PAUL SCHERRER INSTITUT

PSI

Filip Leonarski :: Beamline Data Scientist :: Macromolecular Crystallography

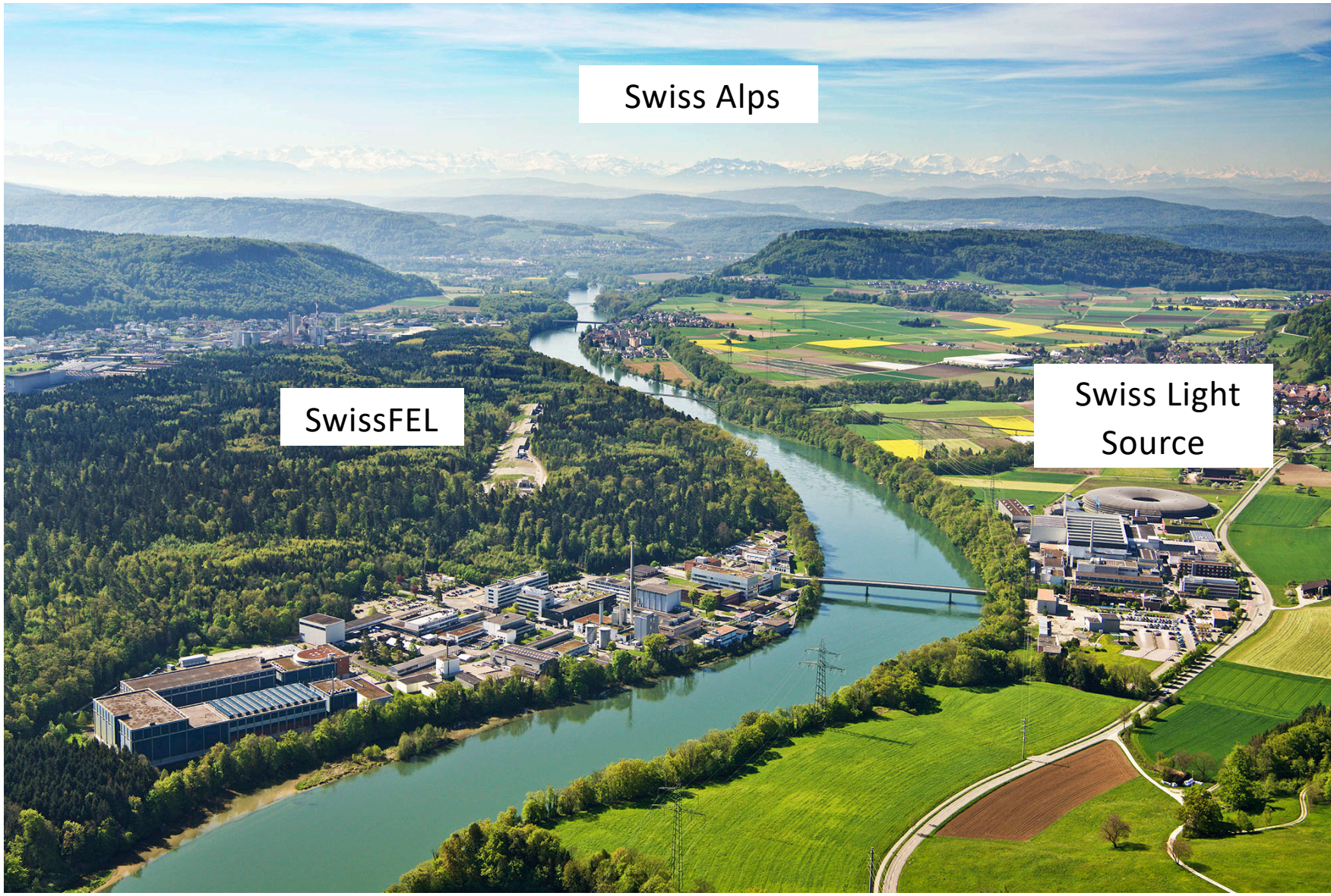
Boost your high bandwidth data acquisition by adding OpenCAPI and memory coherency to FPGA

PAUL SCHERRER INSTITUT

Plan

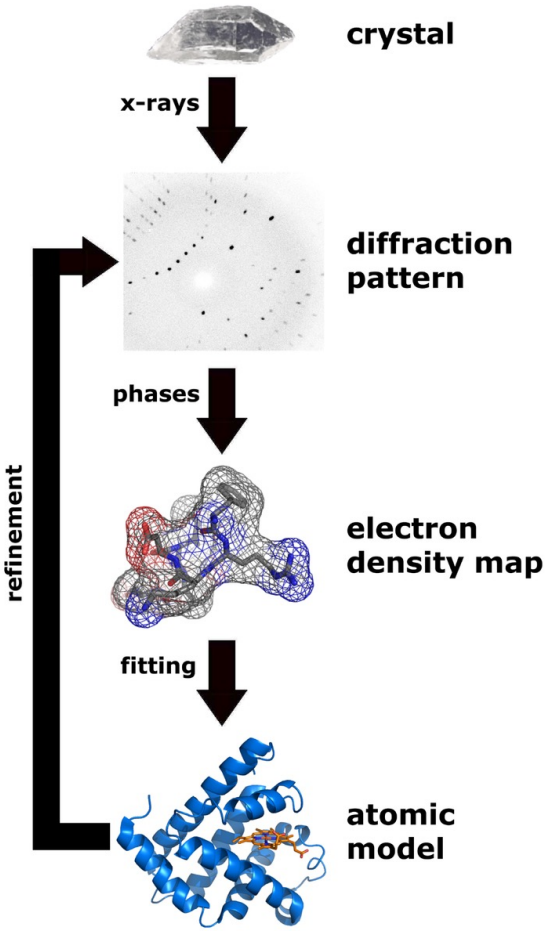
- **Introduction: Macromolecular crystallography at synchrotrons and X-ray detectors**
- **Technology: POWER + OpenCAPI**
- **Solution: Jungfraujoch**

Paul Scherrer Institute



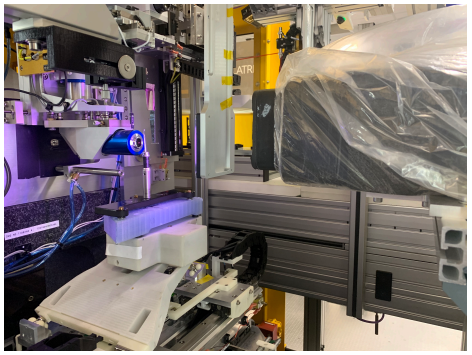
Macromolecular crystallography (MX)

- MX is a technique routinely used to determine 3D structure of proteins at synchrotron beamlines (e.g. ~50% users of Swiss Light Source)
- MX is widely used in structure-based drug discovery (including COVID-19)
- >140,000 high-resolution structures have been determined to date (biosync.org)
- Hybrid pixel X-ray detectors (e.g. PILATUS, EIGER) have make revolutionary impact on MX



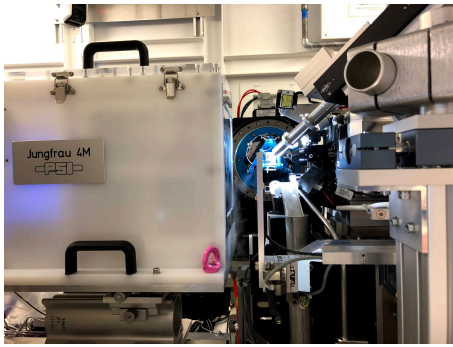
JUNGFRAU X-ray detector

- JUNFGRAU is a hybrid pixel detector with semiconductor sensor (silicon) and ASIC
- Pixel size: 75x75 µm
- Composed of modules, each ~500,000 pixels
- X-ray energy: 2-18 keV
- Frame rate: up to 2.2 kHz
- Designed for X-ray free electron lasers and synchrotrons
- Streams UDP packets over 10 GbE lines (2 x 10 GbE / module)



Test at **VMXi Diamond Light Source** (UK)

