

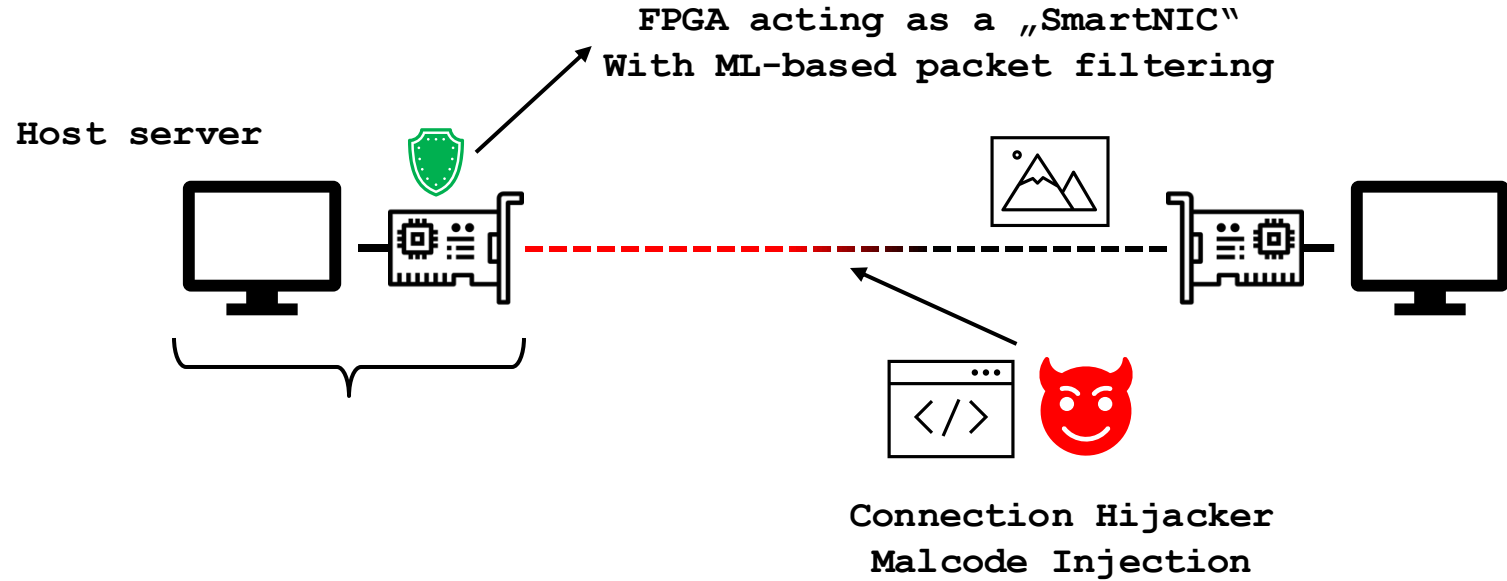
Coyote v2: Open-source Abstractions and Infrastructure for FPGAs

Benjamin Ramhorst, Maximilian Heer, Gustavo Alonso

- ❑ The aim of this tutorial is to familiarize you with **Coyote v2**, an open-source FPGA shell, which facilitates the deployment of complex applications on real FPGAs as well as the integration of FPGAs in more complex systems, with GPUs, SmartNICs, etc.

- ❑ Some of you are already familiar with **hls4ml**; so, first, let's see how **Coyote** can be used to seamlessly deploy an hls4ml-accelerated model for in-network traffic filtering on an FPGA-based SmartNIC.

Getting-started demo

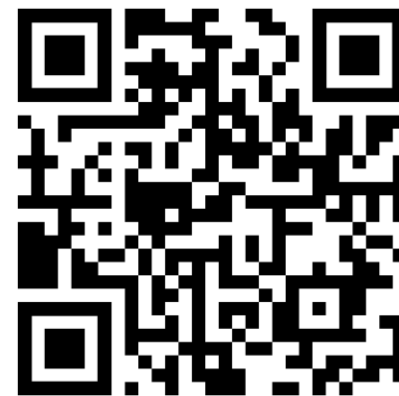


```
Eingabeaufforderung - ssh lu x + v - □ x
(base) hmaximili@alveo-u55c-07:/mnt/scratch/hmaximili/RDMA_tests/mellanox2Coyote_rdma$
```

```
Windows PowerShell x + v - □ x
(base) hmaximili@alveo-u55c-08:/mnt/scratch/hmaximili/RDMA_tests/coyote_dev_16_intrusion_detection_net
work/sw/build$
```

Tutorial Info

- ❑ Coyote repository: <https://github.com/fpgasystems/Coyote>
- ❑ Video recordings and slides from previous tutorial (ASPLOS 2025):
<https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc/asplos25-tutorial-fpgas.html>
- ❑ Coyote documentation: <https://fpgasystems.github.io/Coyote/>
- ❑ All of the above can be found from the following QR code:



Introduction & Motivation

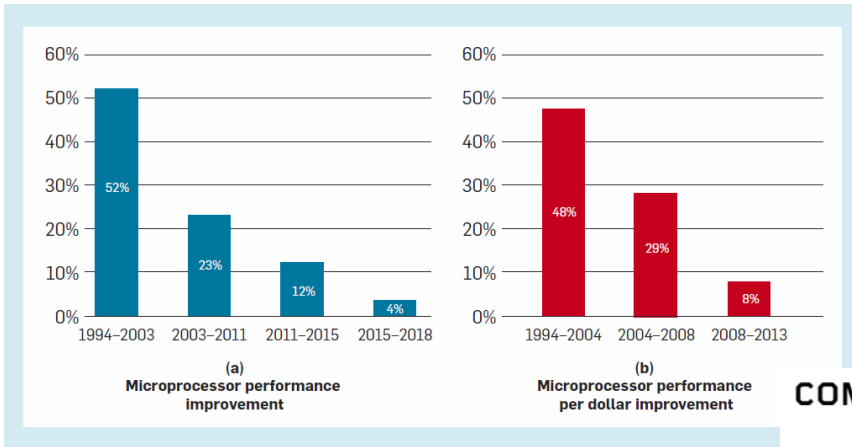
What makes a good FPGA shell?

Coyote v2: An OS for FPGAs

Examples & Demo

General-purpose computing

- Decreasing performance improvements in general-purpose processors (e.g. CPU) → **hardware specialization**



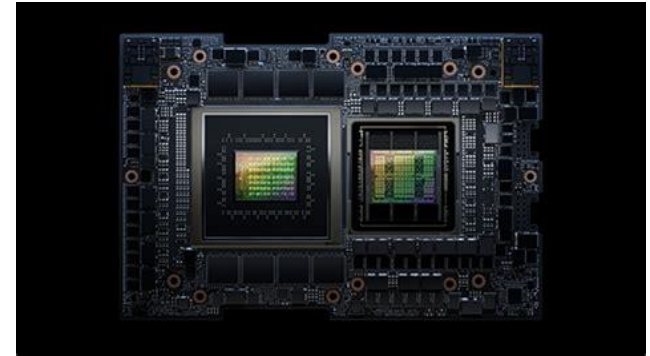
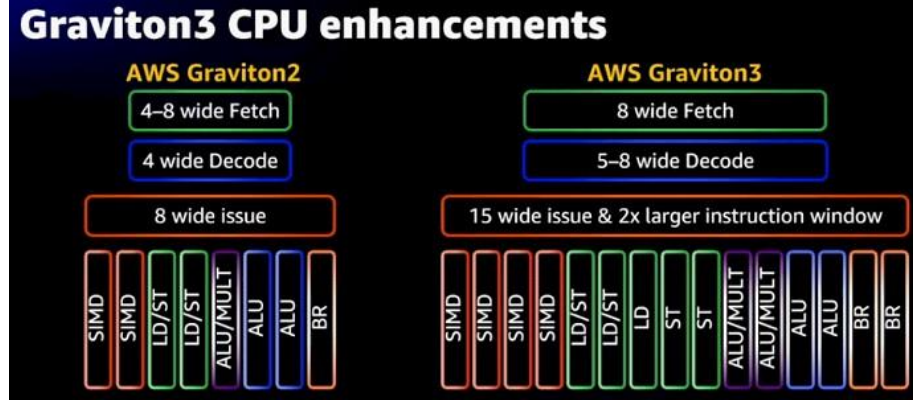
Driving specialization & heterogeneity

- ❑ Cloud computing and big data & AI/ML applications have led to big shifts in computing:
 - ❑ Very large workloads
 - ❑ “On-demand” computing

- ❑ Increasing deployments of specialized hardware:
 - ❑ GPUs
 - ❑ Google TPUs
 - ❑ DPUs & SmartNICs (e.g., BlueField)
 - ❑ AWS Inferentia, Microsoft Maia



Driving specialization & heterogeneity



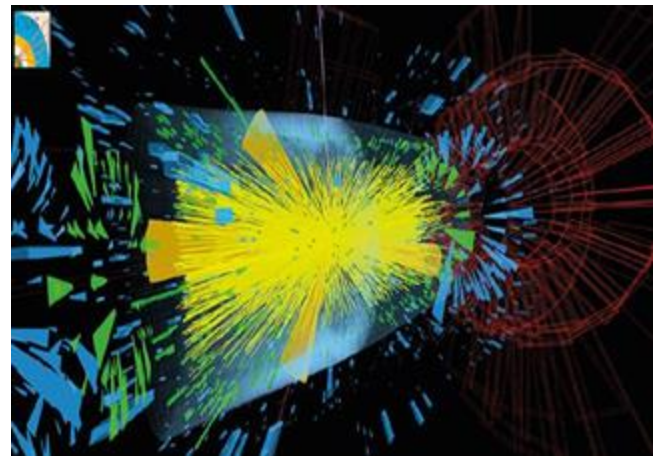
Everything that is demanding enough and common enough will move to dedicated accelerators

This is true for all cloud infrastructure but also for several use cases and applications

How do FPGAs fit in the picture?

