FPGAs in HPC applications and methods

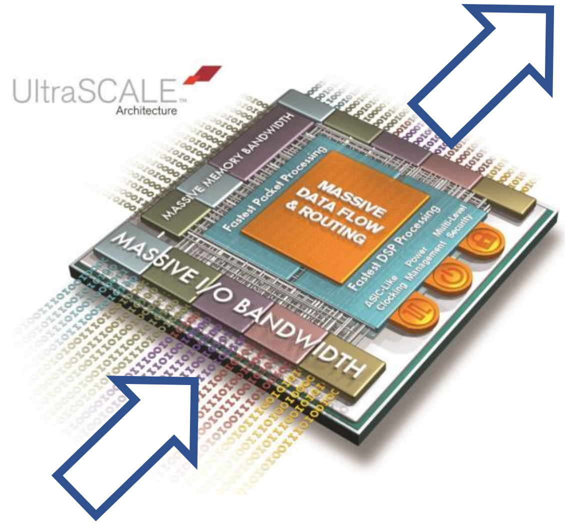
Grzegorz Korcyl

Department of Information Technologies
Jagiellonian University, Cracow

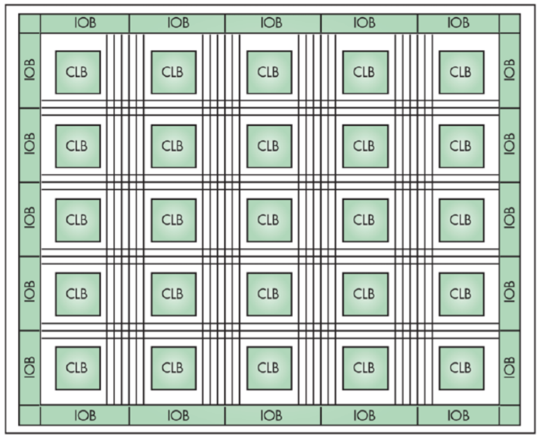
June 2024
WMLQ 2024

What are FPGAs

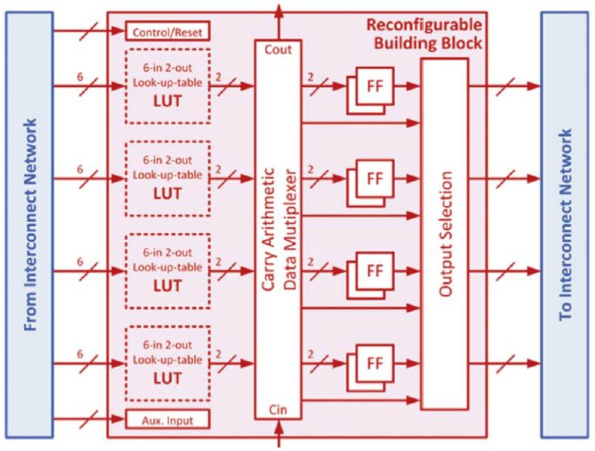
- Field Programmable Gate Arrays
  - Adaptable computing resources
  - Reconfigurable at any time
  - Devices for processing digital data streams



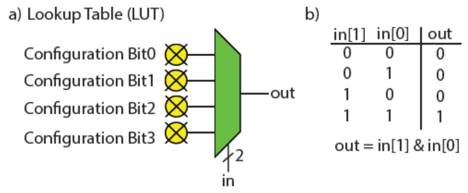
Arrays of Configurable Logic Blocks



Basic Configurable Logic Block


From Interconnect Network
Control/Reset
6-in 2-out Look-up-table LUT
Aux. Input
Cout
Cin
Carry Arithmetic Data Mutiplexer
FF
Output Selection
Reconfigurable Building Block
To Interconnect Network
6
2

