

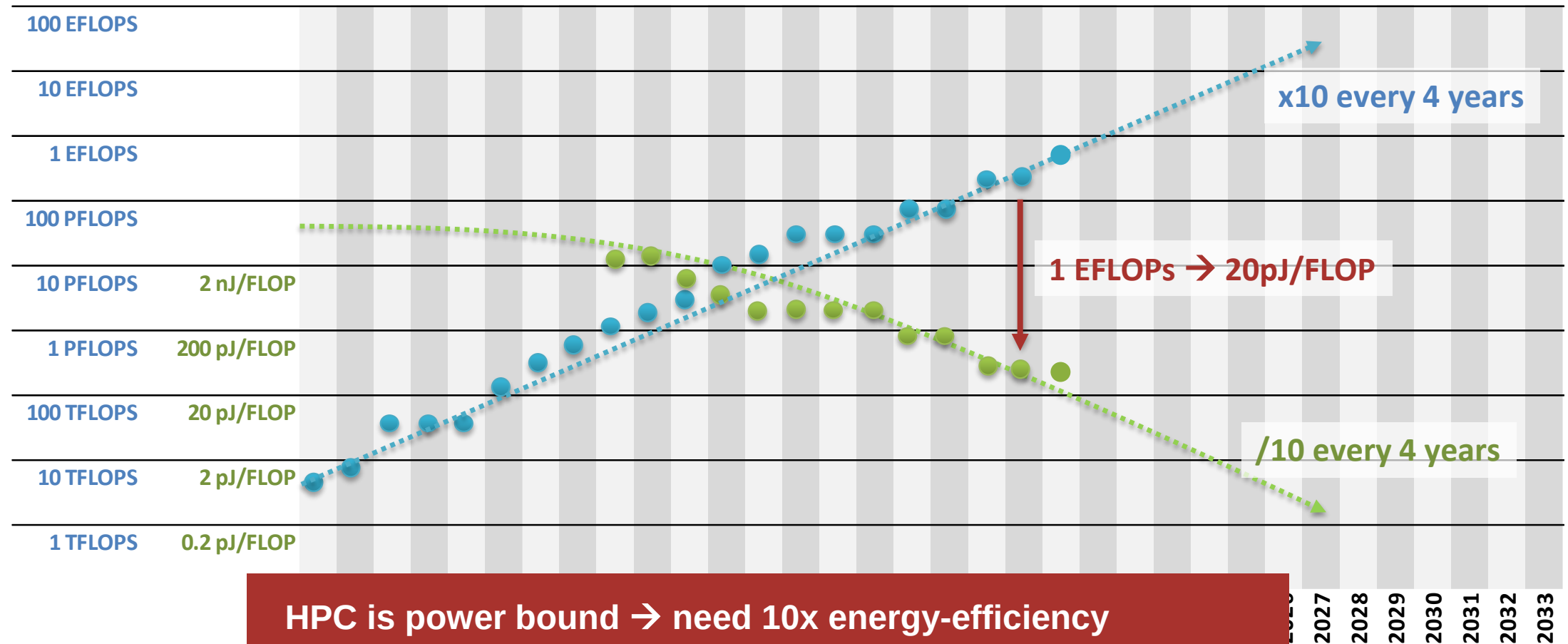


Manticore as an NoC Case Study: A 4096 Chiplet-based Architecture for Ultra-Efficient Floating-Point Computing

Dr. Florian Zaruba, zarubaf@iis.ee.ethz.ch

Energy Efficiency at Exascale

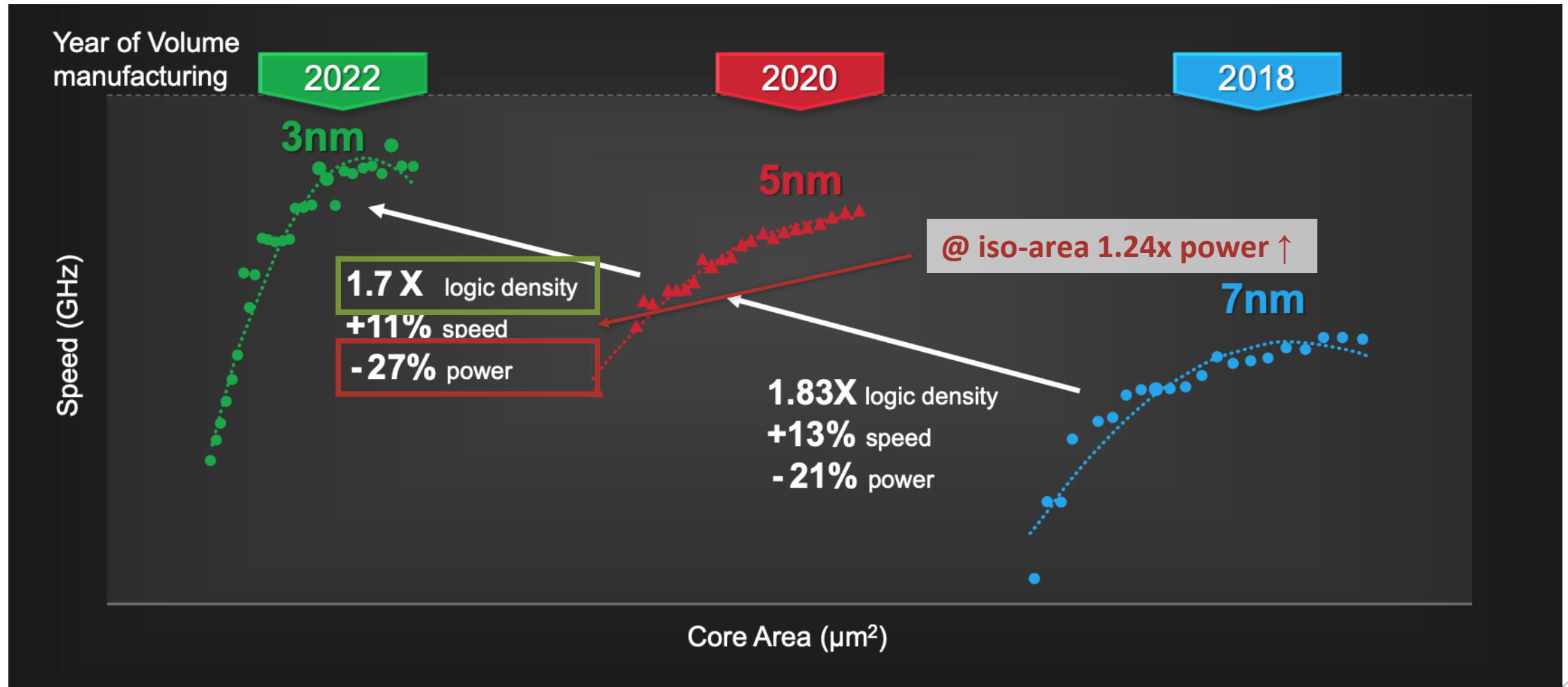
Performance Energy / Op (20MW total hard bound)