- ❑ FPGAs offer some advantage over other accelerators:
 - ❑ High reconfigurability → quick prototyping
 - ❑ Custom logic, precision and memory management
 - ❑ Stream-like processing for networking and dataflow tasks

- ❑ For example, at CERN, hundreds of inter-connected FPGAs are deployed for low-latency particle class.:
 - ❑ Sub-microsecond inference latency
 - ❑ Rich eco-system with tools like hls4ml and conifer



FPGAs in the cloud & large-scale systems

FPGA
Use Cases

Production
&
Applications

Prototyping
&
Research

Amazon AQUA
Microsoft Catapult
Microsoft Azure Boost
Alibaba Fidas
Baidu Kunlun

Database acceleration
Machine learning
acceleration
Cache coherency
Networking

Developments Regarding the Integration of FPGA RDMA into the ATLAS Readout with FELIX in High Luminosity LHC

Matei-Eugen Vasile, Sorin Martoiu, Nayib Boukadida, Gabriel Stoicea, Petru Micu, Alexandru Dumitru, Andrei-Alexandru Ulmamei, Radu Hobincu, Cristina-Cerasela Iordache

Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research

David Cock

Abishek Ramdas

Daniel Schwyn

Non-Relational Databases on FPGAs: Survey, Design Decisions, Challenges

Jonas Dann · Daniel Ritter · Holger Frning

FPGAs in the cloud & large-scale systems

❑ FPGAs offer

❑ ... AND hav

❑ ... BUT prac

❑ Platform-s

❑ Network

❑ Memory (

And in general, a large issue of the FPGA development process is the incompatibility of tooling infrastructure to be migrated to different projects and devices.

involved:
(on PCIe)
of it)

Introduction & Motivation

What makes a good FPGA shell?

Coyote v2: An OS for FPGAs

Examples & Demo

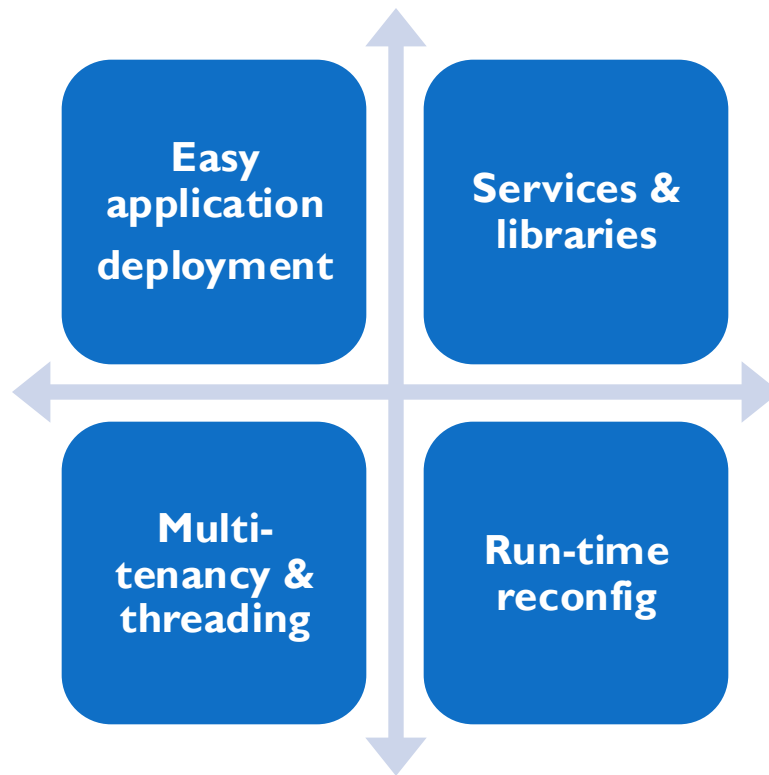
What are FPGA shells?

- ❑ *Shell* is the run-time environment for FPGAs
 - ❑ Similar to an OS on a CPU

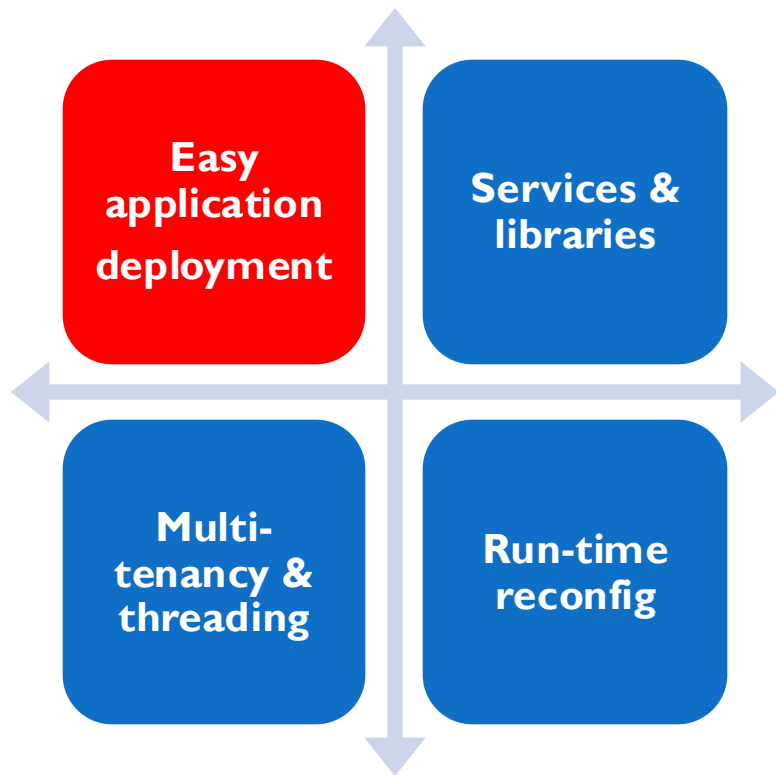
- ❑ For e.g., the default shell for AMD FPGAs is Xilinx Run Time (XRT), also known as Vitis shell

- ❑ A shell is responsible for multiple actions on the FPGA, e.g.,
 - ❑ Moving data between the host memory and the FPGA application
 - ❑ Launching FPGA applications; setting and reading control registers
 - ❑ Sending/receiving data over the network
 - ❑ Triggering run-time reconfiguration

What makes a good FPGA shell?



So, what makes a good shell?



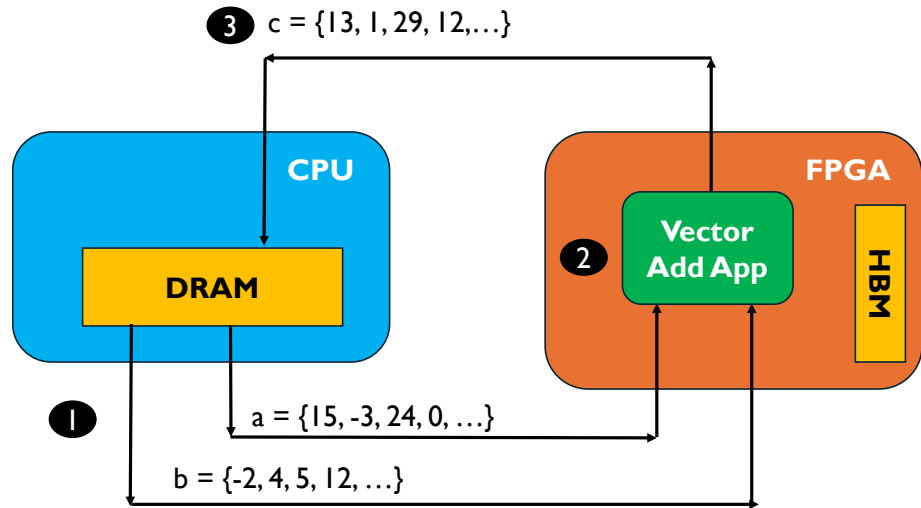
- ❑ A good shell should facilitate easy deployment of user apps on the FPGA and automatically handle data movement between various parts of the system (CPU memory, FPGA HBM)
- ❑ Let's consider two trivial examples

Vector addition example

Ideally, to do vector addition we would have two input data streams and one output streams

In Coyote v1, there is only one input/output stream:

- The developer must pack both vectors into one stream in SW
- And then unpack them in HW