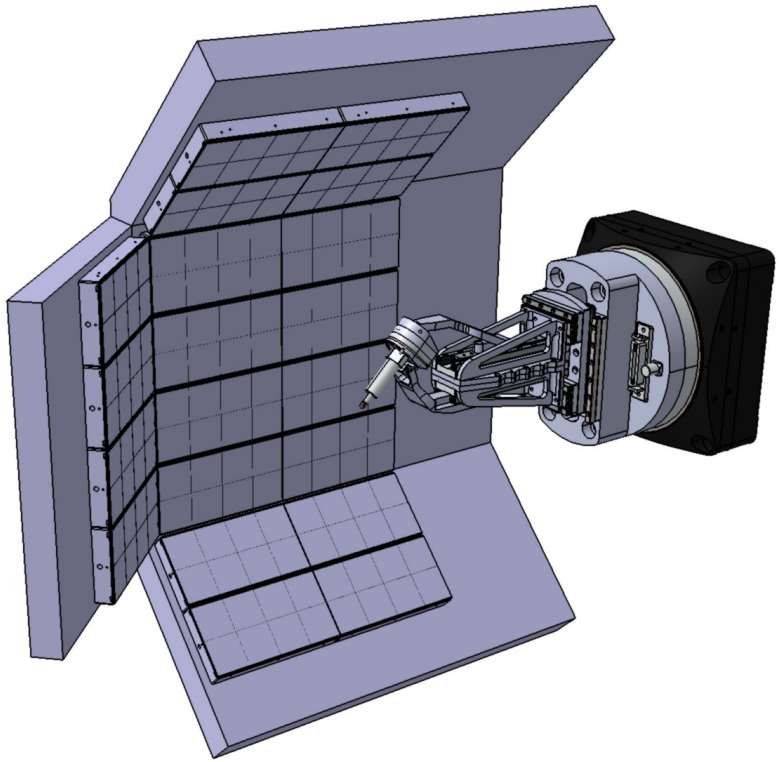
Test at **BL-1A Photon Factory KEK** (JP)



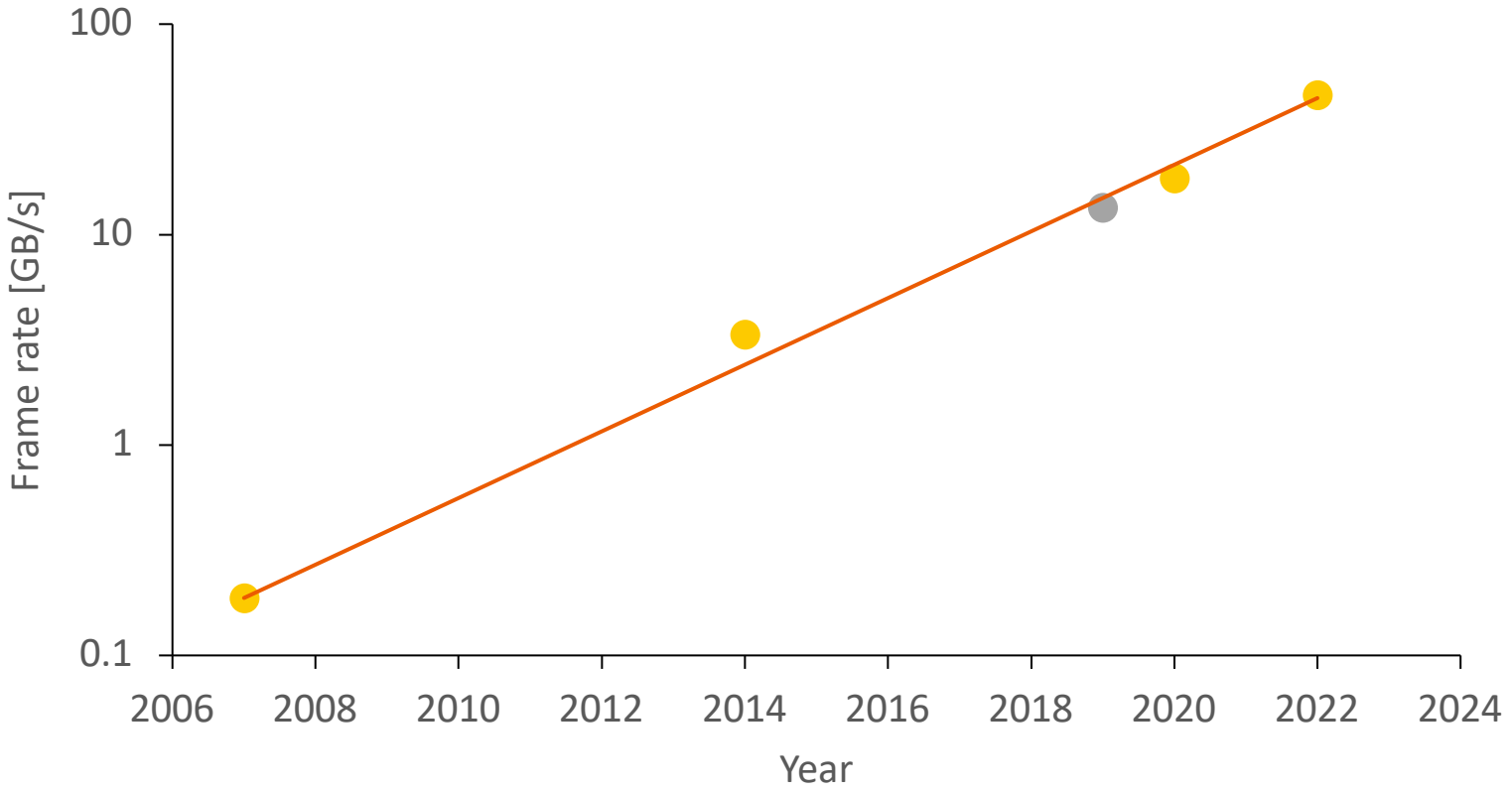
Test at **X06DA Swiss Light Source** (CH)

JUNGFRAU for Swiss Light Source

- **2021**
  - JUNGFRAU 4 Mpixel 2 kHz
    - Up to 18 GB/s data
- **2022**
  - JUNGFRAU 10 Mpixel 2 kHz
    - Up to 46 GB/s data
- Necessary functionality
  - Save diffraction images
  - Provide live feedback
    (do frames contain Bragg spots?)
- We save all the data
  (no community accepted method to reduce data on the fly)



MX detector data rates double every 2 years



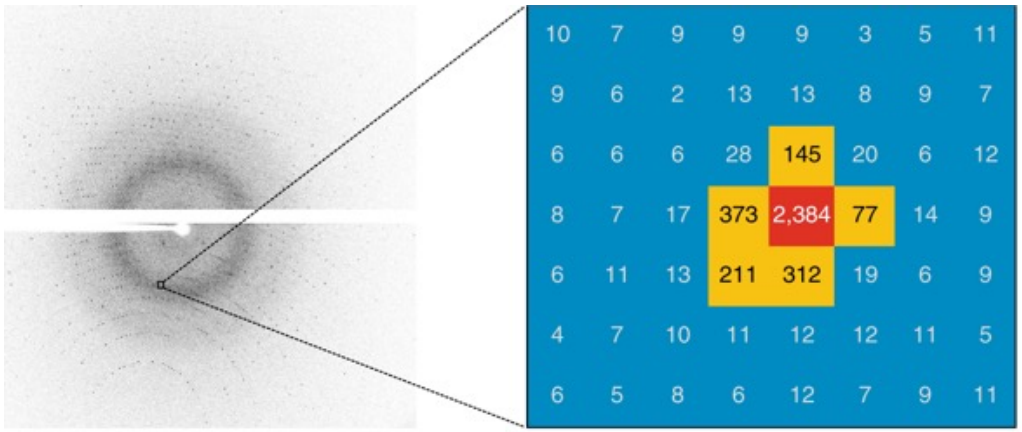
2007	PSI PILATUS	6 Mpixel	12.5 Hz	**0.2 GB/s**
2014	Dectris EIGER	16 Mpixel	133 Hz	**3.4 GB/s**
2019	Dectris EIGER 2 XE	16 Mpixel	400 Hz	**13.5 GB/s**
2020	PSI JUNGFRAU	4 Mpixel	2200 Hz	**18.4 GB/s**
2022	PSI JUNGFRAU	10 Mpixel	2200 Hz	**46.1 GB/s**