EPI Tutorial/Barcelona/17-07-2

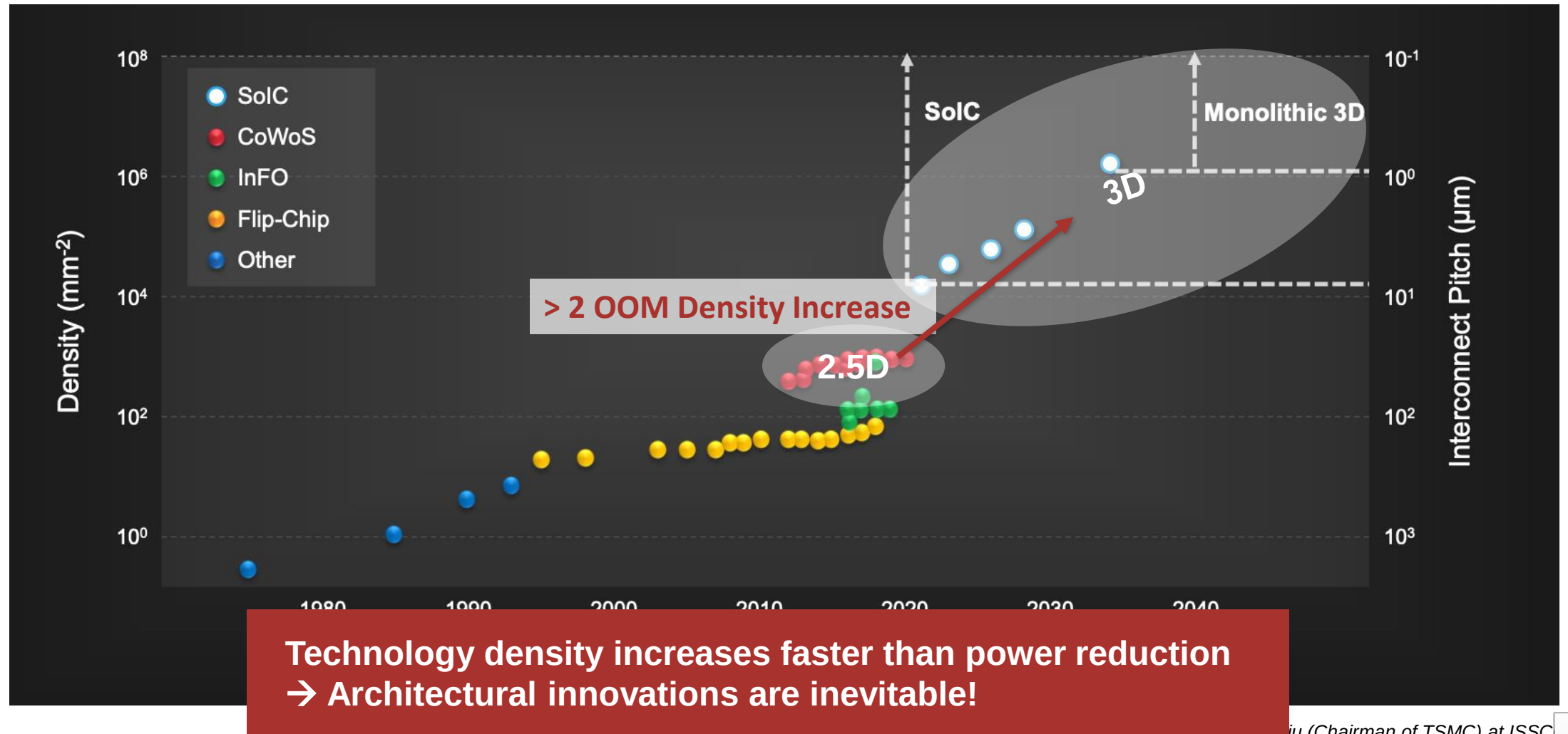


Technology Scaling



Unleashing the Future of Innovation, Dr. Mark Liu (Chairman of TSMC) at ISSC

System Scaling



Embracing the Future of Innovation, Li-Min Liu (Chairman of TSMC) at ISSC

Not All Programs Are Created Equal

1. Processors can do two kinds of useful work:

Decide (jump to different program part)

- Modulate flow of **instructions**
- **Smarts:**
 - Don't work too much
 - Be clever about the battles you pick (e.g., search in a database)
- Lots of decisions
Little number crunching

Compute (plough through numbers)

- Modulate flow of **data**
- **Diligence:**
 - Don't think too much
 - Just plough through the data (e.g., machine learning)
- Few decisions
Lots of number crunching

2. Many of today's challenges are of the **diligence** kind:

1. Tons of data, algorithm just ploughs through, few decisions done based on the computed values
2. **“Data-Oblivious Algorithms” (ML, or better DNNs are so!)**



An Illustrative Example

- Dot product / scalar product
- Multiply elements of a vector and sum up

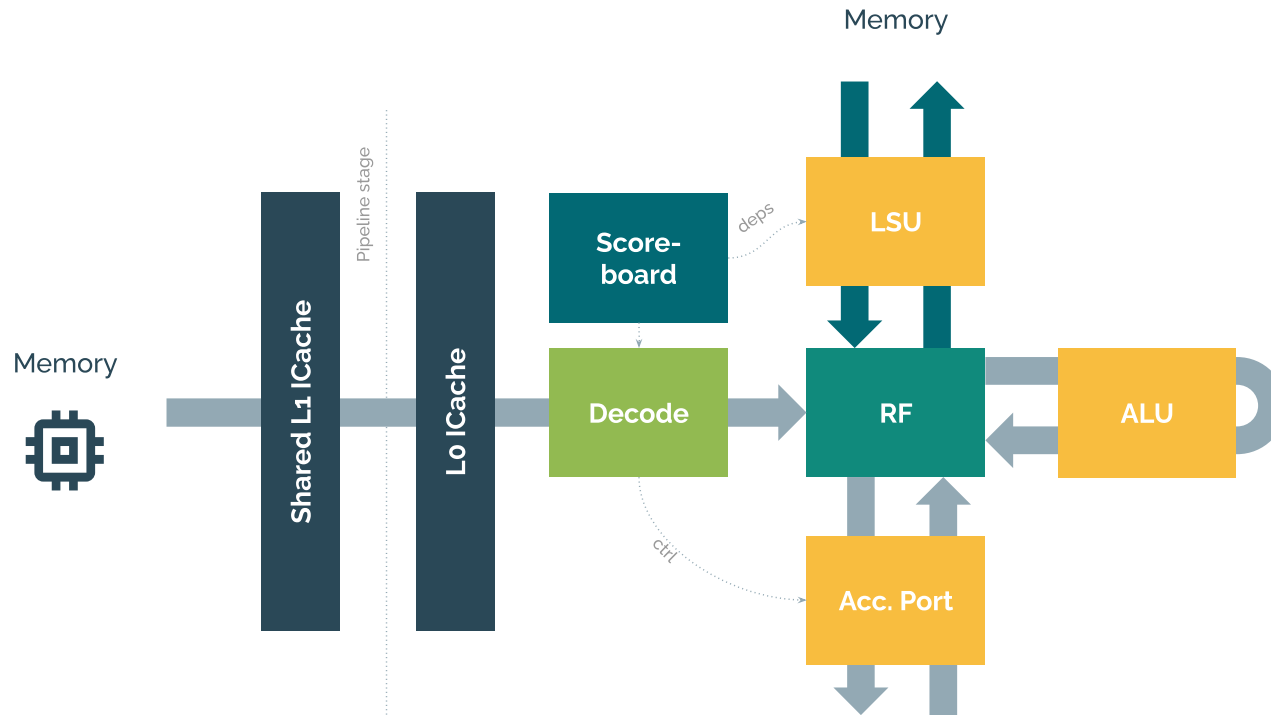
```
double sum = 0;
for (int i = 0; i < N; ++i) {
    sum += A[i] * B[i];
}
```