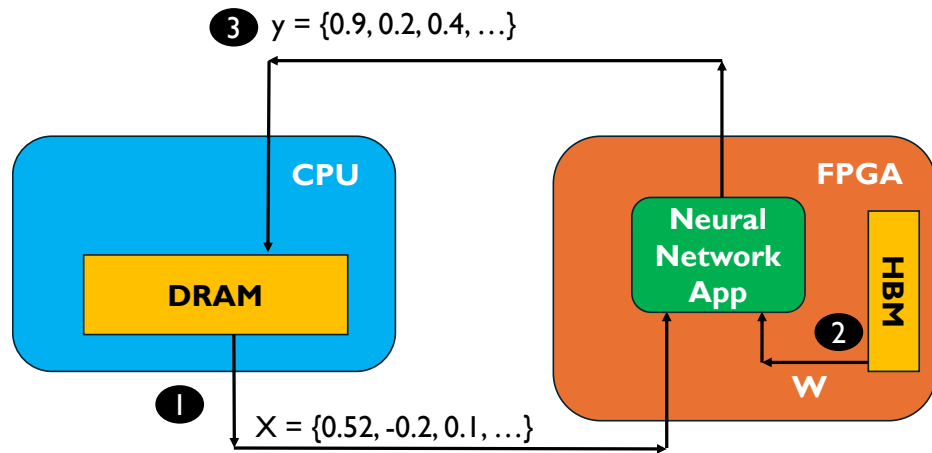


Neural network inference

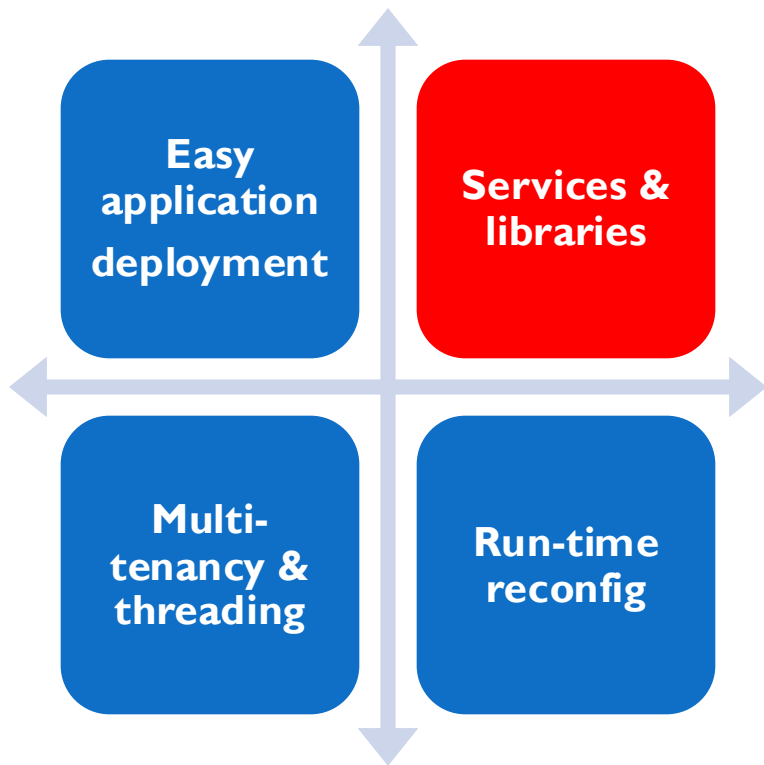
Like on a GPU, we would like weights to come from FPGA's HBM and data from the host memory.

In AmorphOS, there is only support for FPGA's HBM:

→ No data from the host



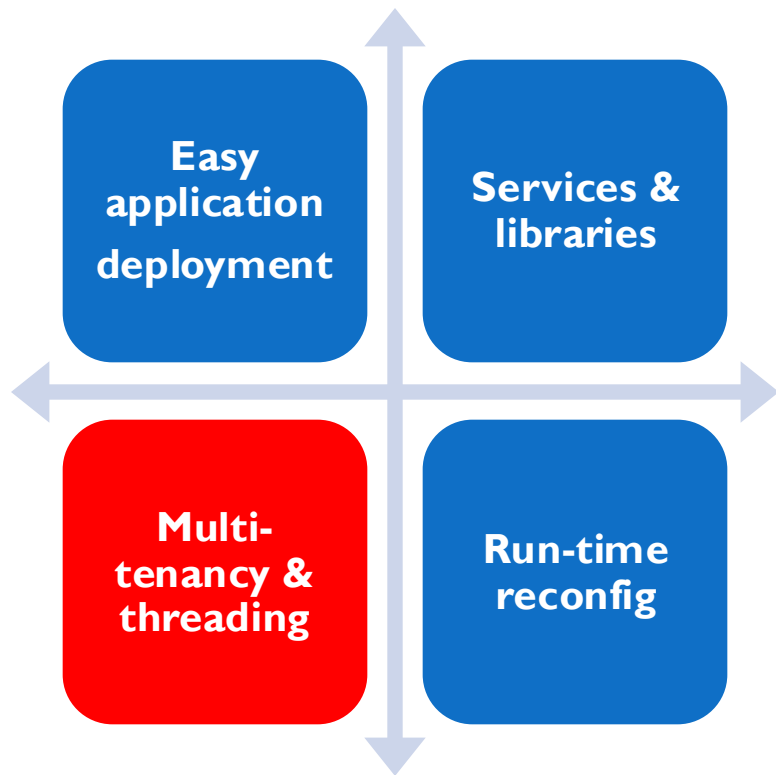
So, what makes a good shell?



- ❑ We shouldn't have to re-invent the wheel (or re-build the infrastructure) for every project we do:
→ **Yet we still do!**

- ❑ For example, in C++ we would never write our own **malloc** function.
- ❑ Similarly, if we want to do networking, we would simply use OpenMPI

So, what makes a good shell?



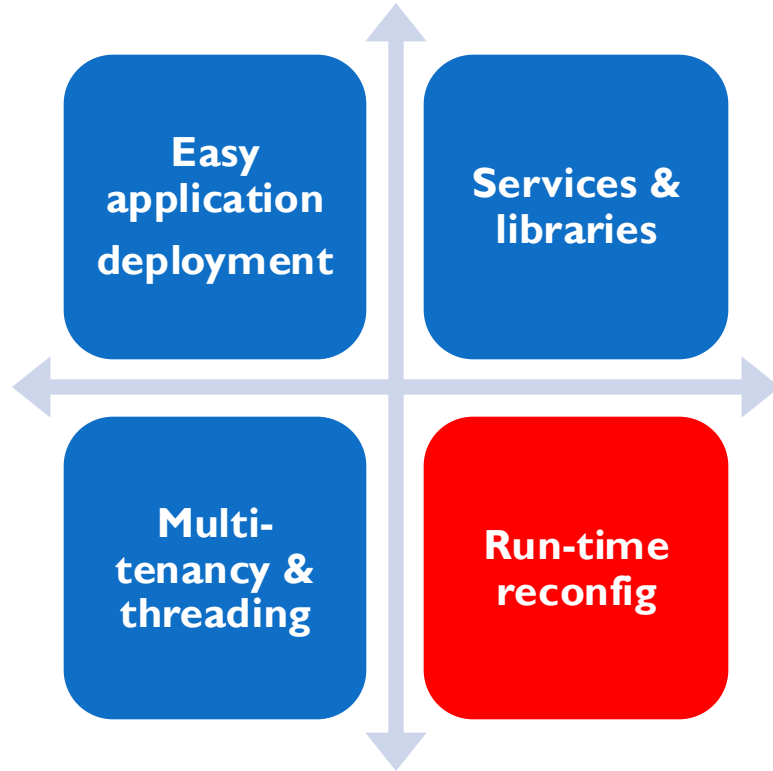
❑ Hardware multi-tenancy:

- ❑ FPGAs are very powerful
- ❑ Applications rarely utilize the full chip
- ❑ Enable multiple “users” to deploy their applications

❑ Software “multi-threading”:

- ❑ Often, the same application can be useful for multiple clients (e.g., LLMs)
- ❑ Often impractical to deploy the application multiple times
- ❑ Therefore, one hardware core – many software threads

So, what makes a good shell?