www.electronicdesign.com

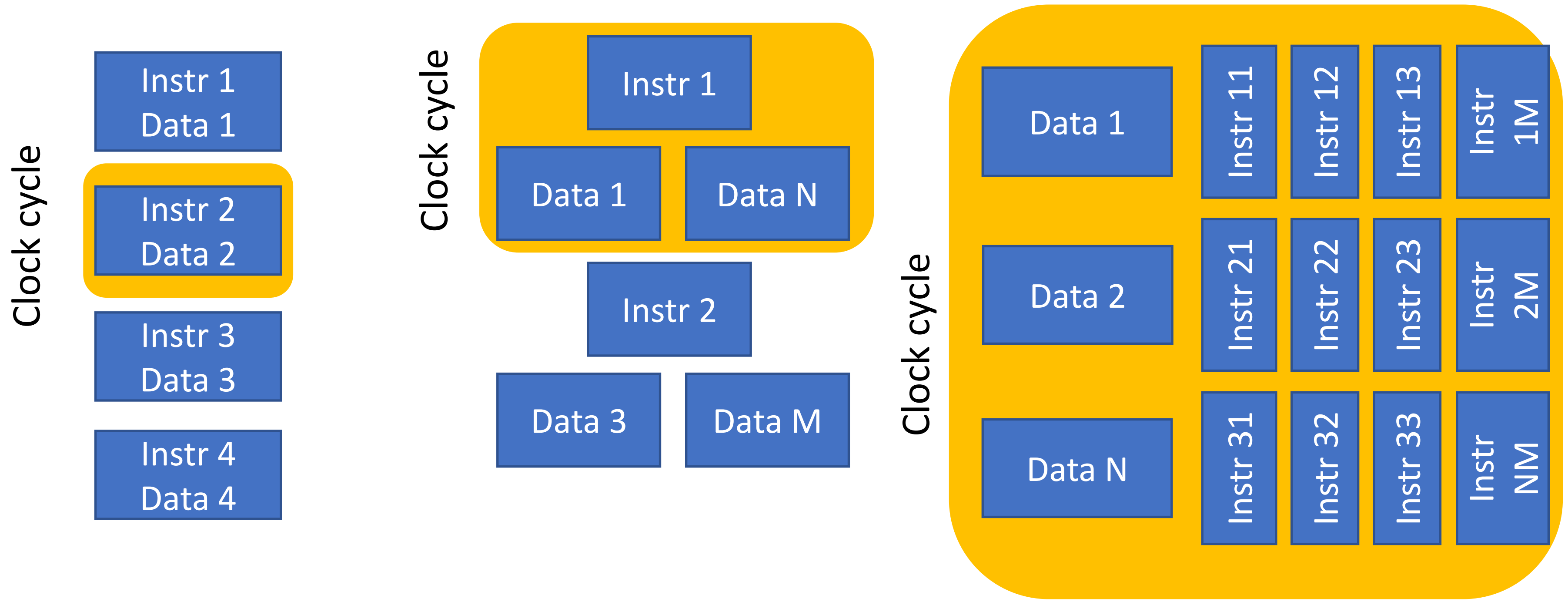


in[1]	in[0]	out
0	0	0
0	1	0
1	0	0
1	1	1

out = in[1] & in[0]

R. Kastner, J. Matai, S. Neuendorffer „Parallel Programming for FPGAs"

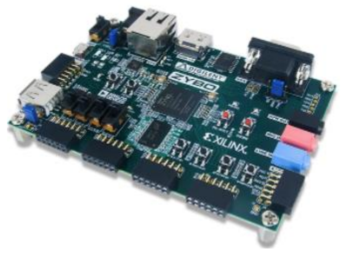
CPU vs GPU vs FPGA



- CPU
  - Single Instruction Single Data per core
  - Fixed instruction set
  - Multiple cores
  - High clock freq.
  - Operating system
- GPU
  - Single Instruction Multiple Data
  - Fixed instruction set
  - High clock freq.
  - Memory access
  - Accelerates CPU
- FPGA
  - Flexible architecture
  - Massive parallelism
  - Streamlined processing
  - Low clock freq.
  - Instant memory access
  - Standalone platforms

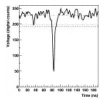
Devices

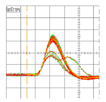
- **First FPGA introduced by Xilinx in 1985**
  - 64 Configurable Logic Blocks with programmable interconnect
  - Addition of two integers consumes ~30 CLB
  - Design built manually by connecting logic gates



- **Typical applications**
  - Interfacing with digitization devices (TDCs, ADCs)
  - Sensor Fusion
  - True real-time device control e.g. servo-motors,
  - Network streams processing

- **In experimental physics**
  - Digitization of analog signals
  - Preprocessing of digitial data streams (e.g. filters, zero-reduction, feature extraction)
  - Control and monitoring of electronics
  - High-speed data transmission
  - Low-level, fast data selection (trigger)





```
0000  ff ff ff ff ff ff 00 00
0010  01 1c 0b 00 00 00 ff 11
0020  ff ff c3 50 c3 50 01 08
0030  00 02 01 03 00 00 00 00
0040  00 02 06 01 00 02 06 02
```

FPGAs and GPUs

- **FPGAs after 40 years**
  - Market value from 10$ bilion in 2020 to estimated 30$ bilion in 2030
  - GPU market value from 25$ billion in 2020 to estimated 300$ bilion in 2030
- **GPU breakthrough into mainstream computing**
  - Generic PCIe interface
  - Superior performance in specific applications
  - Programming environment – CUDA introduced in 2007, Tensorflow in 2017
- **Current FPGA situation**
  - PCIe accelerator cards since 2018
  - Largest device over 2 millions CLBs
  - First mainstream C++ to HDL compiler released in 2015
  - First complete system builder released in 2018
  - Altera acquired by Intel in 2015, Xilinx acquired by AMD in 2022

Development methods

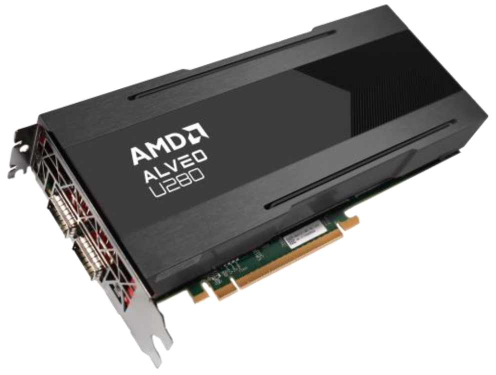
- **Hardware Description Language (HDL)**
  - Natural development flow (VHDL, Verilog, SystemVerilog)
  - Bit- and clock-cycle- level operations
  - Difficult, nonintuitive for software developers
  - Output product of any other design development method