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



Snitch: Tiny Control Core

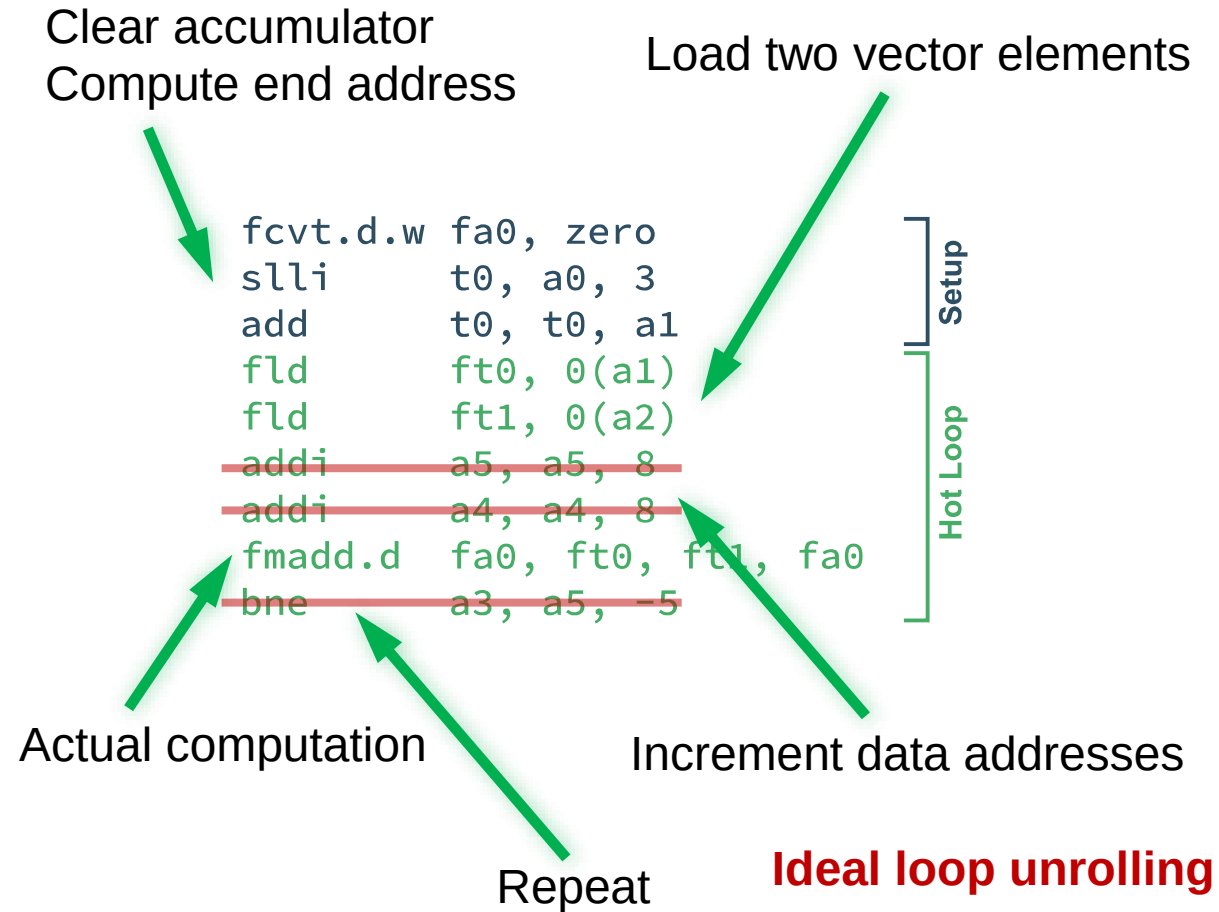
A versatile building block.



- Most simplistic core possible
- Focus on the important aspects:
 - **Tiny, simple, and lightweight control core**
 - **Extensibility**: Performance through ISA extensions
 - **Latency tolerant**
 - **Competitive frequency**
- Around 10-20 kGE

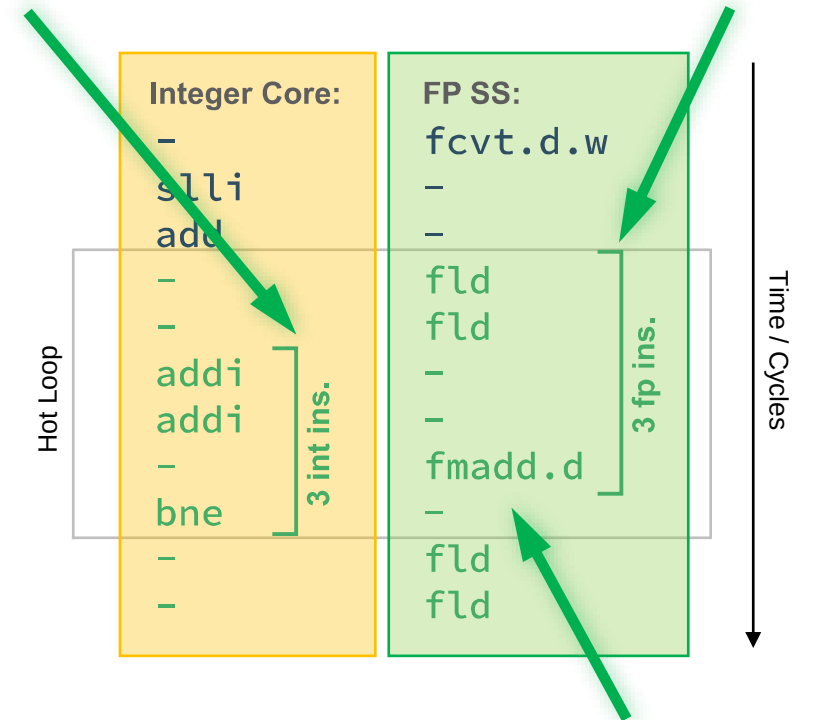


Baseline RISC-V



3 integer ops

3 float ops



33% useful work



Adding SSRs

Configure SSRs (more flexibility = more knobs)
Clear accumulator

```

la      a5, SSR_CFG
li      t1, 8
sw      t1, STEP0(a5)
sw      t1, STEP1(a5)
addi    t1, a0, -1
sw      t1, BOUND0(a5)
sw      t1, BOUND1(a5)
sw      a1, BOUND0(a5)
sw      a2, BOUND1(a5)
csrsi   ssrcfg, 1
fcvt.d.w fa0, zero
fmadd.d fa0, ft0, ft1, fa0
addi    a0, a0, 1
bnez    a0, -2
csrsi   ssrcfg, 0

```

Setup

Hot Loop

Loads elided Implicit in ft0/ft1

2 integer ops

1 float op

Actual computation

→ 2-3x speedup over baseline

Repeat

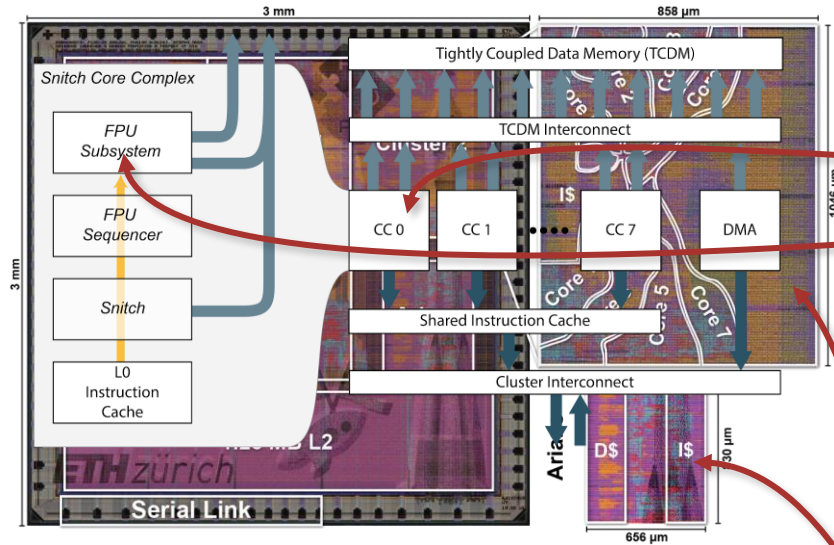
Ideal loop unrolling

100% useful work



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 NTX
- 9 mm² prototype in 22nm
- Testbed for key architectural components
 - Snitch octa-core cluster
 - CVA6 cores

Very efficient, versatile, compute cluster!



And where does the Energy go?