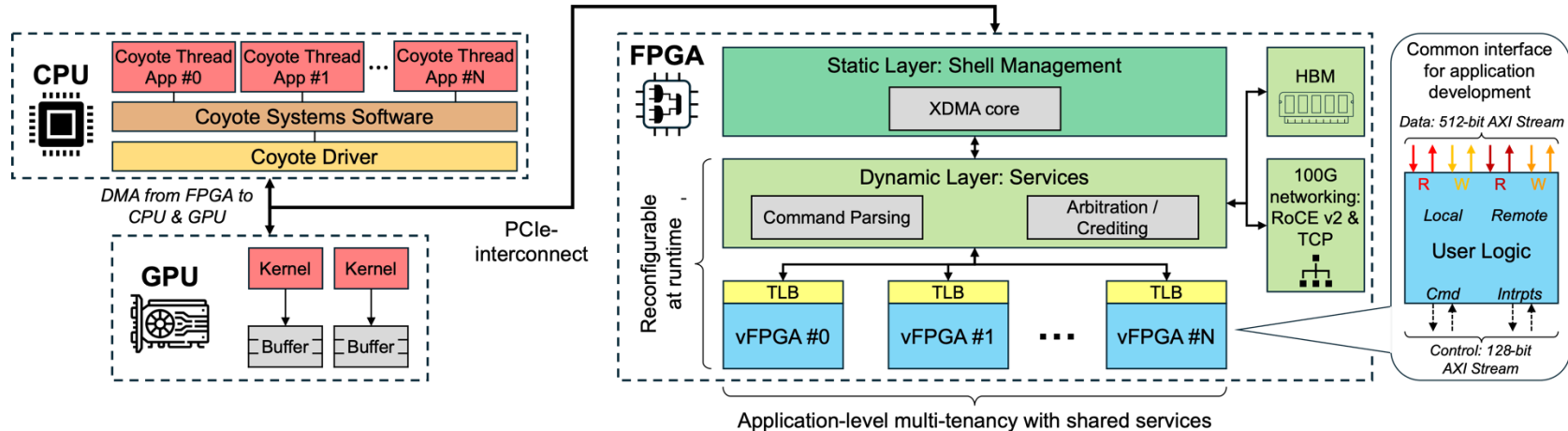


- ❑ **The cloud is not static!**
- ❑ Applications, demands and requirements change over time
- ❑ It should be possible to quickly reconfigure the user application on request

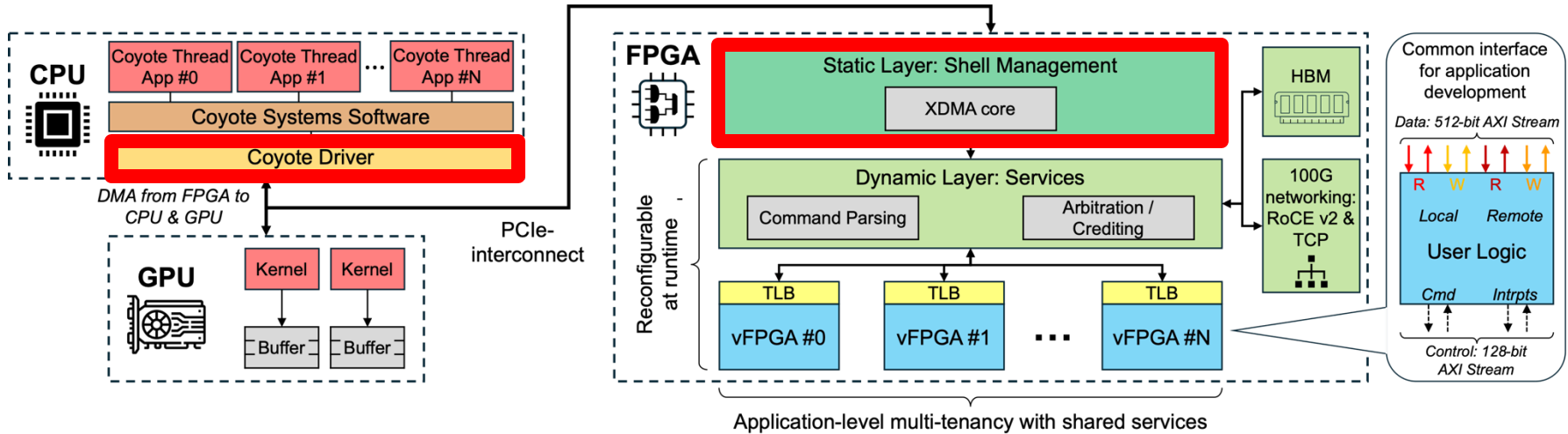
- Introduction & Motivation
- What makes a good FPGA shell?
- **Coyote v2: An OS for FPGAs**
- Examples & Demo

System overview

- Coyote consists of:
 - A static layer
 - A dynamic (services) layer
 - An application (user) layer



Static layer



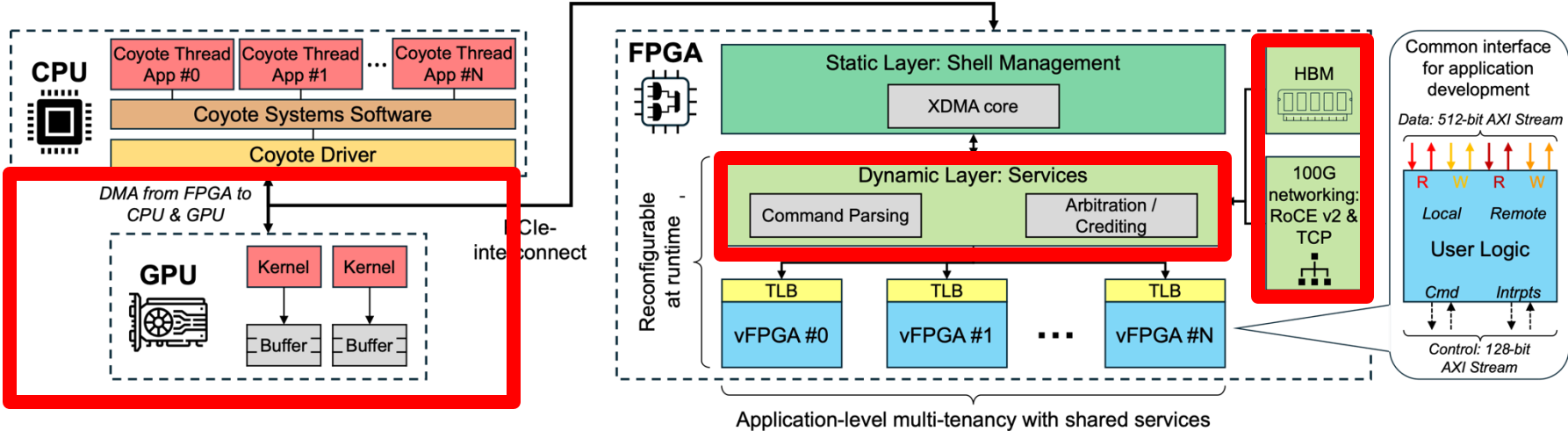
Static layer

- ❑ Primary purpose is to facilitate interaction with the host:
 - ❑ DMA Engine
 - ❑ Interrupt channel
 - ❑ Memory mapping, for e.g. reconfiguration memory

- ❑ The drivers acts like a "middle-man" between the hardware and the software

- ❑ Always "online" – cannot be reconfigured; but is used for re-configuring the rest of the shell

Dynamic (services) layer



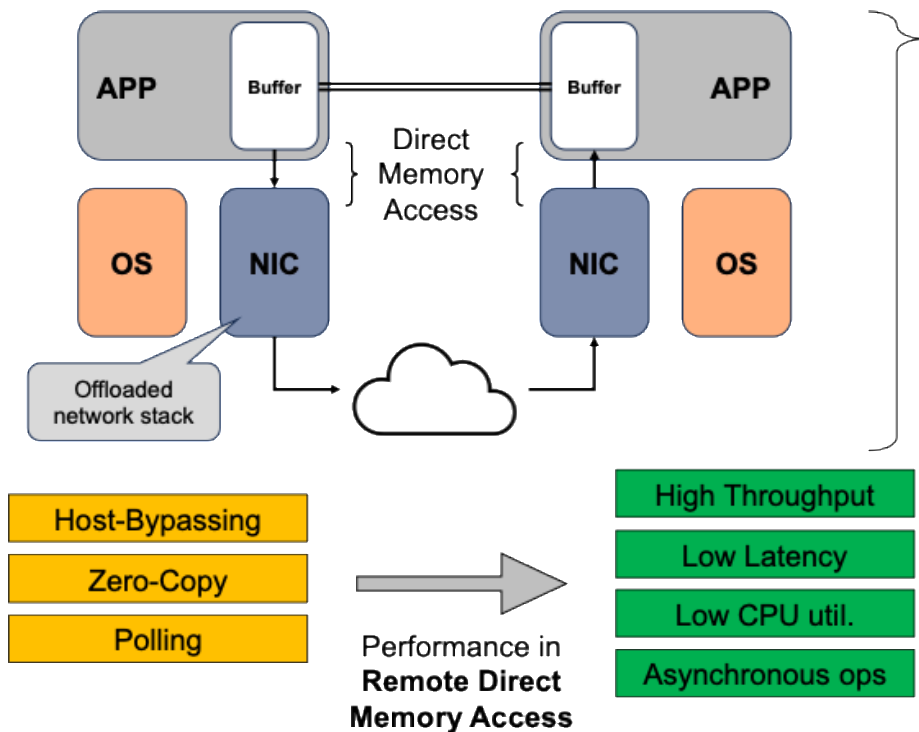
Dynamic (services) layer

- ❑ All of memory in Coyote is virtualized and striped:
 - ❑ From the user point of view, only one pointer is presented and its physical memory location (CPU, FPGA, GPU) is fully managed by the shell

- ❑ Additionally, Coyote includes a rich set of services:
 - ❑ RoCE-v2 compatible RDMA stack
 - ❑ Traffic sniffer, similar to Mellanox PCAPing
 - ❑ In progress: TCP/IP and collectives (ACCL)