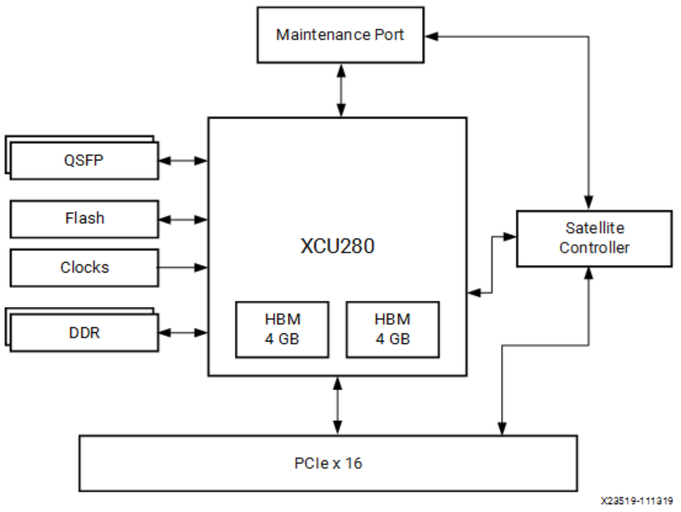
- **High-Level Synthesis**
  - C++ compiler into HDL
  - Enables implementation of complex algorithmics
  - Enables development of acceleration kernels
  - Requires basic understanding of FPGA architecture

- **System builder**
  - Development suite for accelerator cards with PCIe
  - Host – kernel architecture
  - OpenCL or native XRT abstractions
  - Complete flow in C++
  - Compiler automatically generates host – memory – kernels interconnect

FPGA Accelerators

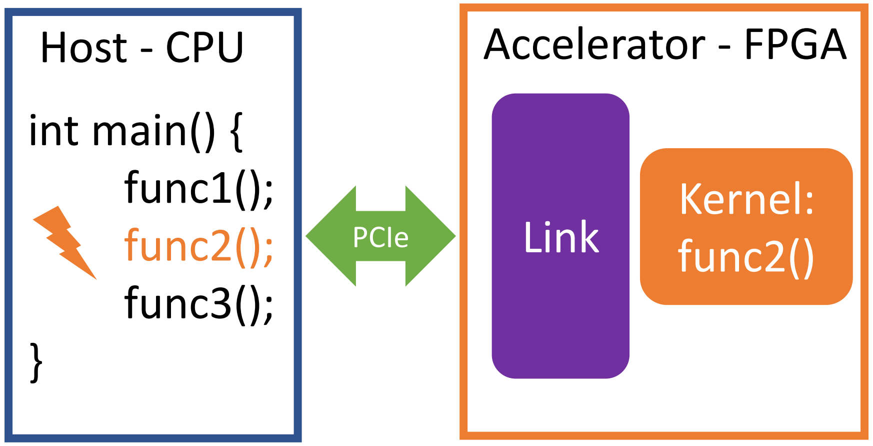
- High-capacity FPGA variants
  - 500k CLB U280 for 8k EUR, equivalent VU35P 40k EUR
- PCIe 4.0x8
- Integrated 8 GB HBM
- External 32 GB DDR4
- 2x QSFP28
- Similar products from various manufacturers





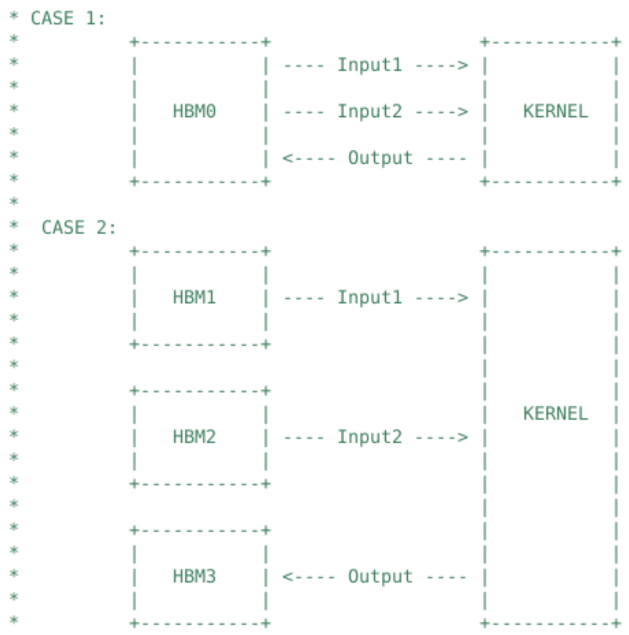
Acceleration kernels

- Delegation of some algorithmic parts to FPGA resources
  - Complete development flow in C/C++
  - FPGA design generated based on configuration files
    - Kernel compiled with High-Level Synthesis
    - Link automatically generated based on configuration
  - FPGA configuration bitfile produced for execution

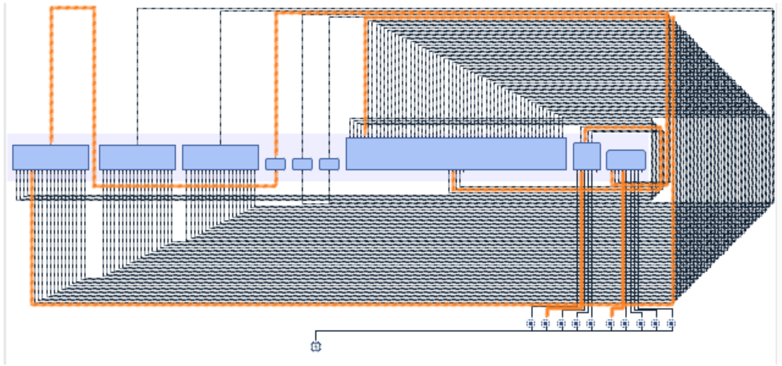


Acceleration kernels

- Optimizations
  - Kernel level (HLS pragmas)
    - Loop unrolling
    - Pipelining computations
    - Memory layout
    - Data types
    - Balance between resource usage and performance
  - System level (System Builder config. file)
    - Types of kernels
    - Instances of kernels
    - Memory layout
    - Kernels interconnections