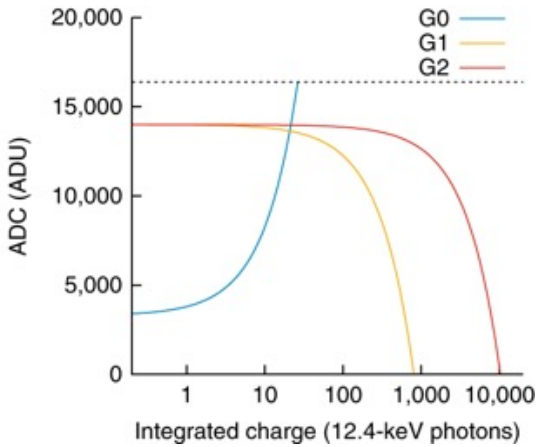
JUNGFRAU – adaptive gain charge integrating detector

- To maximize both sensitivity of detection and dynamic range, JUNGFRAU pixel has three different gain modes (G0, G1, G2)

- G0 is the most sensitive (low noise), but with dynamic range of about 30 photons
- G2 has dynamic range of >10,000 photons, but noise levels don't allow for single photon resolution

- Each pixel in each frame starts in G0 and dynamically switches to G1 and G2





F. Leonarski, S. Redford, A. Mozzanica, … , M. Wang
Nat. Methods, **15**, 799-804 (2018)

JUNGFRAU - conversion

- Each gain mode has its own dark current (pedestal) and gain constants
- Pixels have different sensitivities, so outcome needs to be adjusted
- Conversion procedure to find number of photons is:

```
for each pixel from 0 to N-1
    gain_bit = bits 15:14 from input[pixel]
    ADU      = bits 13: 0 from input[pixel]
    switch (gain_bit)
         case 00:
              output[pixel] = G0[pixel] * (ADU - P0[pixel])
         case 01:
              output[pixel] = G1[pixel] * (ADU - P1[pixel])
         case 11:
              output[pixel] = G2[pixel] * (ADU - P2[pixel])
    end switch
end for
```

JUNGFRAU – conversion is important

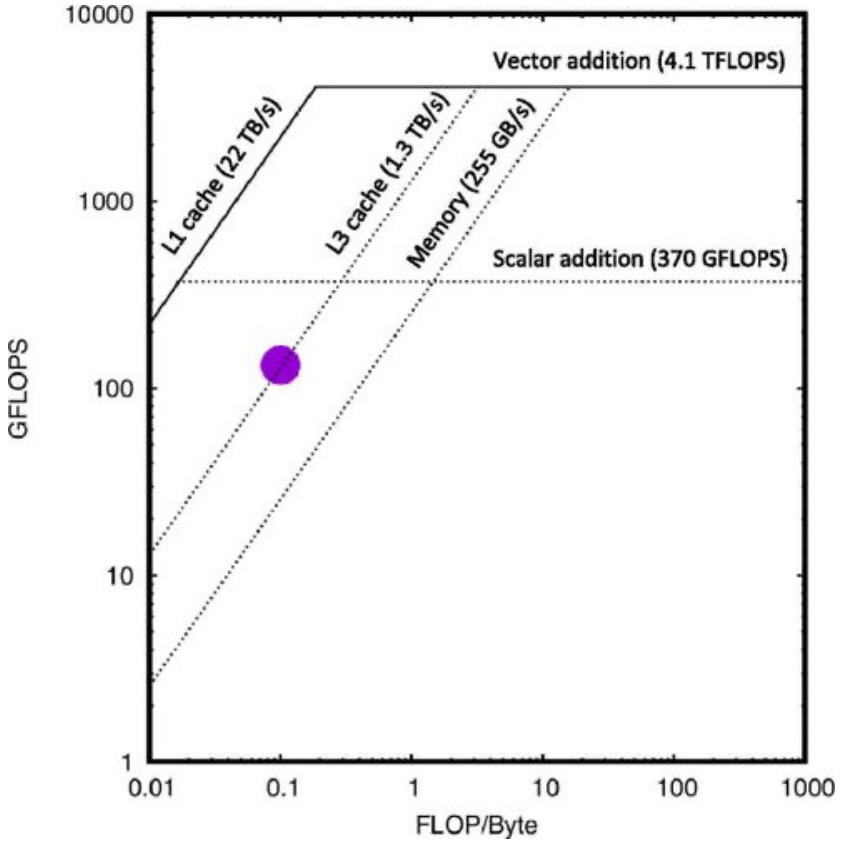
- Comparing ADUs before this procedure is comparing apples and oranges, if gain bits are set differently
  – summation, visualization, etc. are valid only operation after this procedure
  – compression is only efficient for converted data (as conversion cuts noise)

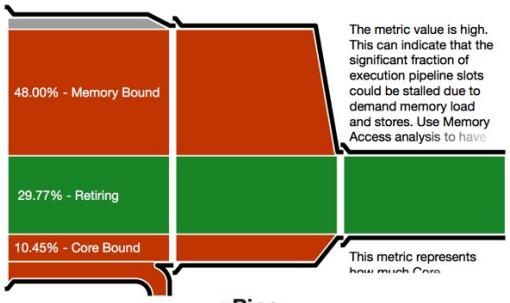
- To operate JUNGFRAU in a comfortable way, this conversion must happen real time

- **Aim: Conversion online at 1-2 kHz for the detector**

JUNGFRAU conversion – CPU profiling results

- Plot on the right presents profiling of conversion procedure, geometry expansion (512x1024 -> 514x1030), and compression (bshuf/LZ4)
- **Saturates 4 socket Intel Xeon server at around 500 Hz for 4 Mpixel detector**





Elapsed Time: 41.725s

SP GFLOPS: 73.079

Effective CPU Utilization: 61.7%
Average Effective CPU Utilization: 29.628 out of 48
Effective CPU Utilization Histogram

Memory Bound: 48.0% of Pipeline Slots
Cache Bound: 21.0% of Clockticks
DRAM Bound: 20.8% of Clockticks
NUMA: % of Remote Accesses: 0.0%

FPU Utilization: 1.3%
SP FLOPs per Cycle: 0.803 Out of 64
Vector Capacity Usage: 97.8%
FP Instruction Mix:
% of Packed FP Instr.: 97.7%
% of 128-bit: 0.1%
% of 256-bit: 0.0%
% of 512-bit: 97.6%
% of Scalar FP Instr.: 2.3%
FP Arith/Mem Wr Instr. Ratio: 0.417

See: "JUNGFRAU detector for brighter x-ray sources: Solutions for IT and data science challenges in macromolecular crystallography" Leonarski et al. Structural Dynamics (2019) https://doi.org/10.1063/1.5143480

JUNGFRAU - receiving

- The results on the previous slide were not including UDP receiving, which is also a challenge
- It would be only worse:
  - Data from network card travel between kernel buffers, increasing memory bandwidth needs
  - Competition for memory bandwidth and CPU cache
- We had to look for another solution to this problem (massively parallel solution would be not sustainable)

PAUL SCHERRER INSTITUT

PSI

POWER / OpenCAPI / FPGA architecture

FPGA are perfect devices for data acquisition

- **Real-time performance**
  - FPGA design is cycle-accurate, with fixed latency and throughput
- **Large memory throughput**
  - FPGAs with HBM2 have 460 GB/s bandwidth to 8 GB large memory
- **Ethernet on-board**
  - FPGA are made to work with network, often having dedicated “hard” cores for ethernet
- **But development of FPGAs is difficult and time consuming**
  - Hardware description languages
  - Need to be Linux kernel expert