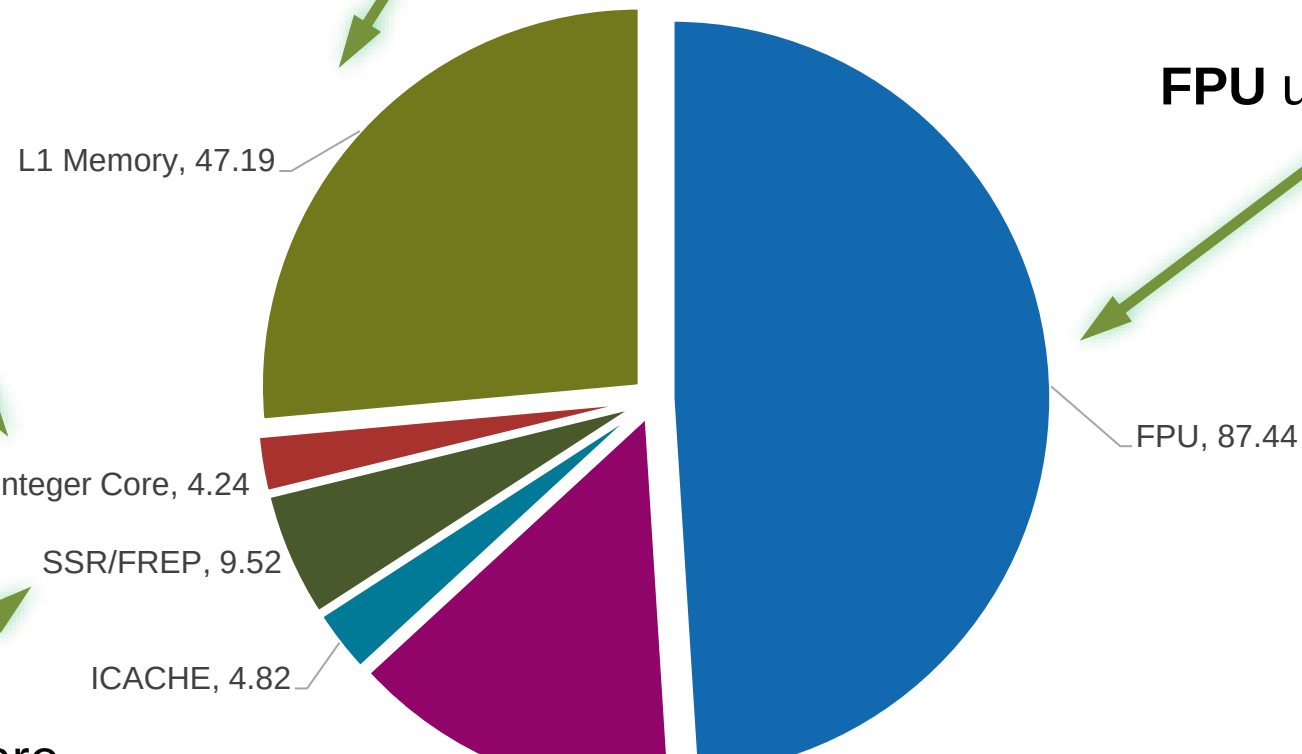
In an 8-core cluster

Inevitable to have local memory
(e.g., CPU/GPU L1 cache, vector register file)

Integer core uses
2% of power

FPU uses 50% of power

SSR/FREP hardware
uses 5% of power

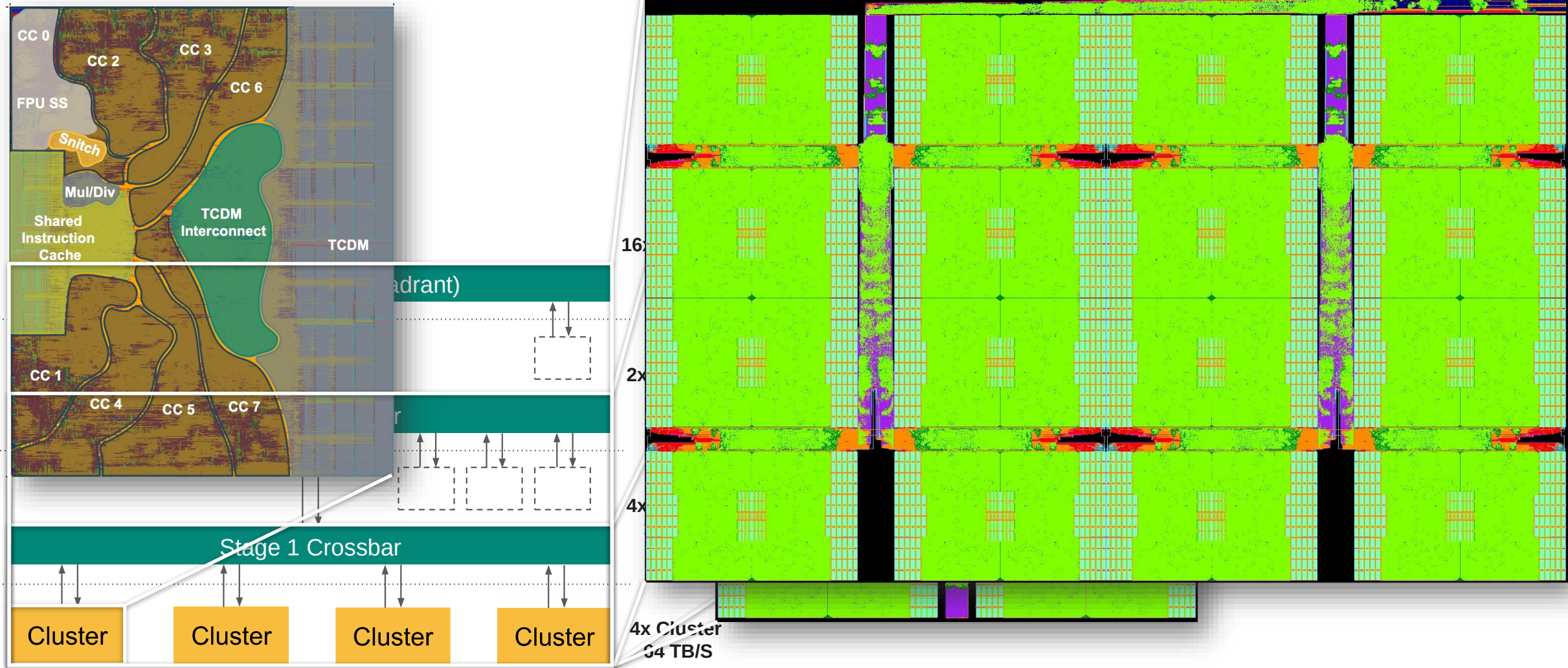


Spending energy where it contributes to the result → **High Efficiency**
How do we scale?