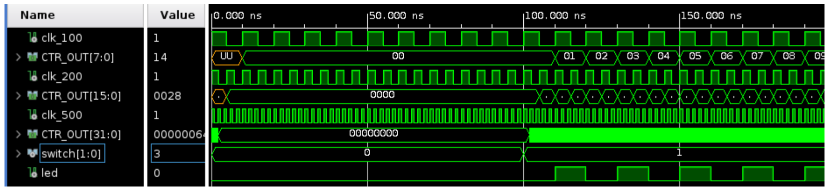
```
* CASE 1:
*            +-----------+                    +-----------+
*            |           | ---- Input1 ---->  |           |
*            |           |                    |           |
*            |   HBM0    | ---- Input2 ---->  |  KERNEL   |
*            |           |                    |           |
*            |           | <---- Output ----  |           |
*            +-----------+                    +-----------+
*
*  CASE 2:
*            +-----------+                    +-----------+
*            |           |                    |           |
*            |   HBM1    | ---- Input1 ---->  |           |
*            |           |                    |           |
*            +-----------+                    |           |
*                                             |           |
*            +-----------+                    |           |
*            |           |                    |  KERNEL   |
*            |   HBM2    | ---- Input2 ---->  |           |
*            |           |                    |           |
*            +-----------+                    |           |
*                                             |           |
*            +-----------+                    |           |
*            |           |                    |           |
*            |   HBM3    | <---- Output ----  |           |
*            |           |                    |           |
*            +-----------+                    +-----------+
```



Acceleration kernels

- Compilation and debugging
  - System builder produces FPGA configuration bitfile
  - Bitfile can be uploaded to the device at any time
  - Configured device can be used by various host executables
    - Unless they use the same kernels and memory layout

- Producing a bitfile is highly time consuming



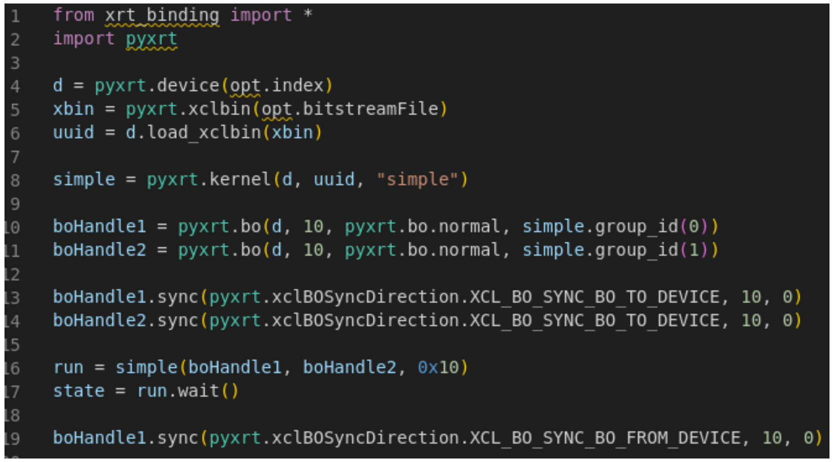
Name	Value
clk_100	1
CTR_OUT[7:0]	14
clk_200	1
CTR_OUT[15:0]	0028
clk_500	1
CTR_OUT[31:0]	0000006
switch[1:0]	3
led	0

- Development flow:
  - Emulation C (fastest)
    - Compilation and execution the entire sourcebase as standard g++ on CPU
    - Consistency check of the algorithm
  - Hardware Emulation (slow)
    - Link with kernels compiled into HDL
    - Execution as simulation of the HDL clock cycle after cock cycle
    - Verification of optimizations, memory layout
  - Hardware (extremly slow)
    - Final bitfile produced
    - Execution on hardware

- Each step produces series of reports to analyze (execution profiles, resource consumption, etc.)

Acceleration kernels

- Execution
  - Host has to configure the device, transfer data, call the kernel and retrieve the results
  - OpenCL abstractions
    - cl::Device, cl::Context, cl::Program
    - cl::Buffer, cl::CommandQueue, cl::Kernel
  - Xilinx Runtime (XRT) abstractions
    - xrt::device, device.load_xclbin
    - xrt::bo, xrt::bo.sync, xrt::kernel
    - pyxrt bidings for Python

