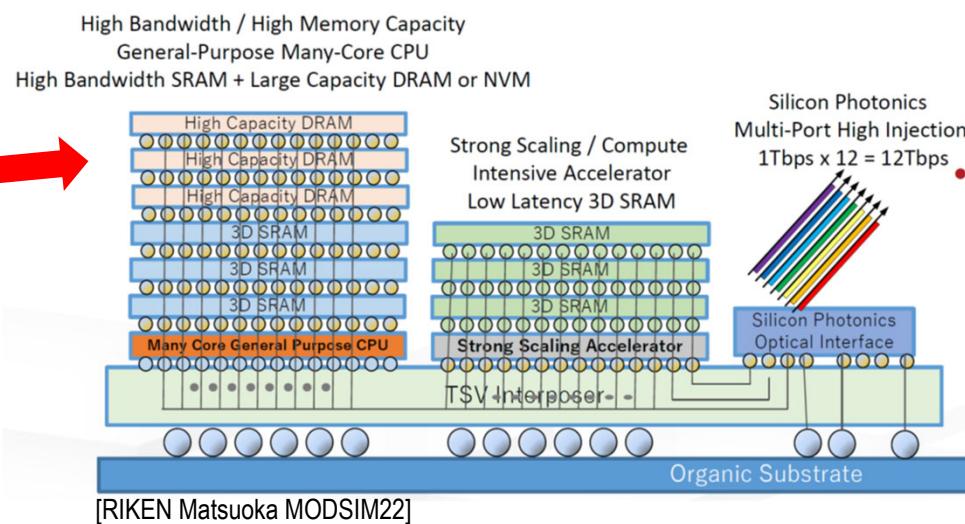
- Key ideas

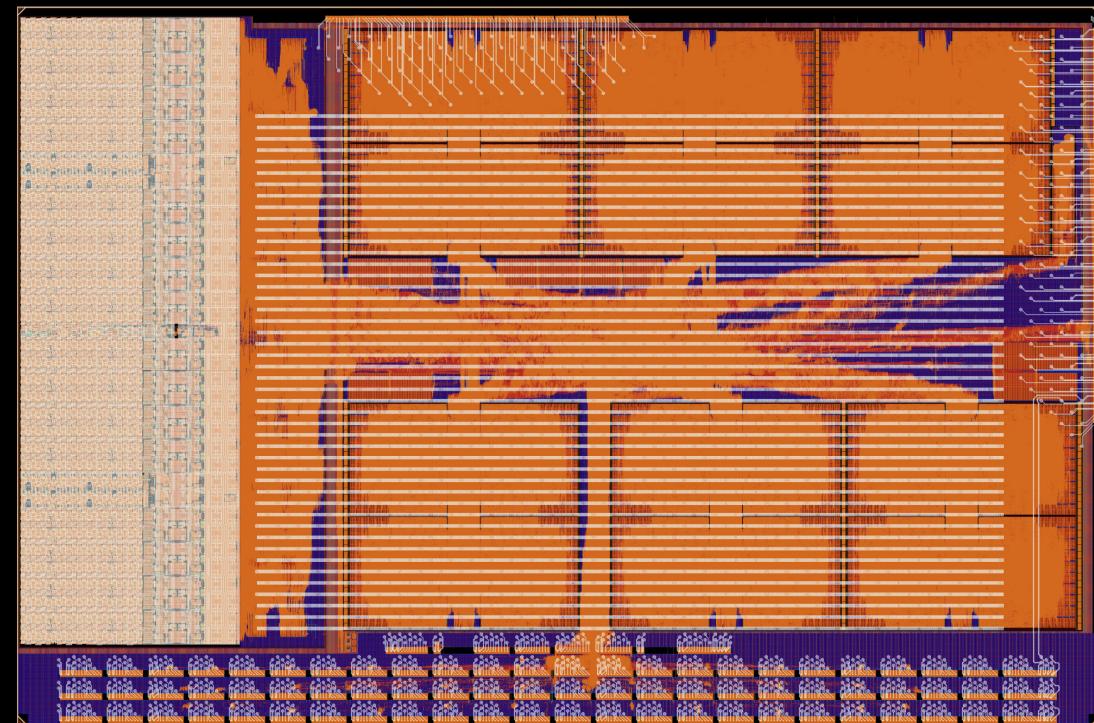
- Application specific PEs → extensible ISAs (RISC-V!)
- VNB removal + Latency hiding: large OOO processors not needed
- Low-overhead work distribution → large “mempool”
- Heterogeneous architecture → host+accelerator(s)



- Game-changing technologies

- Chiplets: 2.5D, 3D, (WFI?)
- Computing “at” memory,
- Coming (less than ten years): optical IO





# PULP

Parallel Ultra Low Power



<http://pulp-platform.org>



@pulp\_platform

Luca Benini, Alessandro Capotondi, Alessandro Ottaviano, Alessandro Nadalini, 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, Davide Nadalini, Fabio Montagna, Florian Glaser, Florian Zaruba, Francesco Conti, Frank K. Gürkaynak, Georg Rutishauser, Germain Haugou, Gianna Paulin, Gianmarco Ottavi, Giuseppe Tagliavini, Hanna Müller, Lorenzo Lamberti, Luca Bertaccini, Luca Valente, Luca Colagrande, Luka Macan, Manuel Eggimann, Manuele Rusci, Marco Guermandi, Marcello Zangheri, Matheus Cavalcante, Matteo Perotti, Matteo Spallanzani, Mattia Sinigaglia, Michael Rogenmoser, Moritz Scherer, Moritz Schneider, Nazareno Bruschi, Nils Wistoff, Pasquale Davide Schiavone, Paul Scheffler, Philipp Mayer, Robert Balas, Samuel Riedel, Sergio Mazzola, Sergei Vostrikov, Simone Benatti, Stefan Mach, Thomas Benz, Thorir Ingolfsson, Tim Fischer, Victor Javier Kartsch Morinigo, Vlad Niculescu, Xiaying Wang, Yichao Zhang, Yvan Tortorella, all our past collaborators **and many more that we forgot to mention**

# Specialize?