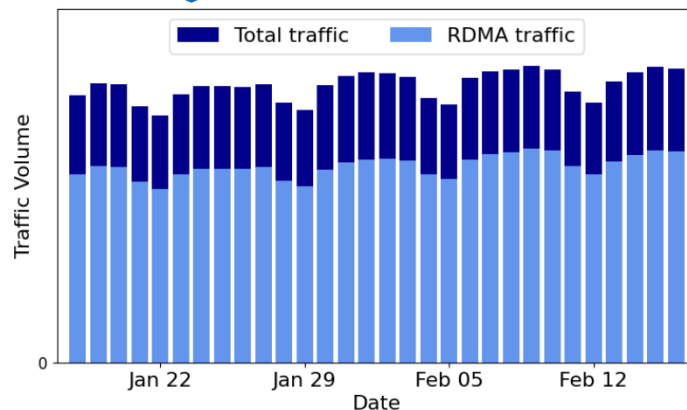
- ❑ Each service can be reconfigured at run-time

A closer look at networking in datacenters: RDMA



RDMA is the de-facto standard for networking in Azure data centers

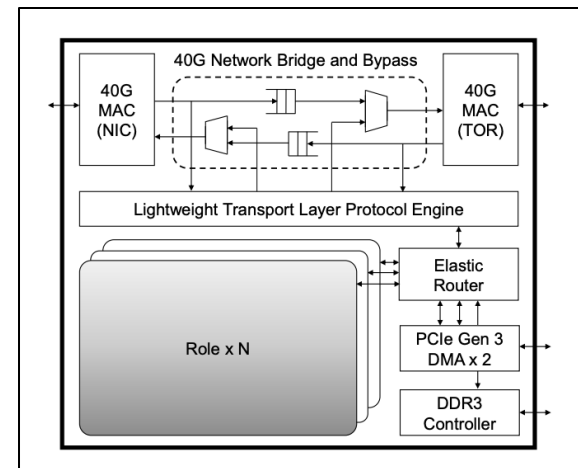
(Empowering Azure Storage with RDMA, W. Bai et al., USENIX NSDI 2023)



FPGAs are very good at networking

- Due to their stream-like processing capabilities and custom logic, FPGAs have been deployed in a wide range of networking roles
- An early example of such production deployments is Microsoft Catapult
- Many papers and research projects on deploying FPGAs in a network-oriented role

A Cloud-Scale Acceleration Architecture. (2016) A. Putnam *et al.* MICRO 2016



FPGAs are very good at networking

Algean: An Open Framework for Deploying Machine Learning on Heterogeneous Clusters

NAIF TARAFDAR, University of Toronto, Canada

ACCL+: an FPGA-Based Collective Engine for Distributed Applications

Zhenhao He
Systems Group, ETH Zurich

Dario Korolija
Systems Group, ETH Zurich

Yu Zhu
Systems Group, ETH Zurich

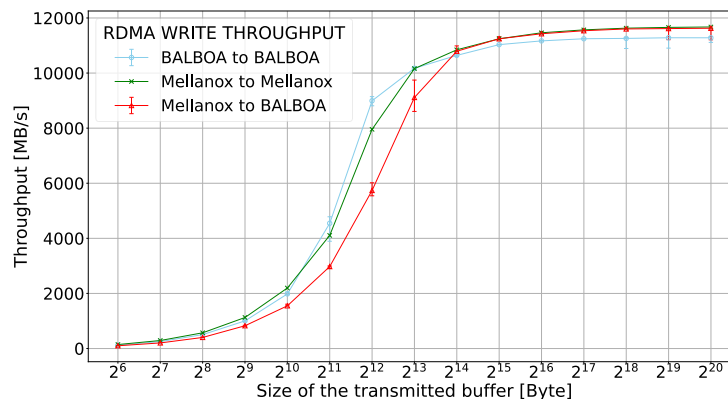
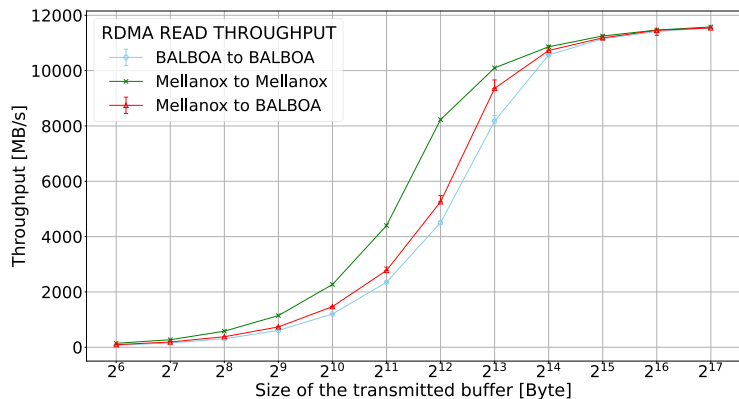
Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack

Mario Ruiz*, David Sidler† Gustavo Sutter*, Gustavo Alonso† and Sergio López-Buedo*‡

Coyote v2's RDMA stack: RoCE BALBOA

- Coyote v2 implements a fully open-source, RoCE v2-compatible RDMA stack for FPGAs:
 - 100G end-to-end throughput
 - Runs over a multi-tier switched network
 - Compatible with commodity NICs (e.g., Mellanox)
 - Support retransmissions for lossy networks

It speaks RoCe, so it must be...



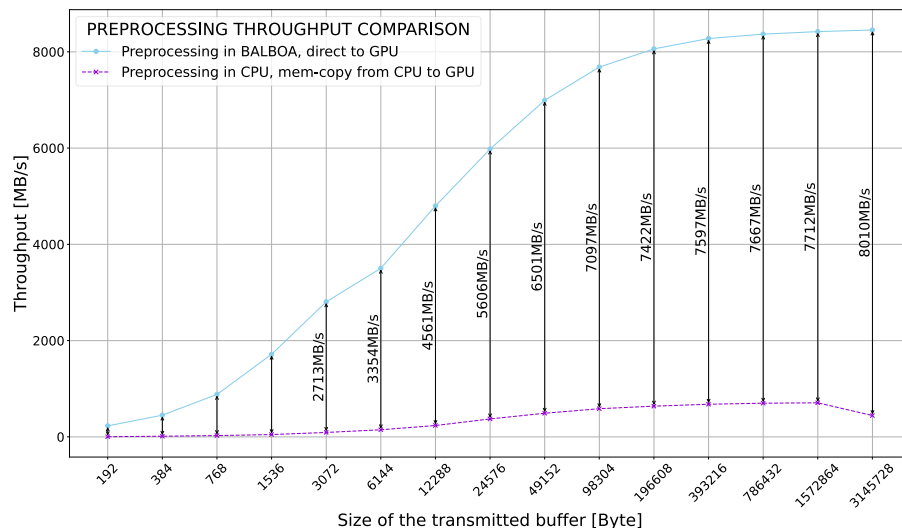
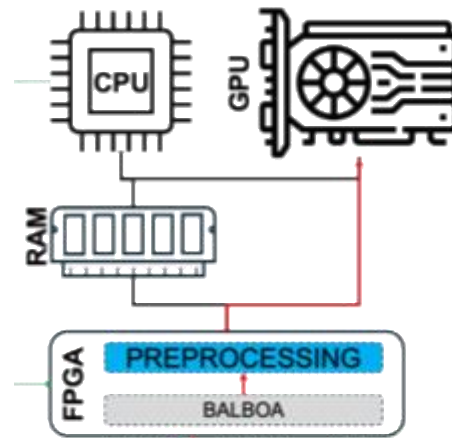
Example use case: In-network ML pre-processing

□ Recommender models are widely used in the cloud (e.g., series & music recommendations, product recommendations, advertising etc.)

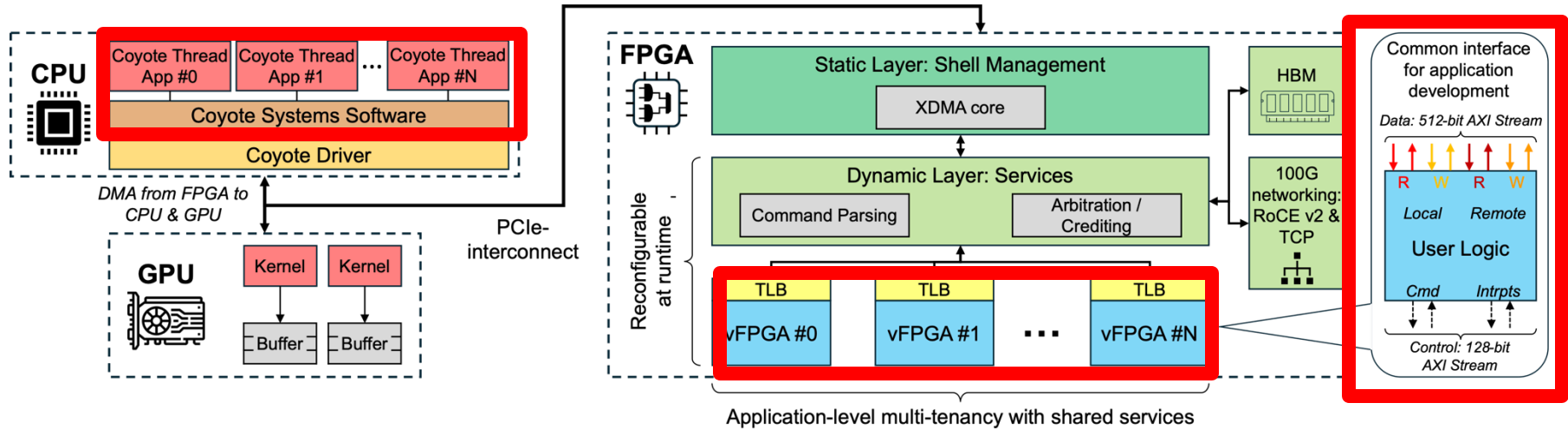
□ Main bottleneck is often pre-processing (up to 60% of total training time):