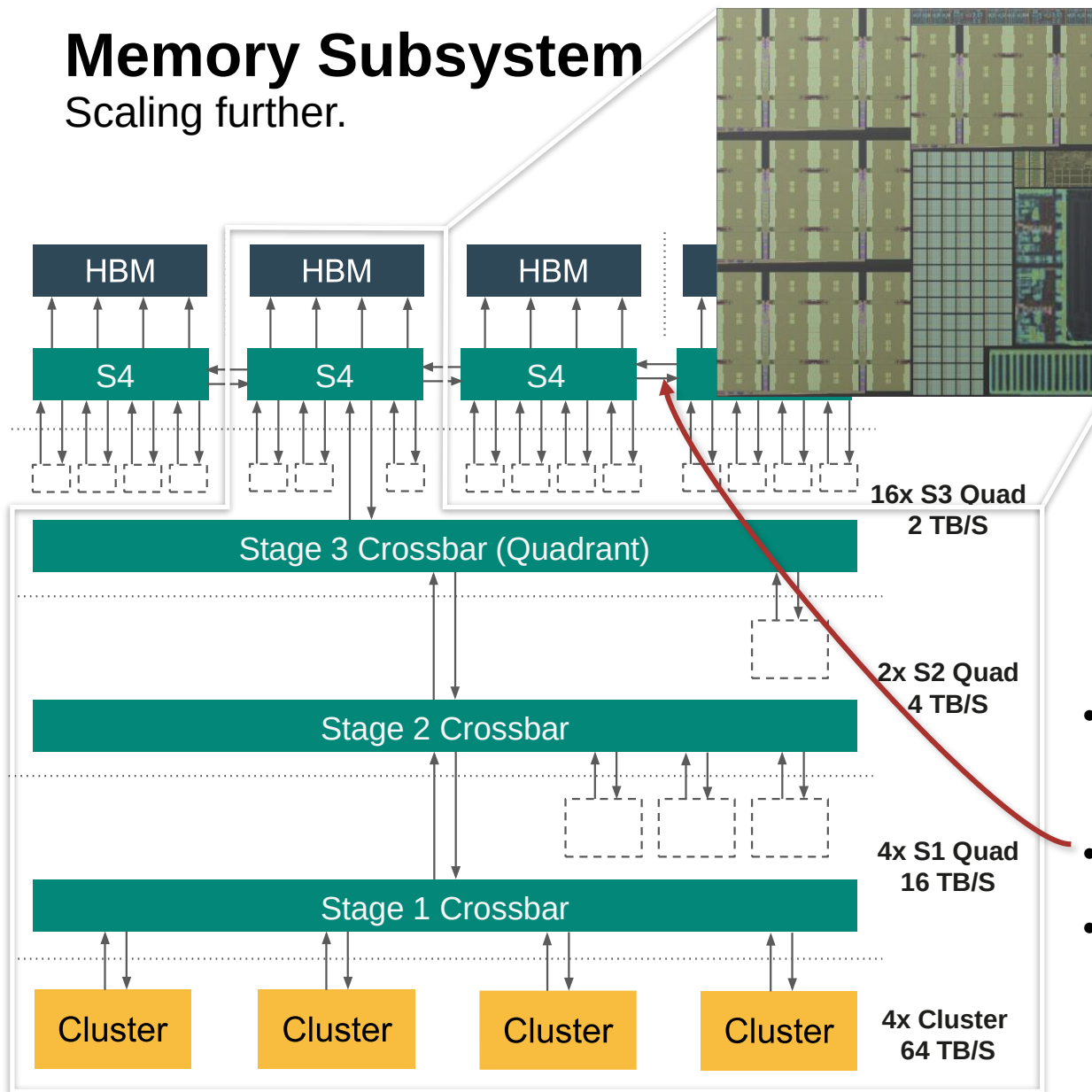
Manticore

The scale-out study.



Memory Subsystem

Scaling further.



HBM: Matching the bandwidth with the memory interface

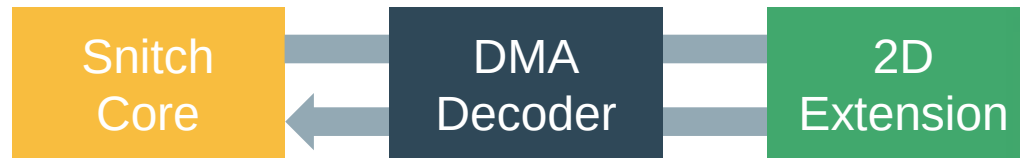
Block-wise DMA accesses:

- High bus utilization $\approx$ high energy-efficiency
- Multi-dimensional blocks with Snitch DMA
- Good fit for parallel interfaces (HBM/HBI)
- Latency tolerance through double buffering

- Different to GPU memory hierarchy
- **Multi-chiplet design**
- **HBI:** Scaling across dies (NUMA)



Efficiently Move Data



- 64-bit AXI DMA
- Operates on wide 512-bit data-bus
- Hardware support to autonomously copy 2D shapes
- Higher-dimensionality can be handled by SW
- Intrinsics/library for easy programming
- Exploiting cluster local memory

Implementing CNN Layers on the Manticore Cluster-Based Many-Core Architecture

Technical Report

Andreas Kurth, Fabian Schuiki, and Luca Benini
Digital Circuits and Systems Group, ETH Zurich

v1.0, April 2021

Abstract

This document presents implementations of fundamental convolutional neural network (CNN) layers on the Manticore cluster-based many-core architecture and discusses their characteristics and trade-offs.

1 Introduction

The *Manticore* architecture [1] is a many-core processor designed for high-performance, high-efficiency data-parallel floating-point computing. The implementation proposed in [1] consists of four chiplets on an interposer, see Fig. 1. Each chiplet contains 1024 cores grouped into 128 clusters, one 8 GiB HBM2E controller and PHY, as well as L2 memory and I/Os to connect to the other chiplets.

Each cluster, shown in Fig. 2, contains eight small 32-bit integer RISC-V cores [2], each paired with a large double-precision floating-point unit (FPU) [3], and 128 KiB of tightly-coupled L1 scratchpad memory (SPM) organized in 32 banks of 64 bit width each. The FPU is capable of computing one double-precision

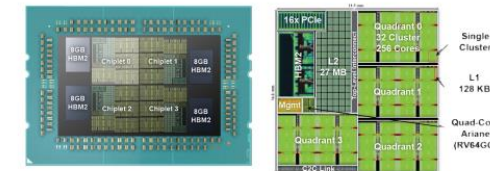


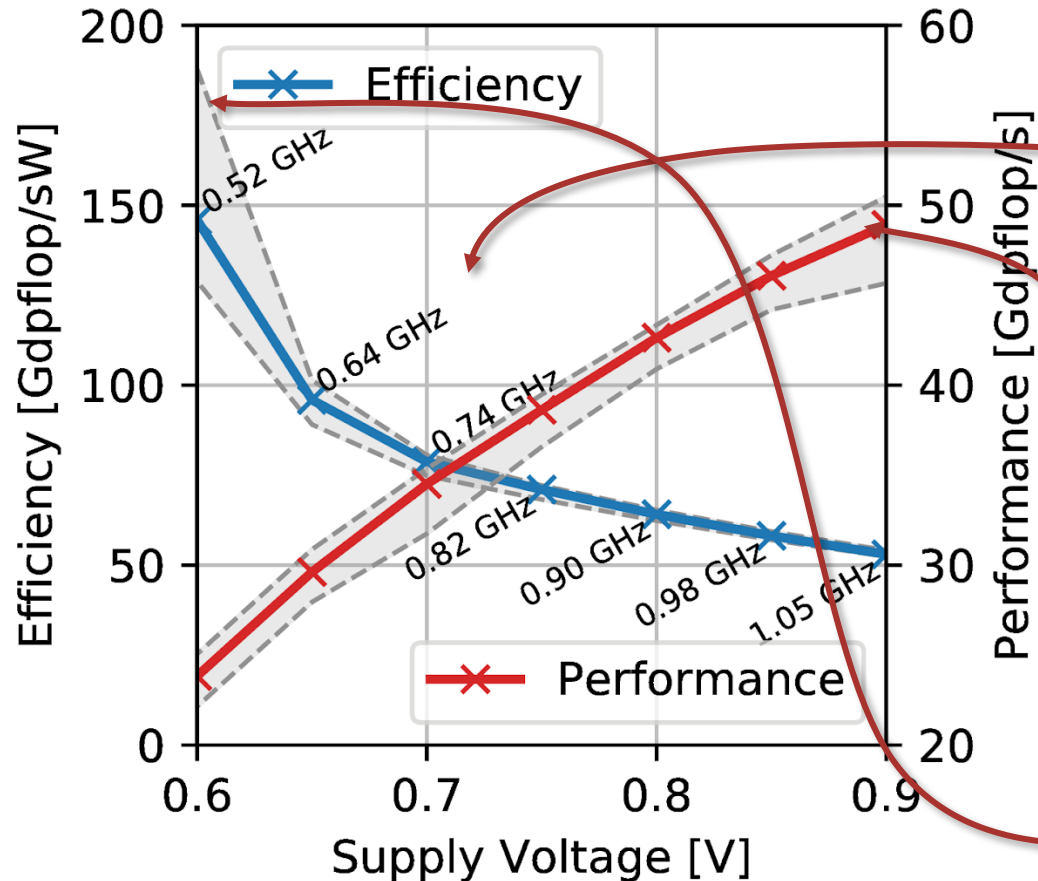
Figure 1: Conceptual floorplan of Manticore's package (left) and an individual chiplet (right) [1].