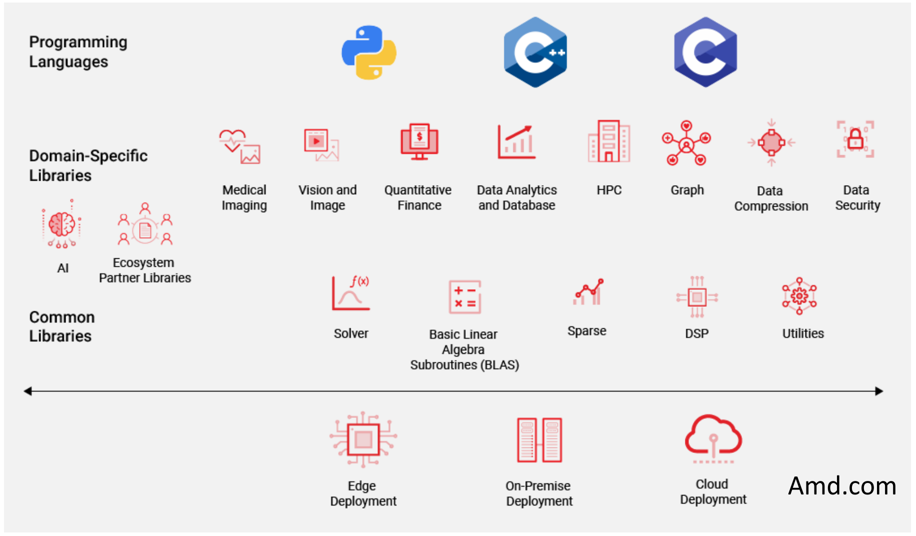


```python
from xrt_binding import *
import pyxrt

d = pyxrt.device(opt.index)
xbin = pyxrt.xclbin(opt.bitstreamFile)
uuid = d.load_xclbin(xbin)

simple = pyxrt.kernel(d, uuid, "simple")

boHandle1 = pyxrt.bo(d, 10, pyxrt.bo.normal, simple.group_id(0))
boHandle2 = pyxrt.bo(d, 10, pyxrt.bo.normal, simple.group_id(1))

boHandle1.sync(pyxrt.xclBOSyncDirection.XCL_BO_SYNC_BO_TO_DEVICE, 10, 0)
boHandle2.sync(pyxrt.xclBOSyncDirection.XCL_BO_SYNC_BO_TO_DEVICE, 10, 0)

run = simple(boHandle1, boHandle2, 0x10)
state = run.wait()

boHandle1.sync(pyxrt.xclBOSyncDirection.XCL_BO_SYNC_BO_FROM_DEVICE, 10, 0)
```

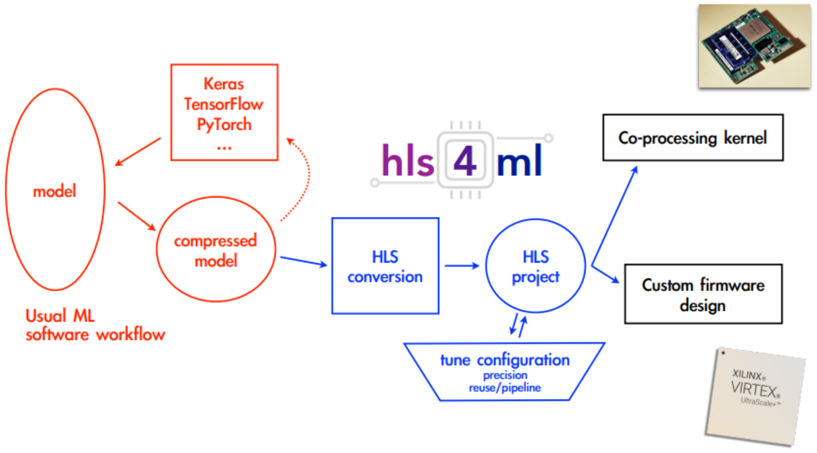
https://github.com/Xilinx/XRT/blob/master/tests/python/02_simple/main.py

Acceleration kernels

- How to construct a kernel?
  - Implement plain C++
  - Search accelerated libraries catalog for subfunctions
    - Configurable and optimised implementations of typical functions
  - Github examples of all FPGA features and optimizations
  - External tools e.g.:
    - HLS-4-ML
      - Keras, PyTorch to C++ and HLS project
      - Direct model inference conversion from Python to FPGA kernel



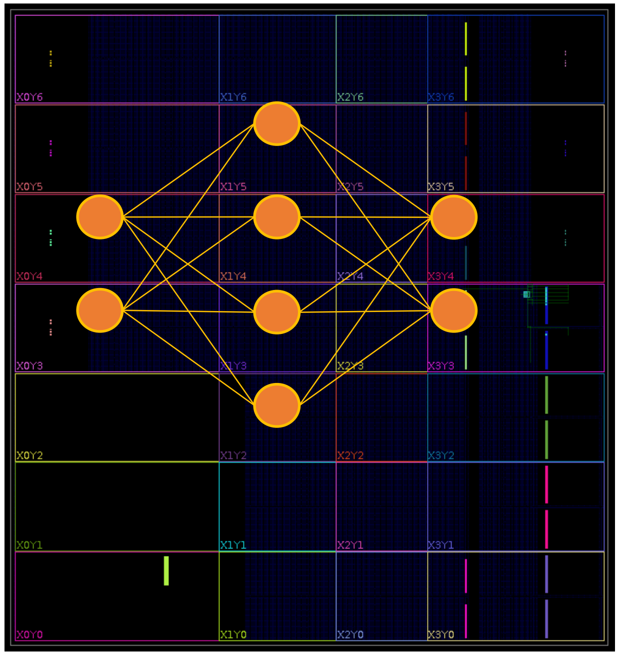
Amd.com



Python C++ Bitstream

AI on FPGA accelerators

- Inference only
  - Inference is feed-forward, simple arithmetics, layer-by-layer pipelined, natural for FPGA
- Optimizations on data types
  - Custom data types are natural for FPGA
  - (int<8>, int<3>, ap_fixed(14, 5), …)
- Control over each clock cycle
  - Low and deterministic latency in true real-time