□ Often off-loaded to dedicated hardware, e.g., SmartNICs, DPUs

□ In our example, we implement the pre-processing on an FPGA and process data that comes from a remote node before sending it to the GPU for the training



Application (user) layer



Application (user) layer

- ❑ Provides a set of standard interfaces, both in hardware and software, for users to easily deploy their applications

- ❑ Each application is called a vFPGA and is reconfigurable at run-time

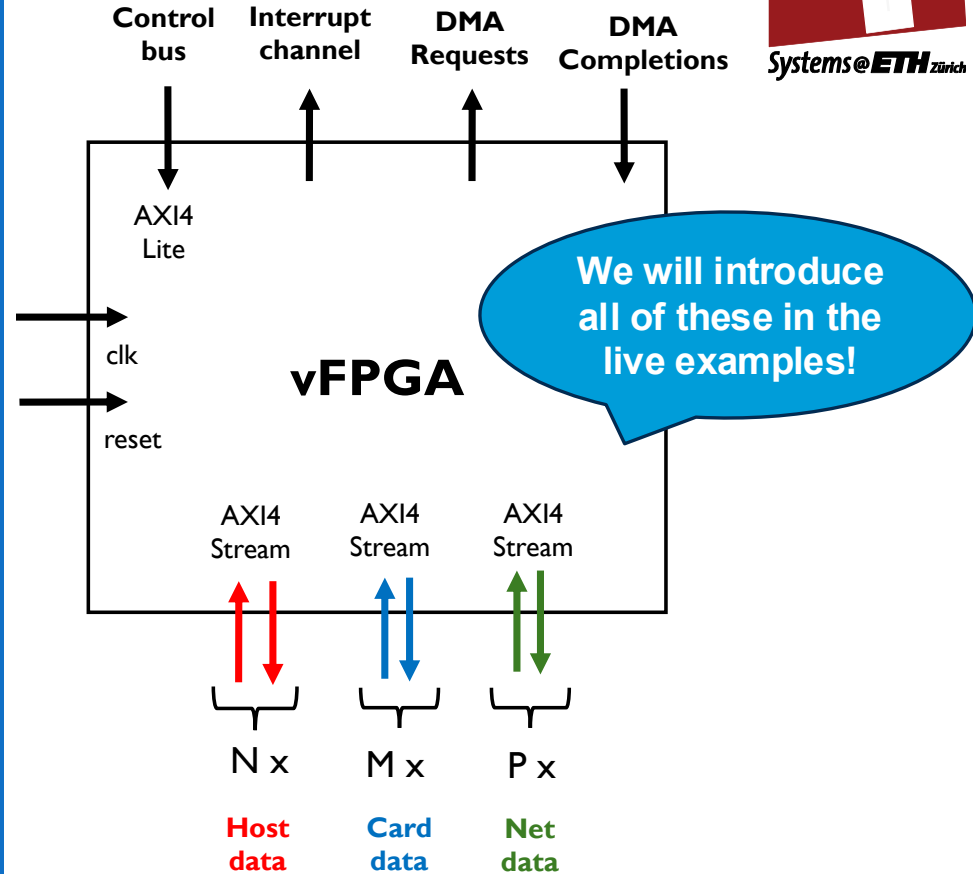
- ❑ Coyote facilitates multi-tenancy:
 - ❑ Simply set the parameter $N_REGIONS$

App layer in hardware

Generic user interfaces, built using industry-standard AXI Streams and Memory-Mapped interfaces.

Multiple data streams, from both host memory, card memory and the network configurable at compile-time

Dedicated interfaces for control signals and user interrupts



- ❑ Provides a set of high-level C++ functions to interact with the FPGA

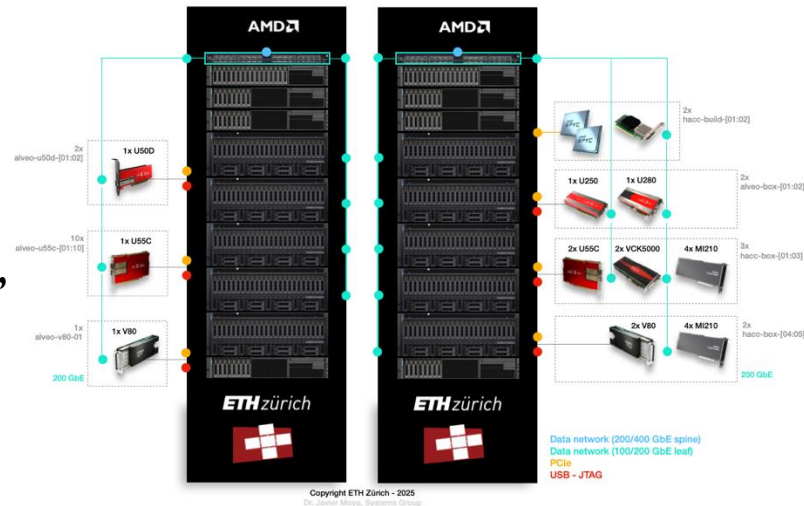
- ❑ Core building component: cThread, which can:
 - ❑ Allocate memory for a specific vFPGA
 - ❑ Set specific control registers in hardware
 - ❑ Poll on interrupts from hardware
 - ❑ Perform data movement and kernel launching

Outline

- Introduction & Motivation
- What makes a good FPGA shell?
- Coyote v2: An OS for FPGAs
- **Examples & Demo**

HACC Overview

- Today we will be using the AMD Heterogeneous Accelerated Compute Clusters (HACC)
- Premiere cluster for systems, computer architecture, networking etc. research:
 - Alveo V80, U55C, U250, U280
 - Instinct GPU
 - VCK5000
- Simplified interaction with the cluster, FPGA programming, driver insertion, system validation etc. through **hdev** utility

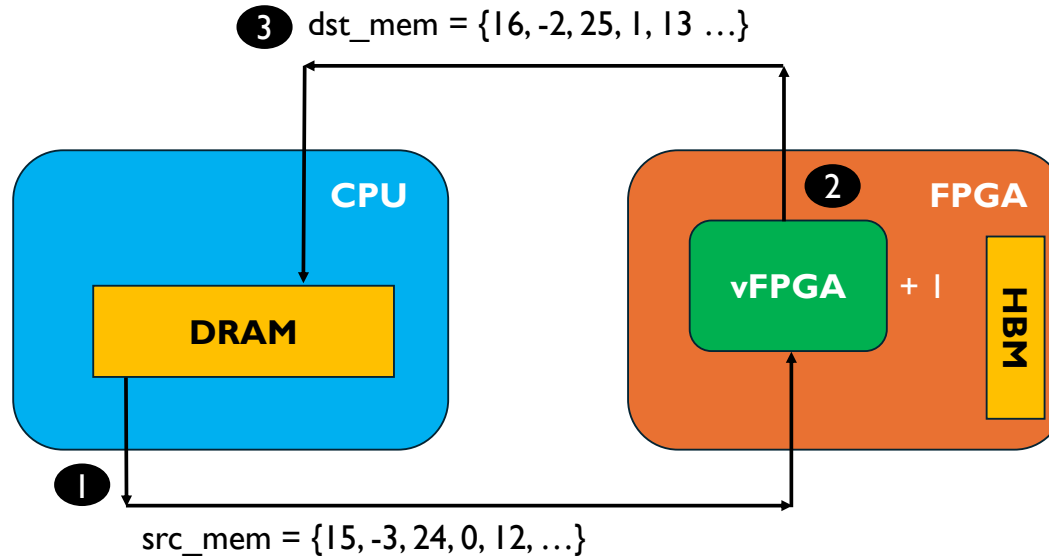


Open to the public;
for more details
follow the link:



Example I: Data movement & simple RTL kernel

- A simple pass-through kernel that pulls data from the host memory, adds one to every integer and writes them back to host memory



Example I: How to implement the hardware

```

// Data movement host memory => vFPGA => host memory
`ifdef EN_STRM
perf_local_inst_host_link (
    .axis_in    (axis_host_recv[0]),
    .axis_out   (axis_host_send[0]),
    .aclk       (aclk),
    .aresetn    (aresetn)
);
`endif

// Data movement card memory => vFPGA => card memory
`ifdef EN_MEM
perf_local_inst_card_link (
    .axis_in    (axis_card_recv[0]),
    .axis_out   (axis_card_send[0]),
    .aclk       (aclk),
    .aresetn    (aresetn)
);
`endif

// Tie-off unused signals to avoid synthesis problems
always_comb notify.tie_off_m();
always_comb sq_rd.tie_off_m();
always_comb sq_wr.tie_off_m();
always_comb cq_rd.tie_off_s();
always_comb cq_wr.tie_off_s();
always_comb axi_ctrl.tie_off_s();
    
```