1

<https://arxiv.org/pdf/2104.08009.pdf>

Performance / Efficiency

Silicon Measurements

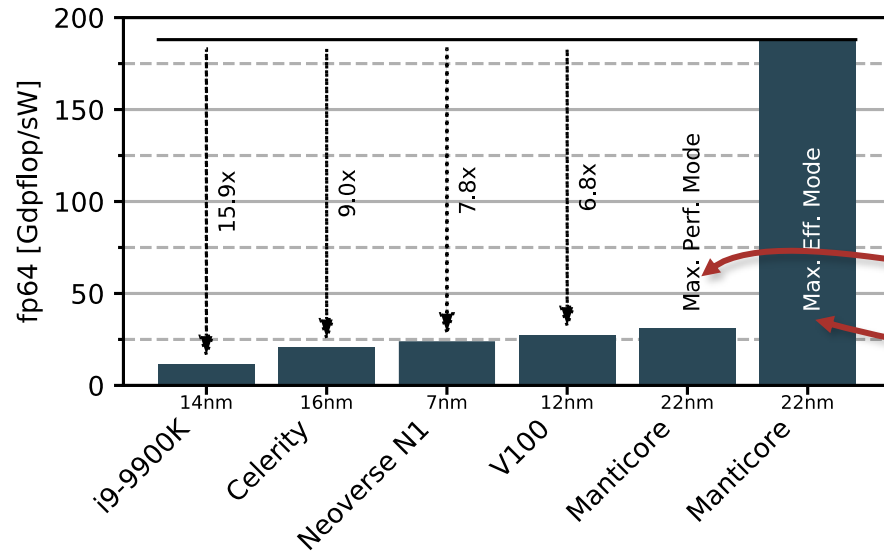


- **>90%** FPU utilization thanks to Snitch cores with Xfrep/Xssr
- Wide range of operating points
- Choice of performance/efficiency trade-off
- On prototype up to **54 DP-Gflop/s** across 24 cores
- On full Manticore system **>27 DP-Tflop/s** across 4096 cores
- Both in **22nm (GF 22FDX)**
- Compute efficiency up to **188 DP-Gflop/sW**
- Compute density up to **20 DP-Gflop/smm²**

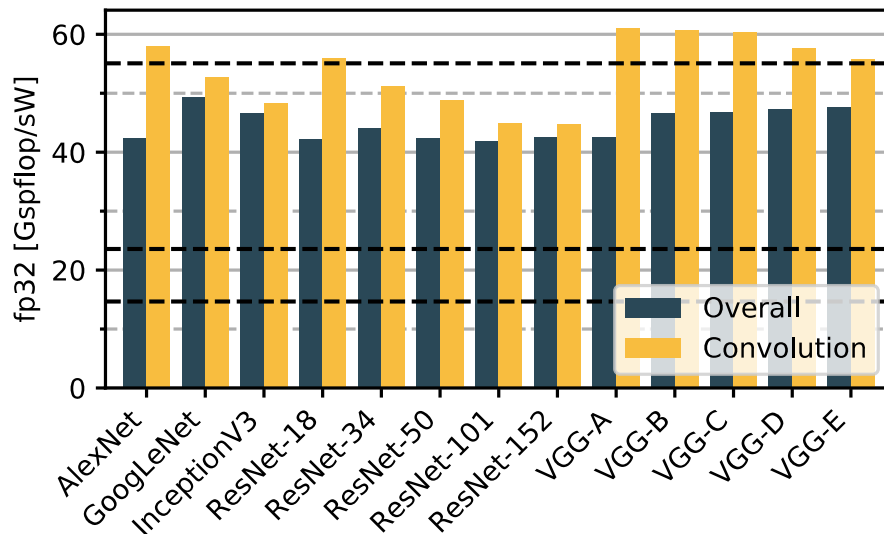


Manticore: Results

How does it compare to state-of-the-art?



- Estimation on **DNN workloads** based on silicon measurements
- Industry-leading fp64 efficiency
 - Maximum-performance mode
 - Maximum-efficiency mode (reduced supply)

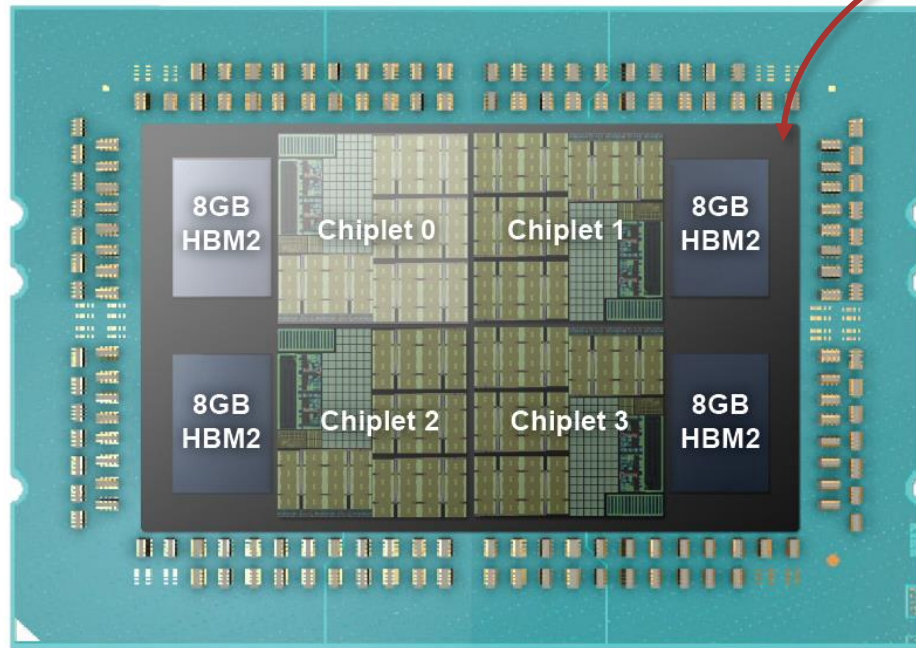


- Competitive fp32 efficiency
 - Competitive with V100
 - >2x better efficiency than i9-9900K
 - >3x better efficiency than N1



Manticore Multi-Chip Concept

Summarizing



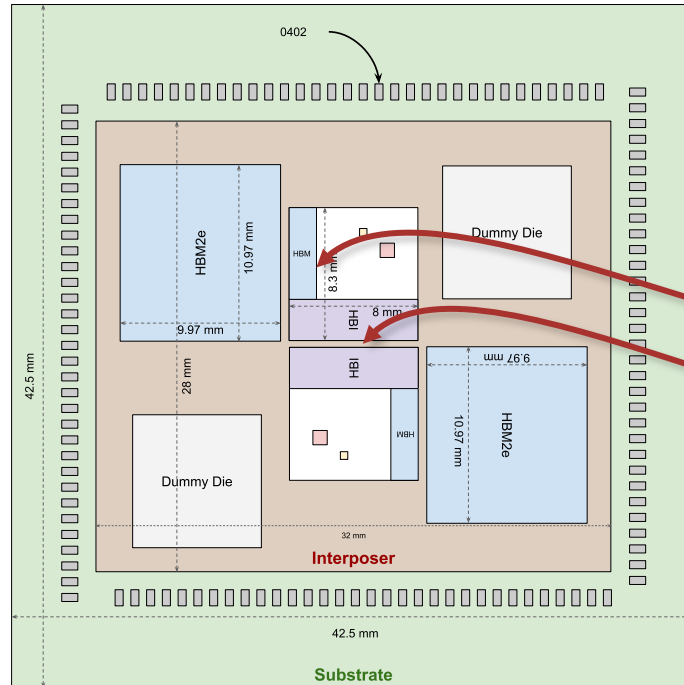
- Four chiplets and 8GB HBM2 on an interposer
 - Interposer enables **high-bandwidth, energy-efficient** parallel interfaces
- High D2D bandwidth
- High die to HBM bandwidth
- Total 4096 Snitch cores, peak > 8 Tdpflop/s
- Four CVA6 "manager" cores

Outperforms SoA, open building-blocks, foundation of next generation high-performance computing systems!