High-level synthesis

- C/C++ compiler to produce hardware design language (Verilog or VHDL)
- All code is valid C++ code, it can be executed on CPU and functionally is generally equivalent (besides parallelism)
- Dedicated pragma to guide FPGA synthesis
- It is generally understandable for software developers, but code may look strange

```
template<int N> ap_uint<16*N> shuf16(const ap_uint<16*N> in) {
#pragma HLS INLINE
#pragma HLS PIPELINE
    ap_uint<16*N> out;
    for (int i = 0; i < N * 16; i++)
        out[(i%16) * N + (i/16)] = in[i];
    return out;
}
```

Bitshuffle for 16-bit numbers

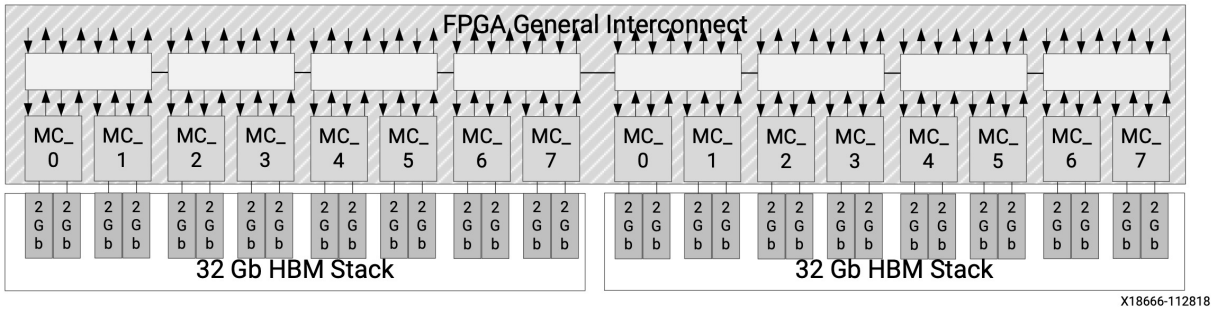
```
+ Detail:
    * Instance:
    +----------------------+---------+---------+---------+-----------+-----------+-----+-----+----------+
    |                      |         |  Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline |
    |       Instance       |  Module |   min   |   max   |    min    |    max    | min | max |   Type   |
    +----------------------+---------+---------+---------+-----------+-----------+-----+-----+----------+
    |grp_convert_fu_609    |convert  |        9|        9| 22.500 ns | 22.500 ns |    1|    1| function |
    +----------------------+---------+---------+---------+-----------+-----------+-----+-----+----------+

    * Loop:
    +----------------------+---------+---------+----------+-----------+-----------+-----------------+----------+
    |                      |  Latency (cycles) | Iteration|  Initiation Interval  |      Trip       |          |
    |      Loop Name       |   min   |   max   |  Latency |  achieved |   target  |      Count      | Pipelined|
    +----------------------+---------+---------+----------+-----------+-----------+-----------------+----------+
    |- Loop 1              |       32|       32|         1|          -|          -|               32|    no    |
    |- save_gainG0         |        0|  1073725440|         2|          1|          1| 0 ~ 1073725440  |    yes   |
    |- save_gainG1         |        0|  1073725440|         2|          1|          1| 0 ~ 1073725440  |    yes   |
    |- save_gainG2         |        0|  1073725440|         2|          1|          1| 0 ~ 1073725440  |    yes   |
    |- save_pedeG1         |        0|  1073725440|         2|          1|          1| 0 ~ 1073725440  |    yes   |
    |- save_pedeG2         |        0|  1073725440|         2|          1|          1| 0 ~ 1073725440  |    yes   |
    |- pixel_conversion    |        ?|        ?|       153|          4|          4|                ?|    yes   |
    +----------------------+---------+---------+----------+-----------+-----------+-----------------+----------+
```

HLS compiler can pipeline functions/loops to fix latency and throughput

High-bandwidth memory

- Available in Xilinx Virtex Ultrascale+
- For VU33/35P:
  - Size: 8 GB
  - Bandwidth: **up to 460 GB/s**
  - Latency (worst case): up to 1 microsecond
- Complex architecture
  - 32 x 256-bit AXI3 interfaces
    - Either operating as 32 separate memories
    - or as single memory with crossbar (at the cost of up to 50% throughput)



PCI Express DMA

- PCI Express is an industry standard peripheral bus

- PCI Express direct memory access (DMA) is operating on physical addresses:

  ⇒ need to maintain own driver
  ⇒ translation to virtual addresses is responsibility of developer
  ⇒ need understanding of CPU and kernel memory mechanisms (streaming memory vs. consistent memory, pinning, cache coherency)

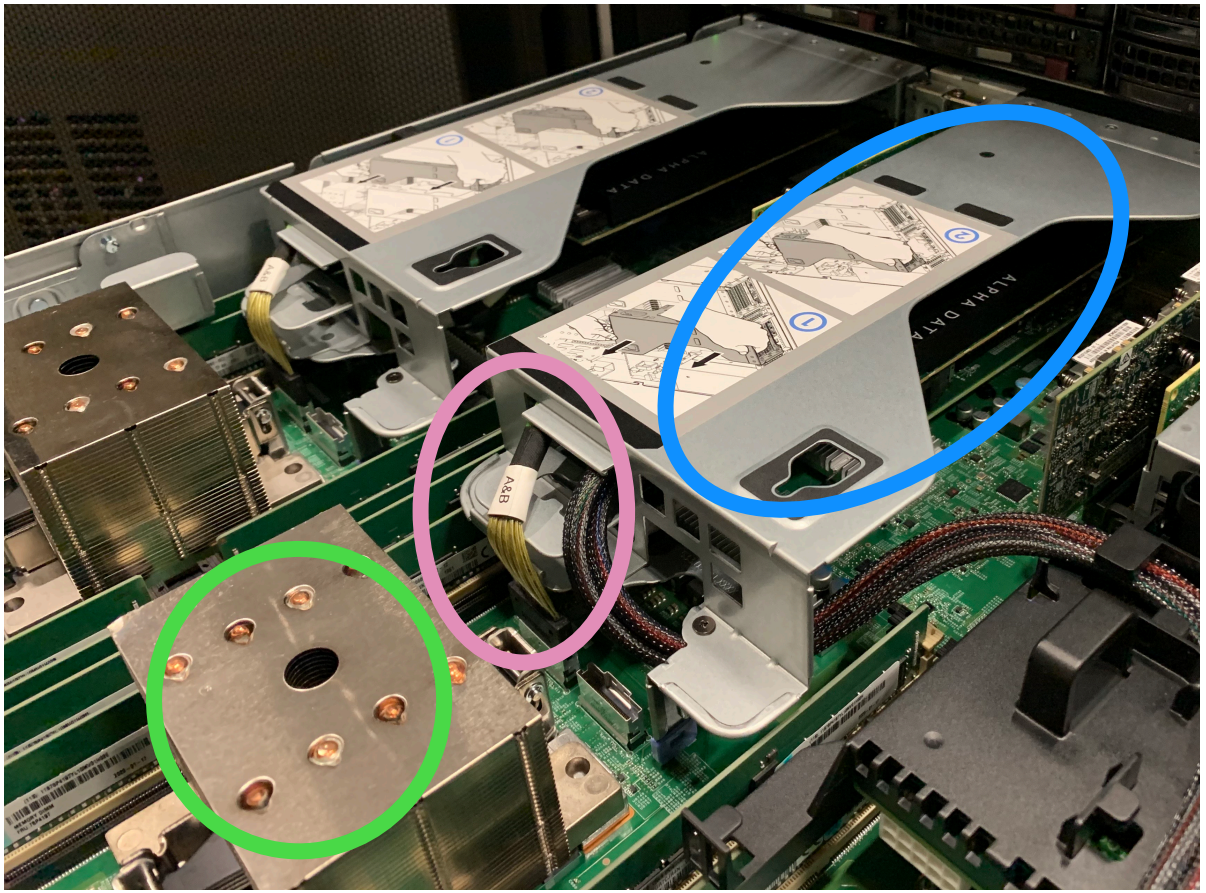
- Only limited number of devices can benefit from PCIe Gen4 standard
  (not many FPGAs with G4 x16)