- Prominent example
  - CMS experiment trigger system
  - Sustained collision rate 40 MHz
  - Fast decision on data quality
  - Implemented with HLS-4-ML

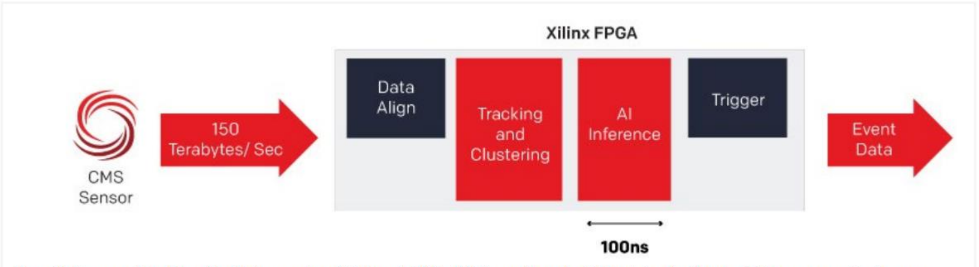
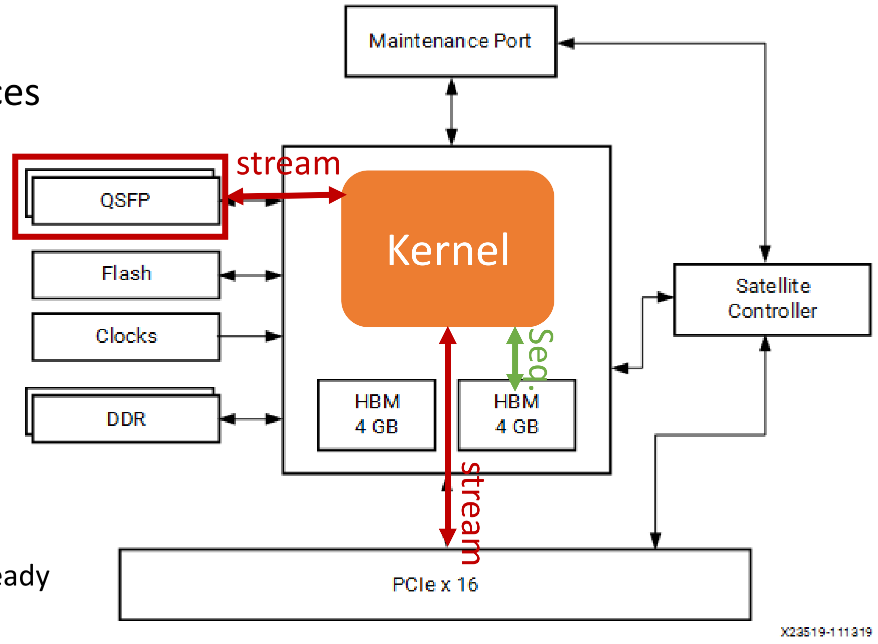


Figure 2: Compared to alternative devices such as GPUs and ASICs, FPGAs are the only viable choice for the event trigger processing because they provide extremely low latency. While the large numerical processing capability of GPUs is attractive, these technologies are optimized for high throughput, not low latency.

https://www.xilinx.com/publications/powered-by-xilinx/cerncasestudy-final.pdf

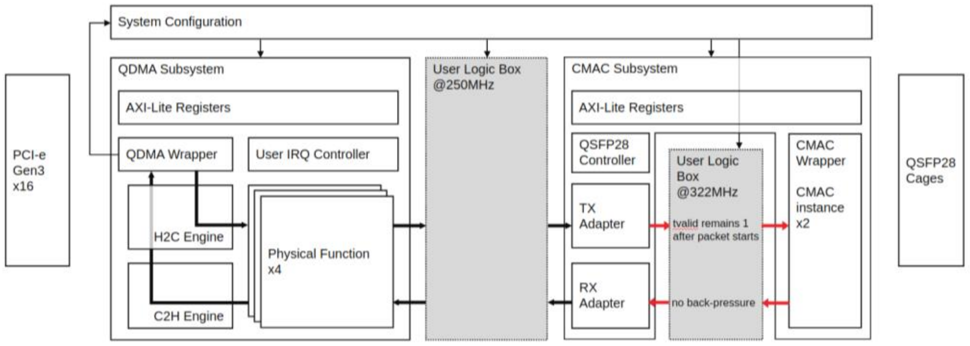
Data sources

- Sources of data for kernels:
  - Buffers allocated and migrated to local resources
    - Migrate to, compute, migrate from scheme
  - Buffer streamed to the kernel through PCIe
    - Dataflow and streamlined computations
  - 100G Ethernet QSFP28 sockets
    - Dataflow and streamlined computations
    - Lowest and deterministic network latency
      - 150ns from electric signals to decoded bytes ready to process