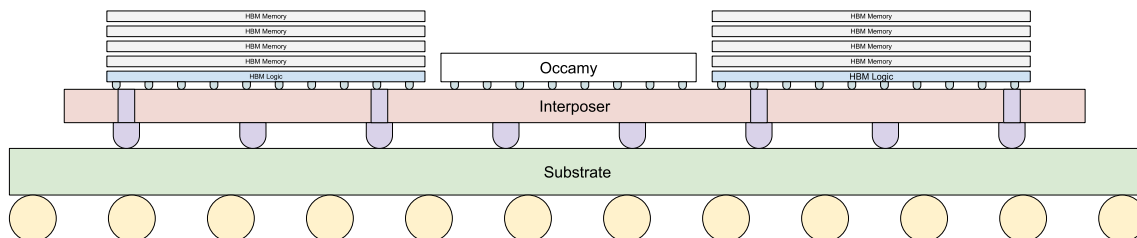


Occamy

Realization of Manticore

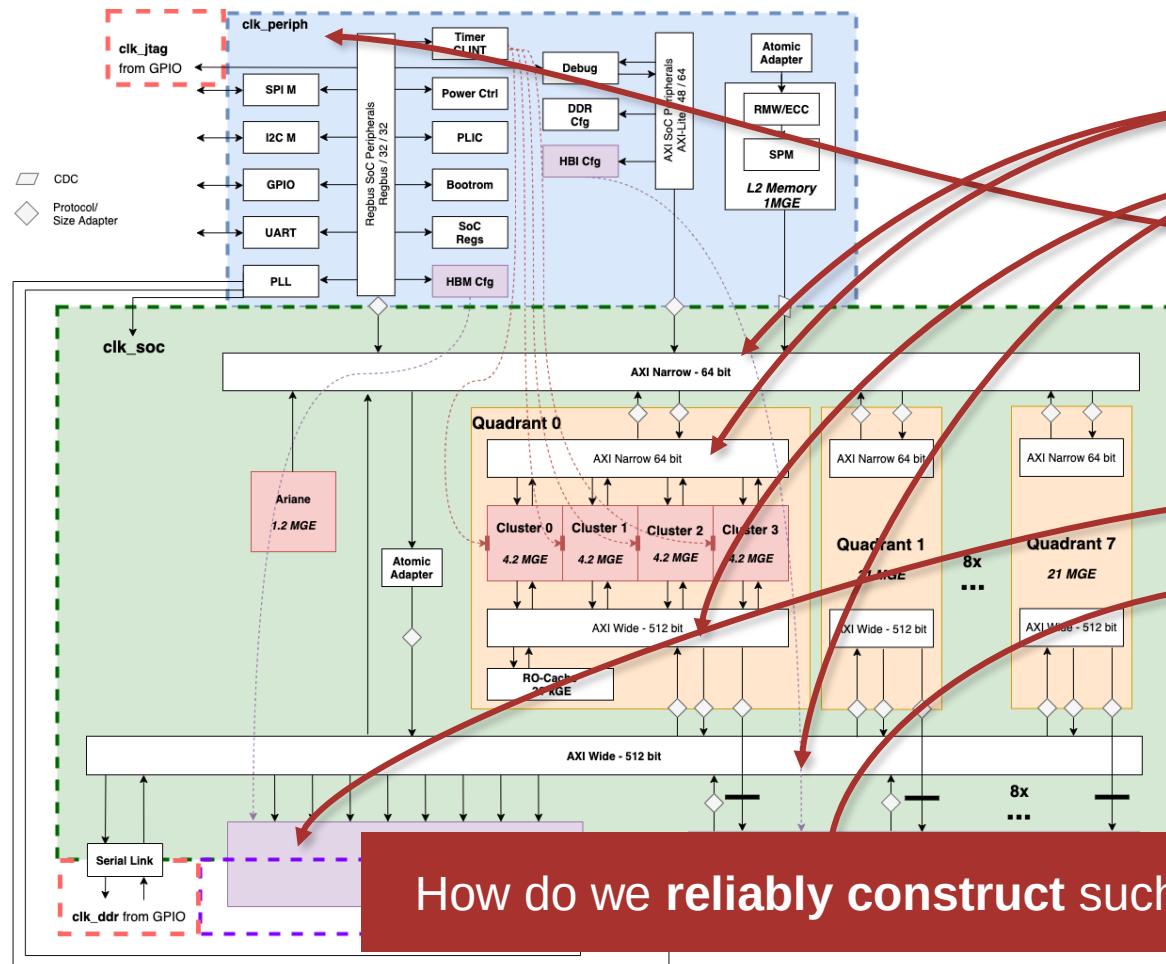


- Dual-chiplet configuration in 12nm FinFET
 - 8-10 L1 Quadrants, ~ 32 clusters/die
- Advanced high BW interfaces on interposer:
 - HBM2e for die-to-memory
 - 24 channel AIB for die-to-die
 - Controller developed in-house
- Architectural improvements:
 - SSR SIMD datatypes for ML
 - Sparsity support



Occamy

The SoC View



- 2 AXI Subsystems (multi-hierarchy)
 - 64-bit
 - 512-bit
- Peripherals
- Complex address space management
- 289 cores/die
- 8-channel HBI D2D link
- 8-channel HBM2e



Solder

Logical View

```
from solder import solder, device_tree, util

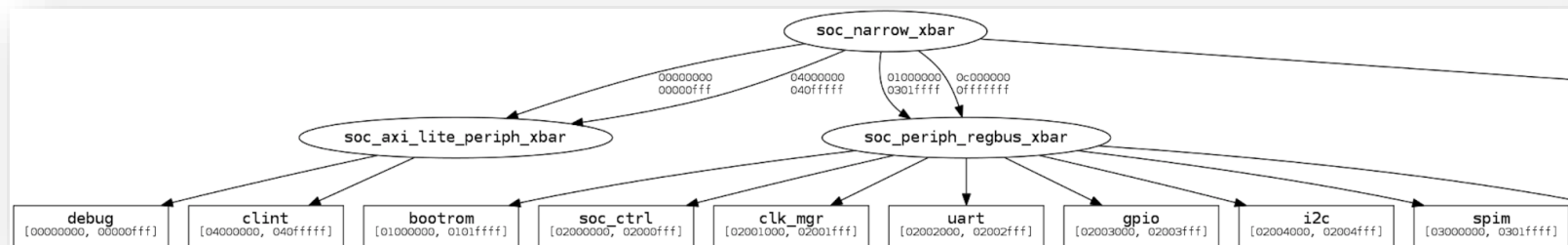
# Create the address map.
am = solder.AddrMap()

am_soc_narrow_xbar = am.new_node("soc_narrow_xbar")
am_debug = am.new_leaf("debug", 0x1000,
                       0x00000000).attach_to(am_soc_narrow_xbar)

# Implementation.
soc_narrow_xbar = solder.AxiLiteXbar(.., node=am_soc_narrow_xbar)
soc_narrow_xbar.add_input("debug", am_debug)

# Generate the Verilog code.
solder.render()
```

- Manual creation error prone
 - Diplomacy (Chisel)
 - [Solder](#) (Python lib)
- Works with AXI infrastructure
- Automatic routing rule creation
- Easily insert:
 - Protocol converters
 - CDCs
 - Isolation domains



Solder

Implementation View

- Based on [Mako templates](#)
- Insert bus modifiers, Solder checks IW, AW, DW, UW
- Bus types created automatically

```
<%
soc_narrow_xbar.out_regbus_periph \
  .cdc(context, "clk_periph_i", "rst_periph_ni", "periph_cdc") \
  .change_dw(context, 32, "axi_to_axi_lite_dw") \
  .to_axi_lite(context, "axi_to_axi_lite_regbus_periph") \
  .to_reg(context, "axi_lite_to_regbus_periph",
to=soc_regbus_periph_xbar.in_soc)
%>
```

Source

```
axi_a48_d64_i8_u0_req_t periph_cdc_req;
axi_a48_d64_i8_u0_resp_t periph_cdc_rsp;
```

```
axi_cdc #(
  .aw_chan_t (axi_a48_d64_i8_u0_aw_chan_t),
  ...
) i_periph_cdc (
  .src_clk_i (clk_i),
  ...
  .dst_resp_i(periph_cdc_rsp)
);
```

```
axi_a48_d32_i8_u0_req_t axi_to_axi_lite_dw_req;
axi_a48_d32_i8_u0_resp_t axi_to_axi_lite_dw_rsp;
```

```
axi_dw_converter #(
  ...
) i_axi_to_axi_lite_dw (
  ...
);
```

```
axi_lite_a48_d32_req_t axi_to_axi_lite_regbus_periph_req;
axi_lite_a48_d32_rsp_t axi_to_axi_lite_regbus_periph_rsp;
```

```
axi_to_axi_lite #(
  ...
) i_axi_to_axi_lite_regbus_periph_pc (
  ...
);
```