POWER architecture

- IBM POWER9 showed great numbers for I/O and memory throughput in Summit and Sierra supercomputers
- IBM designed own memory coherent interface for accelerators (CAPI/OpenCAPI), which has advantages over PCIe

Source: Wikipedia

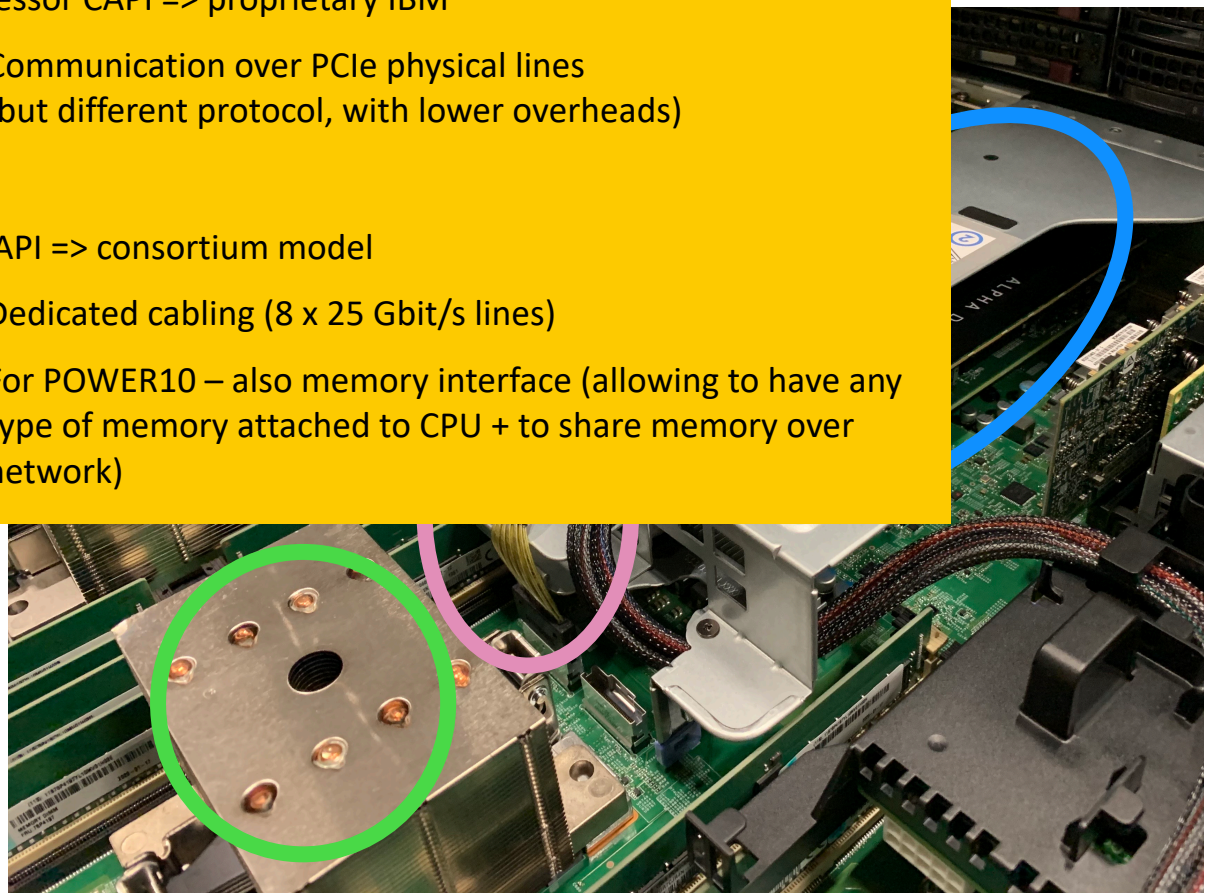
OpenCAPI



POWER9 CPU **OpenCAPI cable** **FPGA board**

OpenCAPI

- Predecessor CAPI => proprietary IBM
  - Communication over PCIe physical lines
    (but different protocol, with lower overheads)
- OpenCAPI => consortium model
  - Dedicated cabling (8 x 25 Gbit/s lines)
  - For POWER10 – also memory interface (allowing to have any type of memory attached to CPU + to share memory over network)



POWER9 CPU **OpenCAPI cable** **FPGA board**

What difference brings OpenCAPI?

- Similar difference what 80286/80386 virtual mode brought to software development
- In OpenCAPI one needs single kernel operation => Attach accelerator to running process
- Then, accelerator has access to virtual address space of running process – it is FPGA that is initiating the communication
- All security/reliability/efficiency/coherency mechanisms in CPU and kernel are available transparently to OpenCAPI attached accelerator

Source: Wikipedia

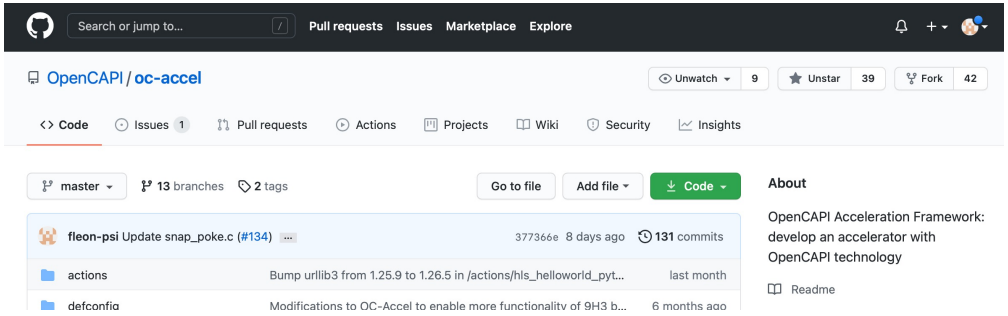
PAUL SCHERRER INSTITUT

How to develop with OpenCAPI?

- Main function for the action contains a pointer to virtual address space
  - On device the pointer will be synthesized as 1024-bit master memory-mapped AXI interface
  - On CPU this pointer has to be just set to zero (which is first address of virtual address space)
- Any cell in virtual memory is just accessed as offset from this pointer
- Only requirement is that memory is aligned to 128-bytes
  - No special memory allocator, malloc or mmap is fine
  - No pinning/registering
- The same memory buffer class for both simulation and working with device
- There is also 4 MiB memory-mapped register space (like PCIe BAR)
  - On device implemented as slave AXI-lite (32-bit)

OC-Accel

- Open source “shell” maintained by IBM
- **http://github.com/OpenCAPI/oc-accel**
- Provides ready made tool to work with OpenCAPI (from transceiver setup to AXImm bridge)
- Provides preconfigured interfaces for I/O peripherals (HBM, 100G, NVMe)
- Provides simulation environment
  - One can simulate both SW and HW in a single simulation (both user FPGA design and software are not modified from their “real” implementation)



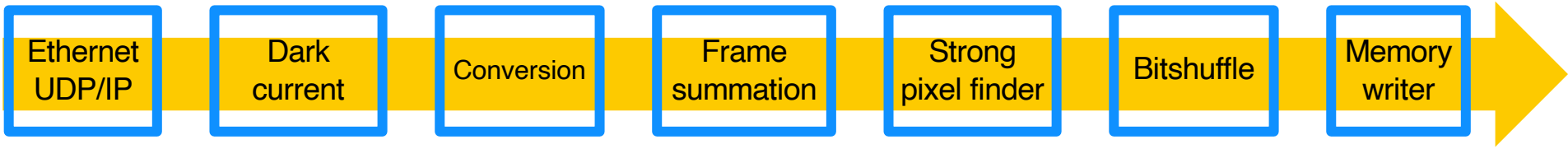
PAUL SCHERRER INSTITUT

PSI

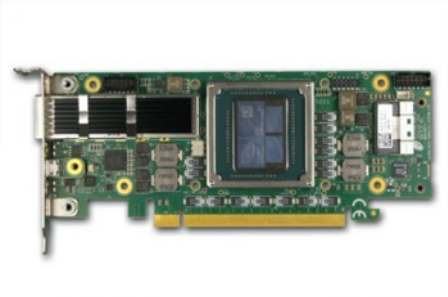
Jungfraujoch – FPGA implementation

Jungfraujoch server

Up to **50 GB/s acquisition and data analysis** in a single 2U IBM POWER9 server with 1-4 FPGA boards