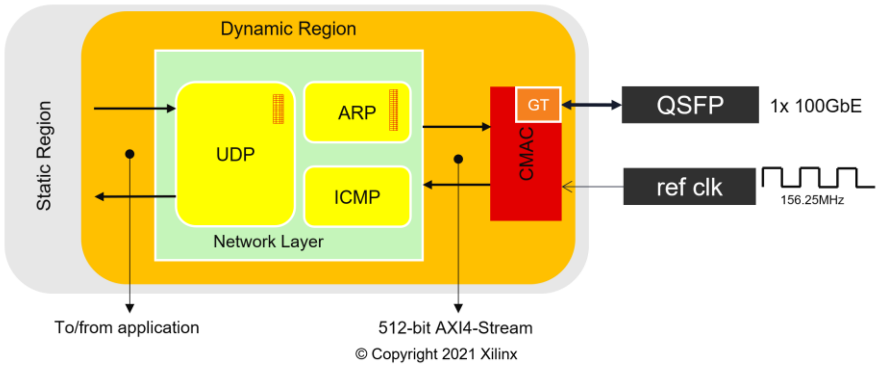
Ethernet solutions

- OpenNIC
  - Acts as a regular network card
  - Protocol stack on CPU and OS
  - Two regions for low-level processing
  - Network security, encryption, encoding…



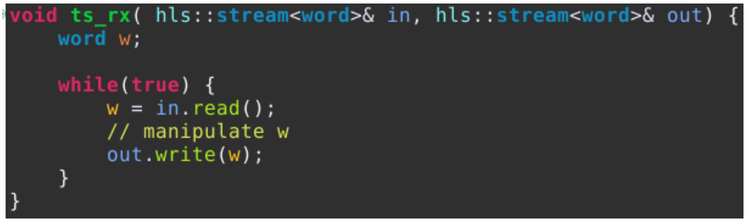
https://github.com/Xilinx/open-nic?tab=readme-ov-file

- VNx
  - ARP, ICMP, UDP implemented in the FPGA
  - UDP transmitter and receiver as a kernel
  - No driver, kernel access from the host via OpenCL or XRT
  - Custom, ultra-low latency applications, …



https://github.com/Xilinx/xup_vitis_network_example

- Packet processing kernels as HLS
  - Single loop iteration – single clock cycle



```
void ts_rx( hls::stream<word>& in, hls::stream<word>& out) {
    word w;

    while(true) {
        w = in.read();
        // manipulate w
        out.write(w);
    }
}
```

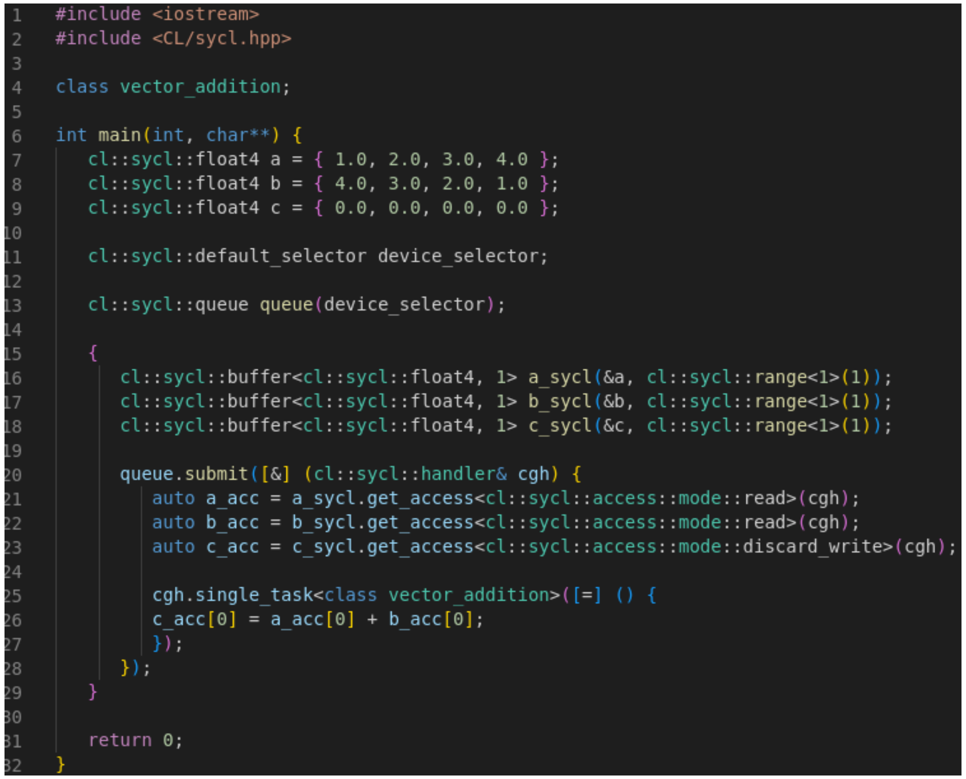
Higher-level abstractions

- Programming models for heterogeneous systems
  - Single codebase compiled to any platform
  - Any C++ callable as kernel
  - Reused OpenCL concepts
  - Implicit host-kernel link

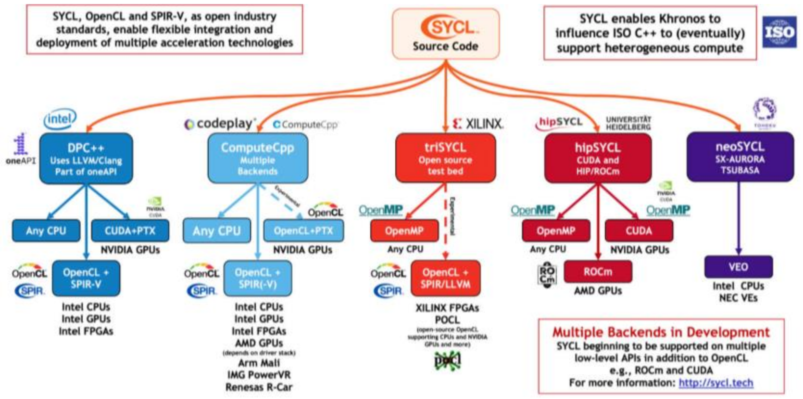
  - Deep optimization requires tech. specific constructs