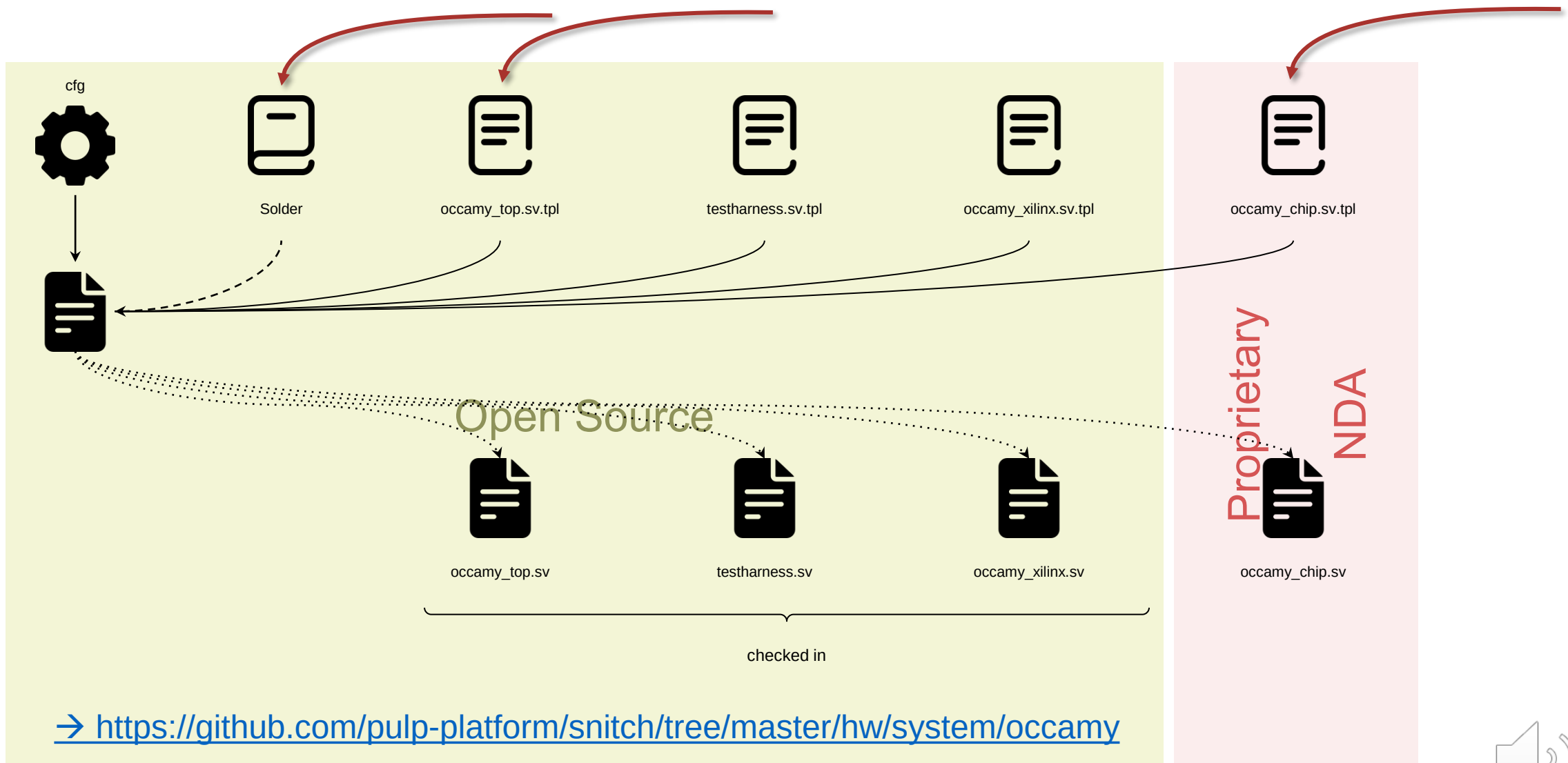
```
axi_lite_to_reg #(
  ...
) i_axi_lite_to_regbus_periph_pc (
  ...
);
```

GENERATED



Occamy

Generation Infrastructure



Final Words

Conclusion and Outlook

- **Manticore**: 4k, ultra-efficient, quad chiplet **concept**
- **Occamy**: dual-chiplet realization in GF12LP+ (open-source) [1]
- **Solder**: Infrastructure to generate complex SoC (open-source) [2]
- Based on open-source **AXI** infrastructure (see Thomas' talk before) [3]
- Can be easily re-used for your project!

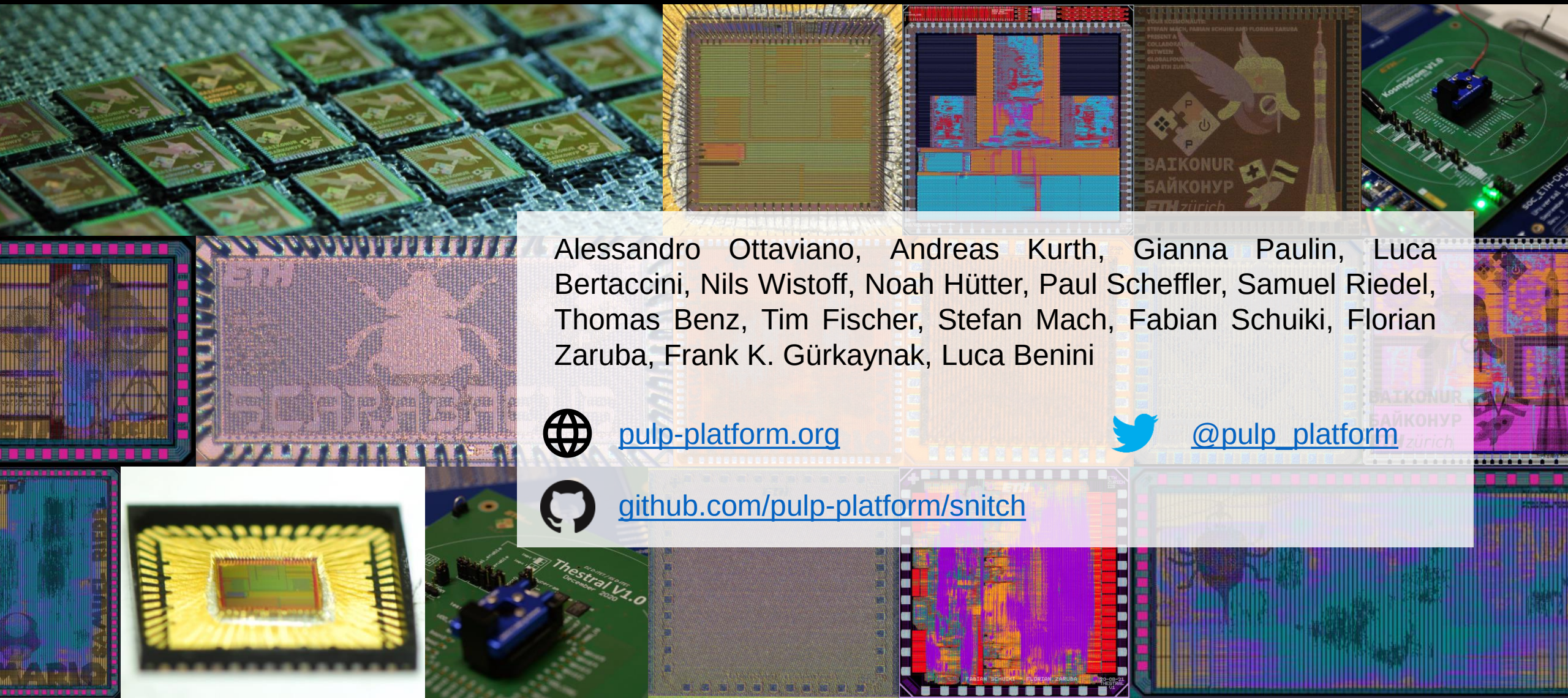
Questions?

[1] <https://github.com/pulp-platform/snitch/tree/master/hw/system/occamy>

[2] <https://github.com/pulp-platform/snitch/blob/master/util/solder/solder.py>

[3] <https://github.com/pulp-platform/axi>





Alessandro Ottaviano, Andreas Kurth, Gianna Paulin, Luca Bertaccini, Nils Wistoff, Noah Hütter, Paul Scheffler, Samuel Riedel, Thomas Benz, Tim Fischer, Stefan Mach, Fabian Schuiki, Florian Zaruba, Frank K. Gürkaynak, Luca Benini



pulp-platform.org



[@pulp_platform](https://twitter.com/pulp_platform)



github.com/pulp-platform/snitch