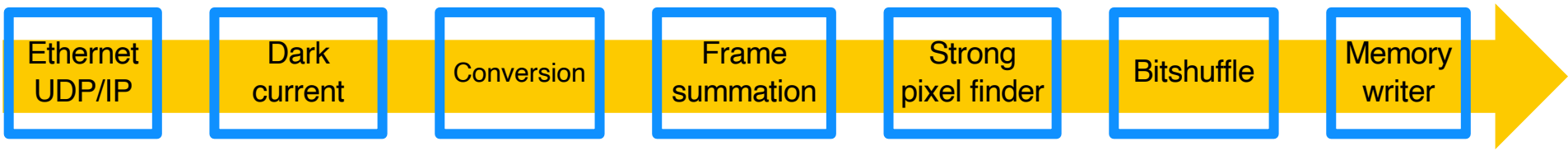
FPGA board with OpenCAPI interface



- Data acquisition
- Initial data analysis
- Pre-compression

(2.5 Mpixel/board for JF)

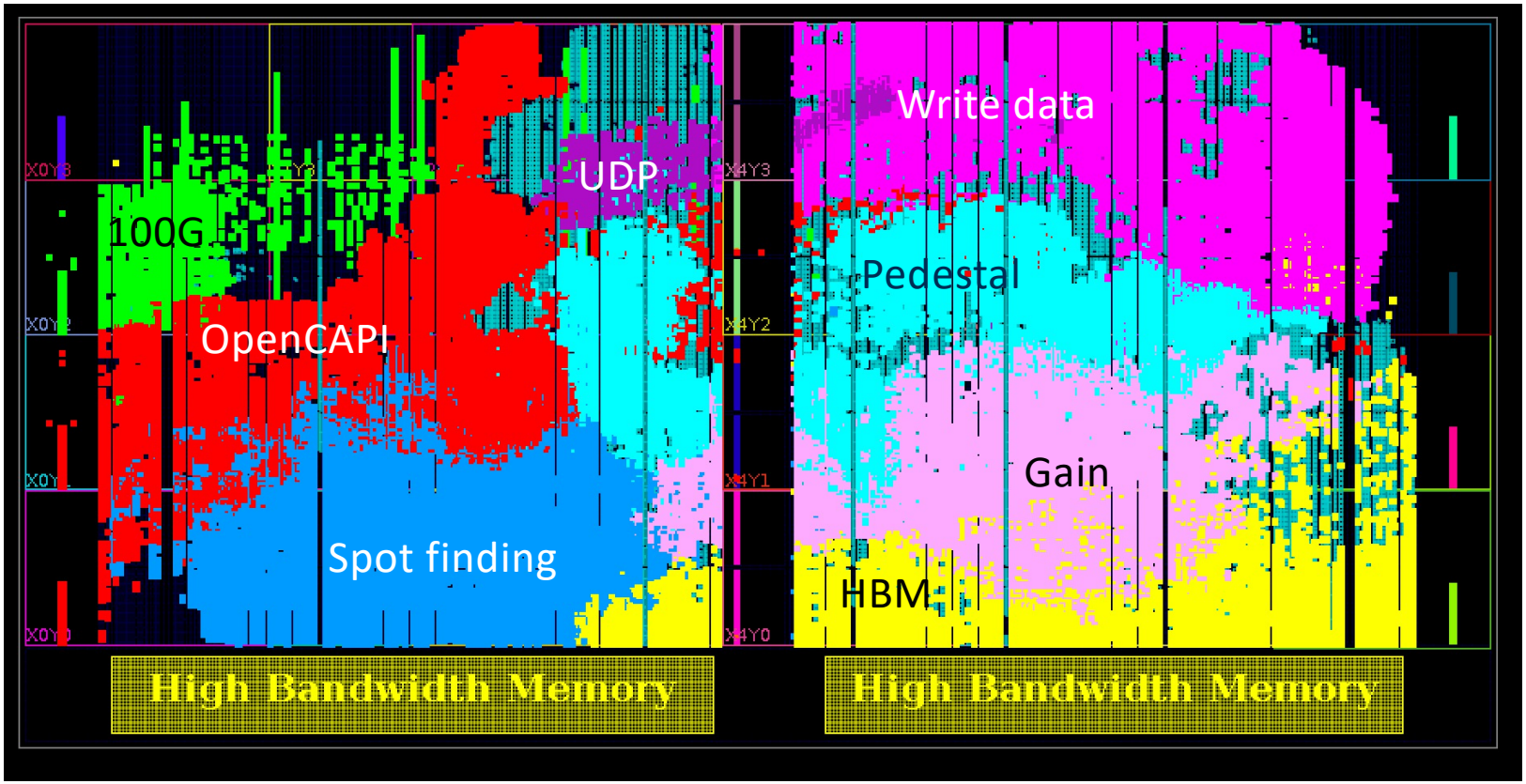
Jungfraujoch FGPA streaming design



Modular design

- Stream of data handled by successive cores doing work in parallel
  → **throughput and latency of each core is determined by the hardware design**
- Extra stages can be relatively simply added, option to bypass cores
- All cores are C++ functions, connected with AXI-Stream FIFOs
- As buffering is expensive on FPGA, it is best suited for algorithm that have limited dependencies between frames

Jungfraujoch implementation on VU33P FPGA



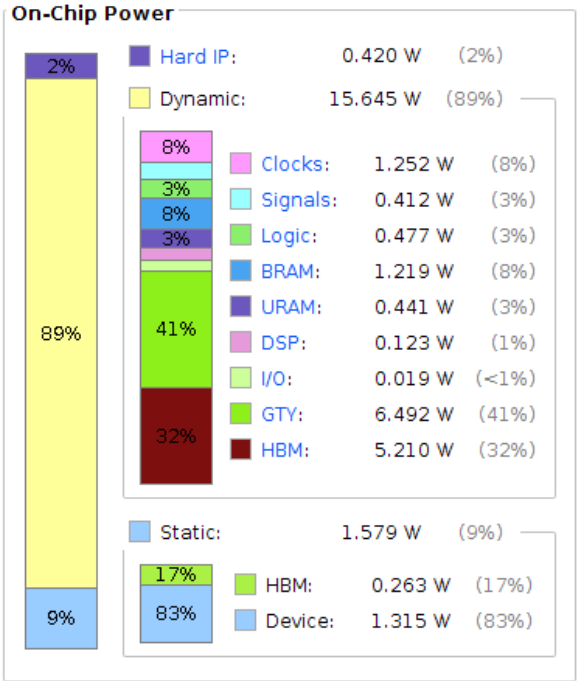
Jungfraujoch FPGA power usage is 18 W/board for the whole streaming functionality

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**17.666 W**
FPGA Power:	14.566 W
HBM Power:	3.1 W
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**34.4°C**
Thermal Margin:	65.6°C (119.7 W)
Effective θJA:	0.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