Coyote User logic (next slide)

Wrapper

file

Wrapper

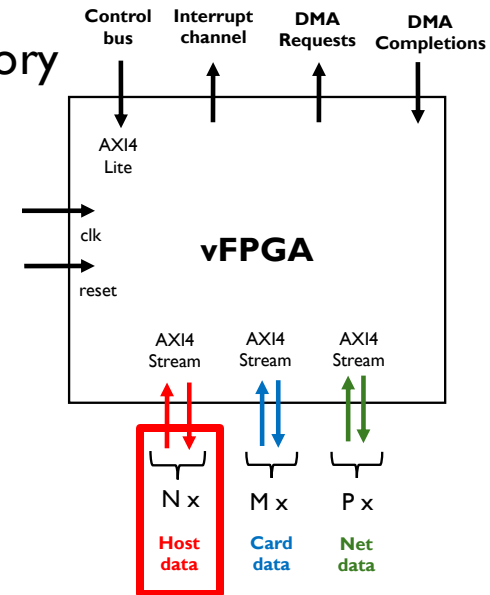
requests Data from card memory

called



Data from host memory

Data from card memory



Example I: How to implement the hardware

User logic

```
module perf_local (
    AXI4SR.s      axis_in,
    AXI4SR.m      axis_out,

    input logic   aclk,
    input logic   aresetn
);

// User logic; adding 1 to the input stream and writing it to the output stream
// The other signals (valid, ready etc.) are simply propagated
always_comb begin
    for(int i = 0; i < 16; i++) begin
        axis_out.tdata[i * 32 + :32] = axis_in.tdata[i * 32 + : 32] + 1;
    end

    axis_out.tvalid = axis_in.tvalid;
    axis_in.tready  = axis_out.tready;
    axis_out.tkeep  = axis_in.tkeep;
    axis_out.tid    = axis_in.tid;
    axis_out.tlast  = axis_in.tlast;
end
```

- ❑ A simple kernel that adds one to each input integer and writes to the output stream

- ❑ Interfaces follow the **AXI4 Stream** standard:
 - ❑ **TDATA**: 512-bit chunks of incoming data
 - ❑ **TVALID**: asserted high when the data is ready to be processed
 - ❑ **TREADY**: Back-pressure mechanism, indicating stream cannot be written to
 - ❑ **TLAST**: Last beat of stream
 - ❑ **TID**: Unique identifier for the stream

Example I: How to build the hardware

```
# CMake configuration
cmake_minimum_required(VERSION 3.5)
set(CYT_DIR ${CMAKE_SOURCE_DIR}/../../..)
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} ${CYT_DIR}/cmake)
find_package(CoyoteHW REQUIRED)
project(example_01_static_local)

# Enables streams from host memory (CPU DDR) and card memory (FPGA HBM or DDR)
set(EN_STRM 1)
set(EN_MEM 1)

# Number of vFPGAs (user applications)
set(N_REGIONS 1)

# Confirm that the selected options are allowed
validation_checks_hw()

# Load a user application in Configuration #0, Region #0
load_apps (
  VFPGA_C0_0 "src"
)

# Create the hardware project
create_hw()
```

- ❑ The hardware is configurable through *CMake* parameters
- ❑ Here we define:
 - ❑ A single data stream from host memory and card memory
 - ❑ And one user application which can be found inside the folder *src*
- ❑ Simply run **make** to get bitsream

Example I: How to implement the software

- 1 Create a Coyote thread and assign it to the target vFPGA

```
// Obtain a Coyote thread
coyote::cThread coyote_thread(DEFAULT_VFPGA_ID, getpid());
```

- 2 Allocate memory, huge or regular pages

```
if (hugepages) {
    src_mem = (int *) coyote_thread.getMem({coyote::CoyoteAllocType::HPF, max_size});
    dst_mem = (int *) coyote_thread.getMem({coyote::CoyoteAllocType::HPF, max_size});
} else {
    src_mem = (int *) coyote_thread.getMem({coyote::CoyoteAllocType::REG, max_size});
    dst_mem = (int *) coyote_thread.getMem({coyote::CoyoteAllocType::REG, max_size});
}
```

Example I: How to implement the software

3 Create a scatter-gather entry

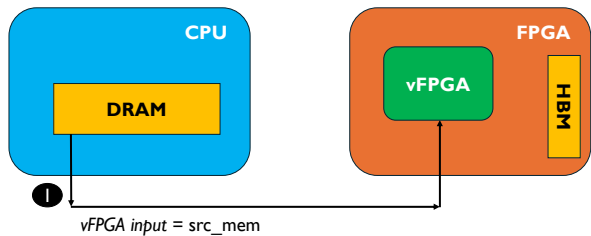
```
coyote::localSg src_sg = { .addr = src_mem, .stream = stream };  
coyote::localSg dst_sg = { .addr = dst_mem, .stream = stream };
```

Where does the data reside?
HOST (1) or CARD (0)

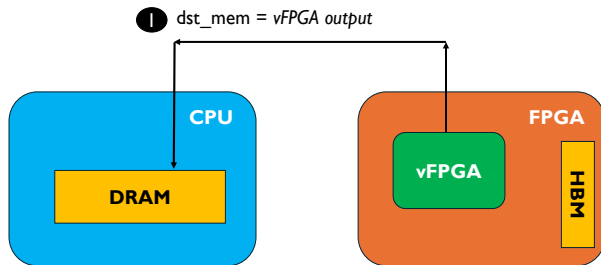
4 Launch multiple transfers in parallel

```
for (int i = 0; i < transfers; i++) {  
    coyote_thread.invoke(coyote::CoyoteOper::LOCAL_TRANSFER, src_sg, dst_sg);  
}  
  
// Wait until all of them are finished; short sleep to avoid busy-waiting  
while (coyote_thread.checkCompleted(coyote::CoyoteOper::LOCAL_TRANSFER) != transfers) {}
```

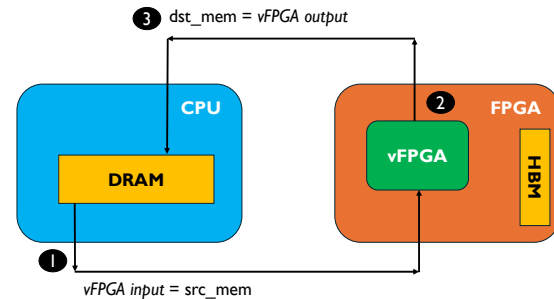
LOCAL_READ



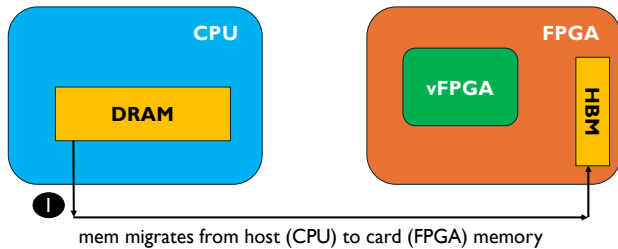
LOCAL_WRITE



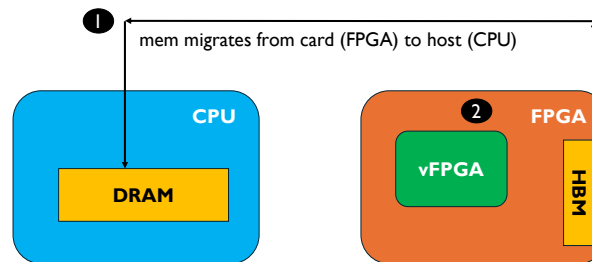
LOCAL_TRANSFER



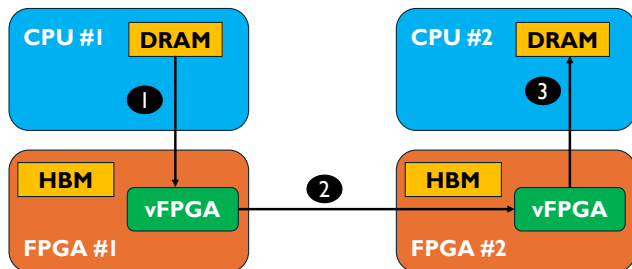
LOCAL_OFFLOAD



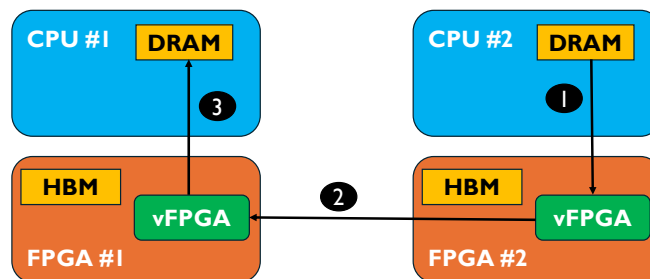
LOCAL_SYNC



REMOTE_WRITE



REMOTE_READ



Example I: Live demo

```
bramhorst@hacc-box-03:~/coyote-asplos/examples/01_static_local/sw/build$ bin/test
```

```
-- CLI PARAMETERS:
```