  - Target platform selected in a Makefile
  - Compilers chain run underneath

  - Interesting solution for data analysis in physics



```cpp
#include <iostream>
#include <CL/sycl.hpp>

class vector_addition;

int main(int, char**) {
    cl::sycl::float4 a = { 1.0, 2.0, 3.0, 4.0 };
    cl::sycl::float4 b = { 4.0, 3.0, 2.0, 1.0 };
    cl::sycl::float4 c = { 0.0, 0.0, 0.0, 0.0 };

    cl::sycl::default_selector device_selector;

    cl::sycl::queue queue(device_selector);

    {
        cl::sycl::buffer<cl::sycl::float4, 1> a_sycl(&a, cl::sycl::range<1>(1));
        cl::sycl::buffer<cl::sycl::float4, 1> b_sycl(&b, cl::sycl::range<1>(1));
        cl::sycl::buffer<cl::sycl::float4, 1> c_sycl(&c, cl::sycl::range<1>(1));

        queue.submit([&] (cl::sycl::handler& cgh) {
            auto a_acc = a_sycl.get_access<cl::sycl::access::mode::read>(cgh);
            auto b_acc = b_sycl.get_access<cl::sycl::access::mode::read>(cgh);
            auto c_acc = c_sycl.get_access<cl::sycl::access::mode::discard_write>(cgh);

            cgh.single_task<class vector_addition>([=] () {
            c_acc[0] = a_acc[0] + b_acc[0];
            });
        });
    }

    return 0;
}
```


SYCL Source Code
intel
oneAPI
DPC++ Uses LLVM/Clang Part of oneAPI
Any CPU
CUDA+PTX
NVIDIA GPUs
OpenCL + SPIR-V
Intel CPUs Intel GPUs Intel FPGAs
codeplay
ComputeCpp
ComputeCpp Multiple Backends
Experimental
Any CPU
OpenCL+PTX
NVIDIA GPUs
OpenCL + SPIR(-V)
Intel CPUs Intel GPUs Intel FPGAs AMD GPUs (depends on driver stack) Arm Mali IMG PowerVR Renesas R-Car
XILINX
triSYCL Open source test bed
OpenMP
Any CPU
Experimental
OpenCL + SPIR/LLVM
XILINX FPGAs POCL (open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
pocl
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL CUDA and HIP/ROCm
OpenMP
Any CPU
CUDA
NVIDIA GPUs
ROCm
AMD GPUs
neoSYCL SX-AURORA TSUBASA
VEO
Intel CPUs NEC VEs
ISO

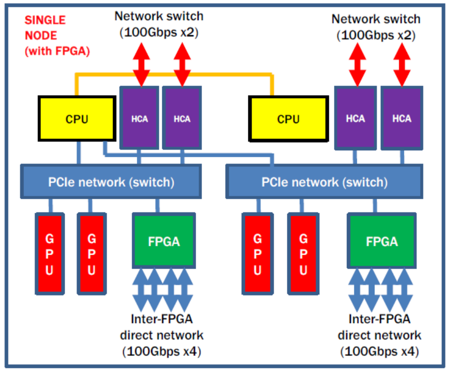
SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

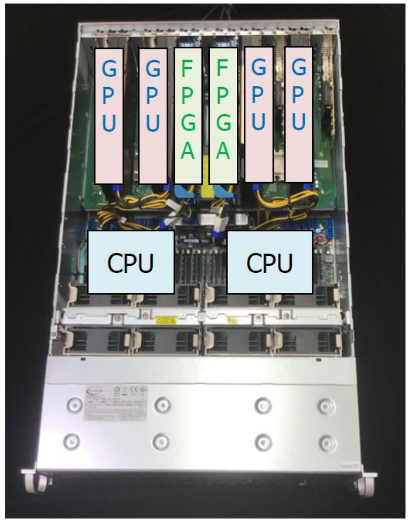
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute

Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL e.g., ROCm and CUDA
For more information: http://sycl.tech

FPGAs in HPC

- FPGAs have a unique set of features:
  - Naturally parallel and pipelined processing
  - Ultra-low latency, true real-time processing
  - Adaptable resources, dynamic reconfiguration
- Devices have sufficient amount of resources
  - Capable of accelerating complex algorithms
  - Problematic for the compilers
- Difficulty in transition from sequential programming to HDL
  - High-level abstractions, compilers and tools introduced
- Software maturity required for another technology breaktrough
- Hardware Acceleration Cluster at UJ
  - 4x nodes
  - 7x AMD Alveo acceleration cards (U280, U50)
  - 4x Nvidia GPUs (RTX 4090, 2080)
  - 100G Ethernet network


SINGLE NODE (with FPGA)
Network switch (100Gbps x2)
Network switch (100Gbps x2)
CPU
HCA
HCA
CPU
HCA
HCA
PCIe network (switch)
PCIe network (switch)
GPU
GPU
FPGA
GPU
GPU
FPGA
Inter-FPGA direct network (100Gbps x4)
Inter-FPGA direct network (100Gbps x4)



T. Boku, „Japanese Supercomputer development and hybrid accelerated supercomputing"