Xilinx Vivado Power Report

2 boards for 4 Mpixel JUNGFRAU and 4 boards for 10 Mpixel JUNGFRAU

Testing of Jungfraujoch

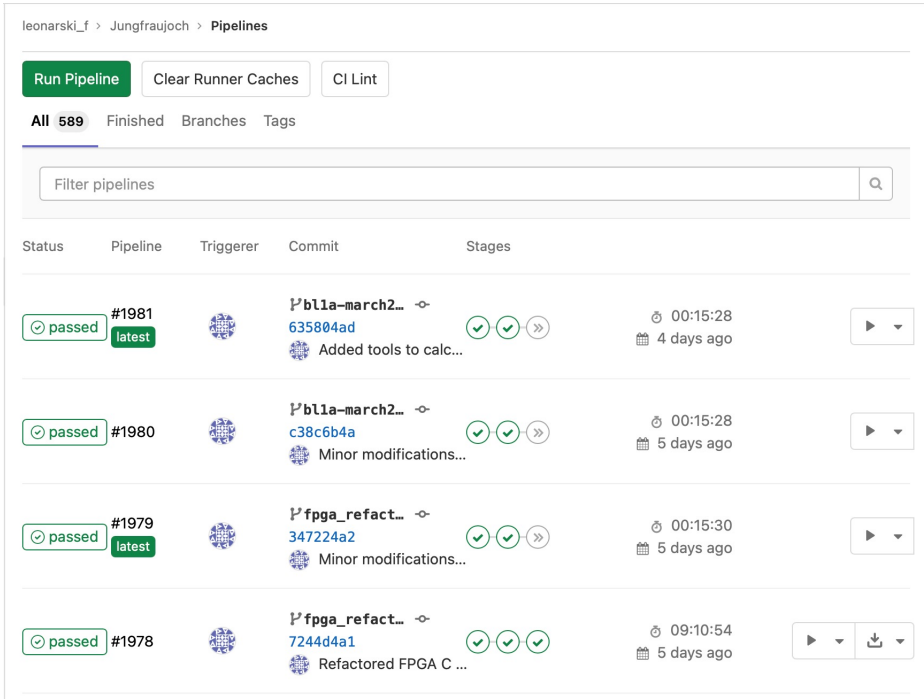
Software tests	Hardware simulation
Seconds	Hours
GCC + Catch2 framework	OCSE + Cadence Xcelium
C++	Hardware description language
Can cover multiple functionalities and scenarios (> 90% HLS code coverage)	Very close to real hardware behavior
Code can still fail on device, due to deadlock in FIFOs	Can only test 1-2 functionalities in reasonable time

OCSE = OpenCAPI simulation engine

Allows to simulate fully functional OpenCAPI interconnect on x86 system, can be run with multiple HW simulators

Testing of Jungfraujoch

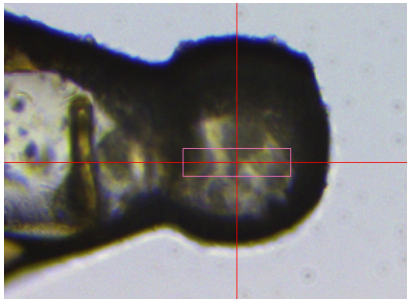
- Software tests are cheap, can be done from C++ IDE while working on the code
- Both can be scripted for CI pipeline, e.g. Gitlab at PSI
- Success of software tests is prerequisite to run hardware tests
- After hardware tests, FPGA image is built (both for on device testing and to know if there is any problem in timing closure)

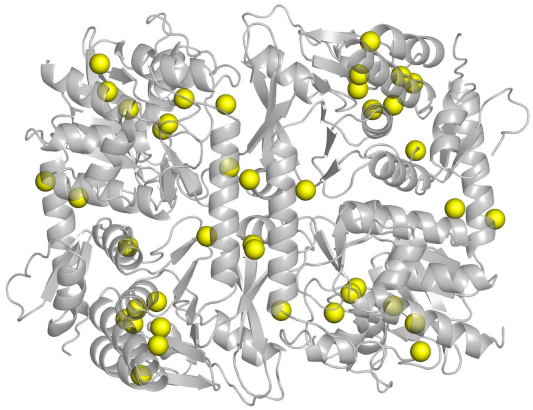


Commissioning in KEK (Jan – May 2021)

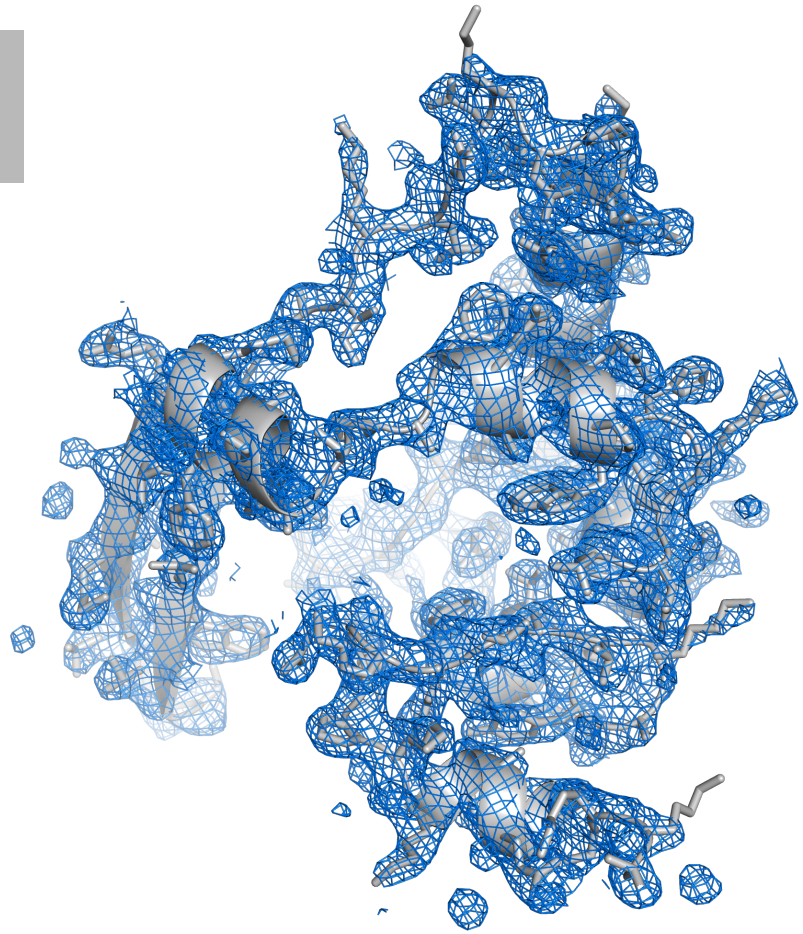
- Detector and data acquisition system was sent in November for an experiment in Photon Factory, KEK
- More than 2,000 datasets collected for protein targets, few real-life native-SAD structures solved
- Due to pandemic, detector support and development (including deployment of new FPGA design) was done fully remotely from Switzerland

BL-1A Photon Factory
JUNGFRAU detector (up) tested in helium chamber for native-SAD measurements with 3.75 keV X-rays



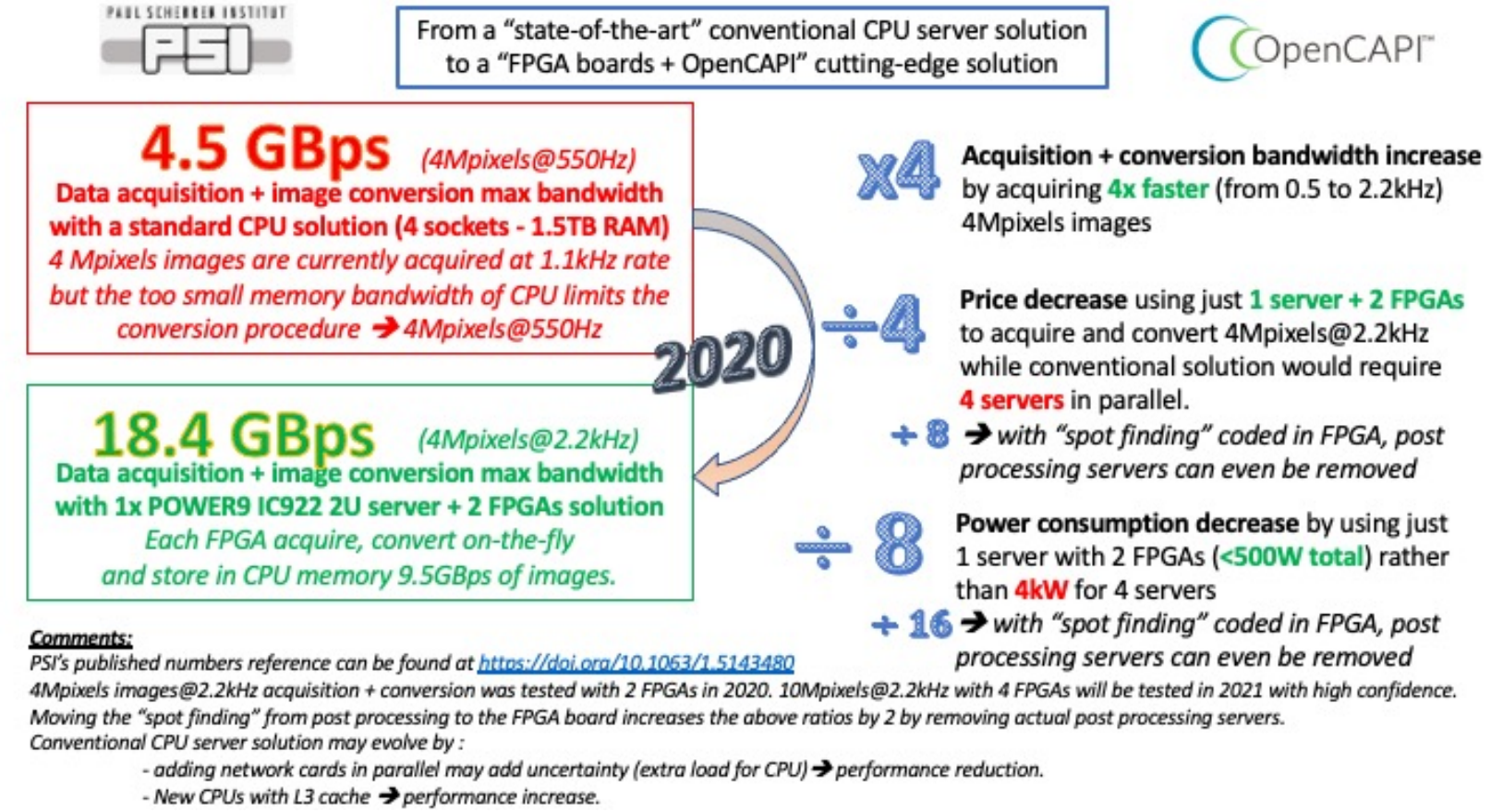


Structure of Nucleocapsid Phosphoprotein from SARS-CoV-2 solved in 1 second



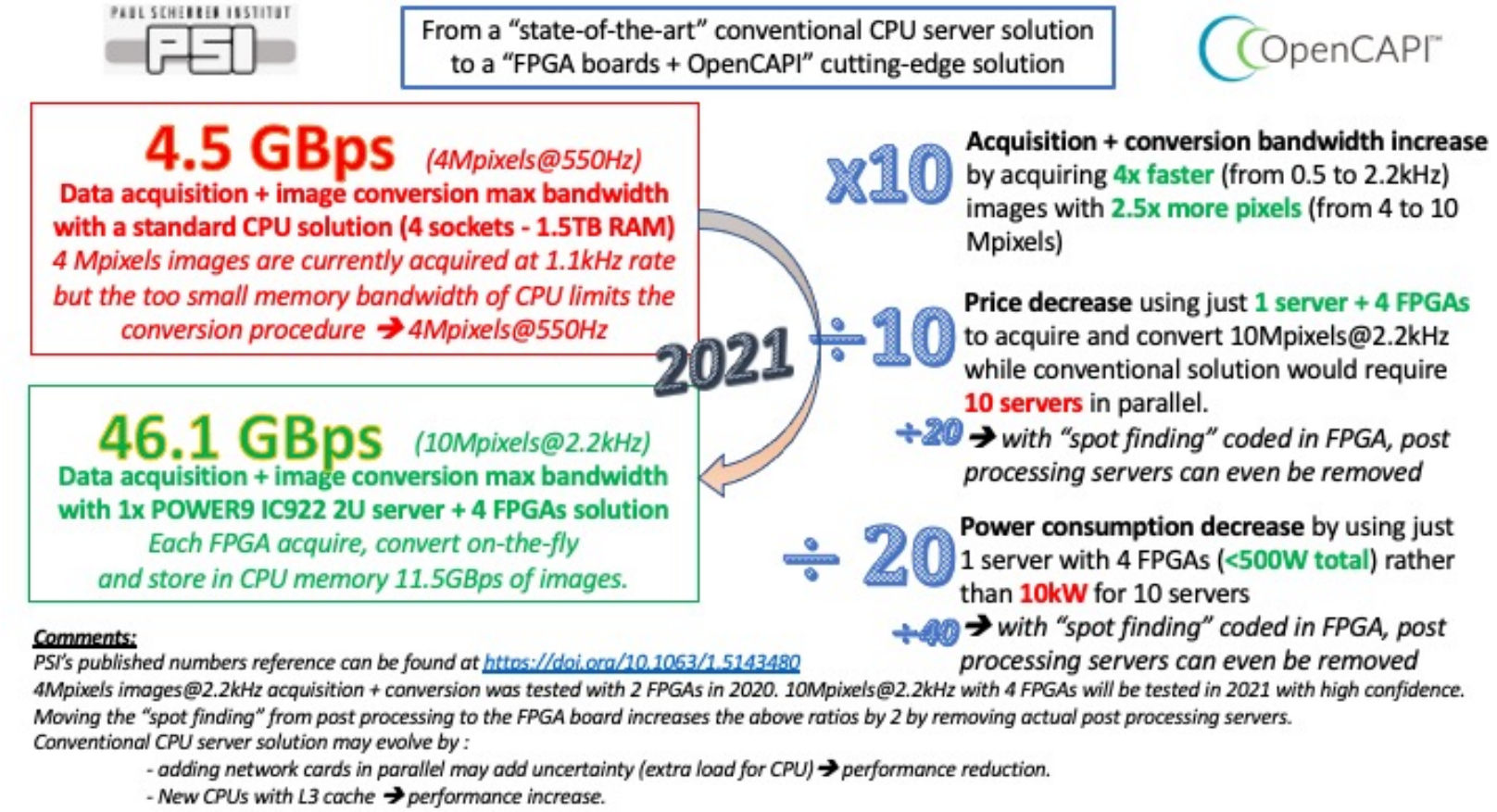
- Crystal was previously measured with conventional setup at our beamline – with measurement taking longer than **one minute**
- With JUNGFRAU detector and OpenCAPI readout, 2000 images collected in **one second** allowed to solve structure of this protein
- Experimental team: Filip Leonarski, Sylvain Engilberge, Vincent Olieric, Meitian Wang (MX Group), Aldo Mozzanica (PSI Detector Group)
- SARS-CoV-2 protein was produced by Zinzula, L., Basquin, J., Bracher, A., Baumeister, W. (MPI, Martinsried)

Possible gain from using FPGA based system



Courtesy: B. Mesnet (IBM)

Possible gain from using FPGA based system



Courtesy: B. Mesnet (IBM)

Acknowledgements

MX Group (PSI)
- Vincent Olieric
- Takashi Tomizaki
- Chia-Ying Huang
- Sylvain Engilberg
- Justyna Wojdyła
- Meitian Wang

Detector Group (PSI)
- Aldo Mozzanica
- Martin Brückner
- Carlos Lopez-Cuenca
- Bernd Schmitt

Science IT (PSI)
- Leonardo Sala

Controls (PSI)
- Andrej Babic
- Leonardo Hax-Damiani

SLS management (PSI)
- Oliver Bunk

Photon Factory, KEK
- Naohiro Matsugaki
- Yusuke Yamada
- Masahide Hikita

MAX IV
- Jie Nan
- Zdenek Matej

Uni Konstanz
- Kay Diederichs

LBL
- Aaron Brewster

DLS
- Graeme Winter

ESRF
- Jerome Kieffer

CERN
- Niko Neufeld

IBM Systems (France)
- Alexandre Castellane
- Bruno Mesnet

InnoBoost SA
- Lionel Clavien