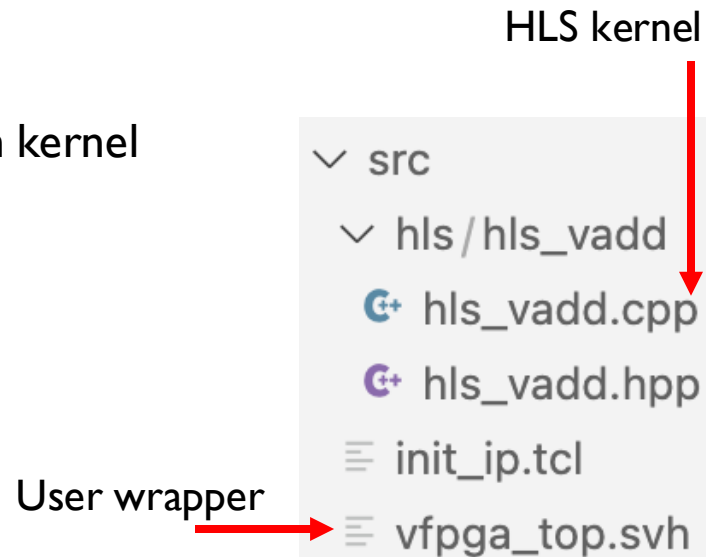
```
-----  
Enable hugepages: 1  
Enable mapped pages: 1  
Data stream: HOST  
Number of test runs: 100  
Starting transfer size: 64  
Ending transfer size: 4194304
```

```
-- PERF LOCAL
```

```
-----  
Size:      64; Average throughput: 120.224 MB/s; Average latency: 3.04415 us  
Size:     128; Average throughput:  243.47 MB/s; Average latency: 3.05475 us  
Size:     256; Average throughput: 482.603 MB/s; Average latency: 3.12227 us  
Size:     512; Average throughput: 957.475 MB/s; Average latency: 3.15518 us  
Size:    1024; Average throughput: 1833.96 MB/s; Average latency:  3.2342 us  
Size:    2048; Average throughput: 3474.03 MB/s; Average latency:  3.4335 us  
Size:    4096; Average throughput: 6395.09 MB/s; Average latency:  3.7086 us  
Size:    8192; Average throughput: 8973.59 MB/s; Average latency:  4.15676 us  
Size:   16384; Average throughput: 11126.6 MB/s; Average latency:  4.87368 us  
Size:   32768; Average throughput: 11658.6 MB/s; Average latency:  6.07235 us  
Size:   65536; Average throughput: 11943.7 MB/s; Average latency:  8.57423 us  
Size:  131072; Average throughput: 12073.4 MB/s; Average latency: 13.6323 us  
Size:  262144; Average throughput: 12142.5 MB/s; Average latency: 23.8809 us  
Size:  524288; Average throughput: 12173.9 MB/s; Average latency: 44.3695 us  
█
```

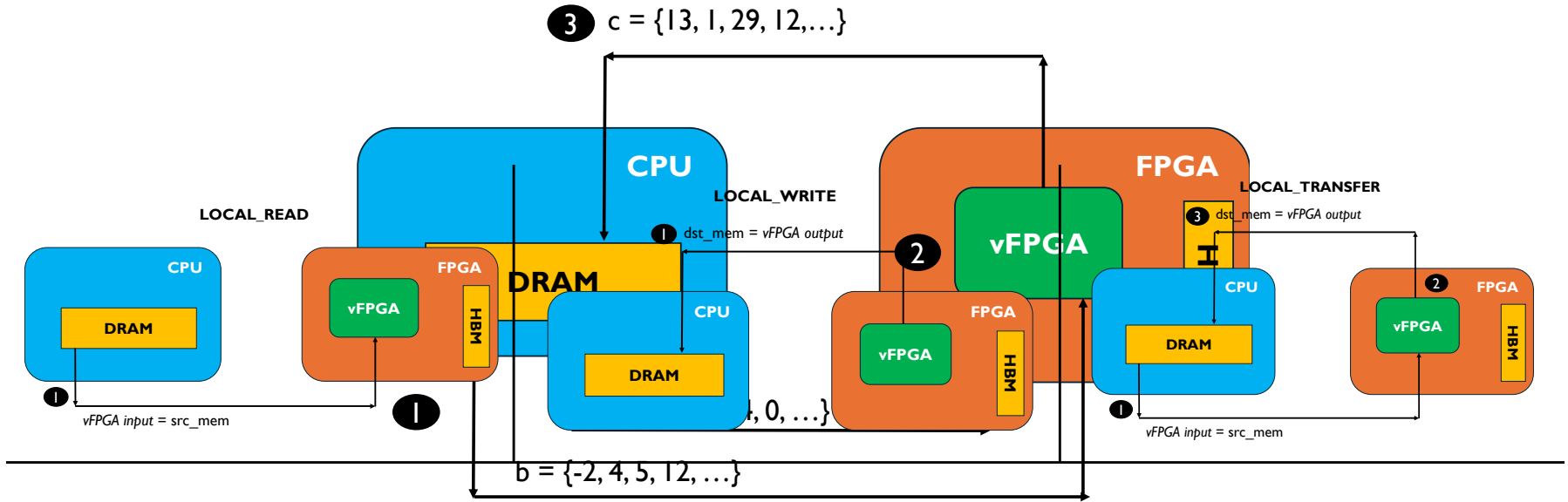
Example 2: Deploying HLS kernels

- ❑ HLS is programming paradigm that enables developing FPGA applications in C++ with additional pragmas that dictate hardware behaviour
 - ❑ We will not cover how to write HLS kernels today; please refer to other materials for that
- ❑ In this example, we consider a simple vector addition kernel
- ❑ Coyote will automatically compile any HLS kernel in the project, as long as they are in the correct folder
- ❑ Development process stays the same as in Example 1: user wrapper + user logic



Example 2: Overview

- We will send the input vectors, a and b, via dedicated data streams, and set the vector c back to host memory



Example 2: The hardware

```
void hls_vadd (  
    hls::stream<axi_s> &axi_in1,  
    hls::stream<axi_s> &axi_in2,  
    hls::stream<axi_s> &axi_out  
) {  
    // A free-running kernel; no control interfaces needed to start the openCL kernel  
    #pragma HLS INTERFACE ap_ctrl_none port=return  
  
    // Specify that the input/output signals are AXI streams (axis)  
    #pragma HLS INTERFACE axis register port=axi_in1 name=s_axi_in1  
    #pragma HLS INTERFACE axis register port=axi_in2 name=s_axi_in2  
    #pragma HLS INTERFACE axis register port=axi_out name=m_axi_out  
}
```

```
hls_vadd inst_vadd(  
    .s_axi_in1_TDATA      (axis_host_rcv[0].tdata),  
    .s_axi_in1_TKEEP     (axis_host_rcv[0].tkeep),  
    .s_axi_in1_TLAST     (axis_host_rcv[0].tlast),  
    .s_axi_in1_TSTRB     (0),  
    .s_axi_in1_TVALID    (axis_host_rcv[0].tvalid),  
    .s_axi_in1_TREADY    (axis_host_rcv[0].tready),  
  
    .s_axi_in2_TDATA      (axis_host_rcv[1].tdata),  
    .s_axi_in2_TKEEP     (axis_host_rcv[1].tkeep),  
    .s_axi_in2_TLAST     (axis_host_rcv[1].tlast),  
    .s_axi_in2_TSTRB     (0),  
    .s_axi_in2_TVALID    (axis_host_rcv[1].tvalid),  
    .s_axi_in2_TREADY    (axis_host_rcv[1].tready),  
  
    .m_axi_out_TDATA      (axis_host_send[0].tdata),  
    .m_axi_out_TKEEP     (axis_host_send[0].tkeep),  
    .m_axi_out_TLAST     (axis_host_send[0].tlast),  
    .m_axi_out_TSTRB     (),  
    .m_axi_out_TVALID    (axis_host_send[0].tvalid),  
    .m_axi_out_TREADY    (axis_host_send[0].tready),  
  
    .ap_clk                (aclk),  
    .ap_rst_n              (aresetn)  
);
```

First input stream from host

Second input stream from host

Out-going stream to host

Example 2: The software

- 1 Create a *Coyote thread* and allocate memory (like in the 1st example)
- 2 Create a scatter-gather entry for the 3 data transfers

```
coyote::localSg sg_a = {.addr = a, .len = size * (uint) sizeof(float), .dest = 0};  
coyote::localSg sg_b = {.addr = b, .len = size * (uint) sizeof(float), .dest = 1};  
coyote::localSg sg_c = {.addr = c, .len = size * (uint) sizeof(float), .dest = 0};
```

- 3 Perform the operation

```
// Run kernel and wait until complete  
coyote_thread.invoke(coyote::CoyoteOper::LOCAL_READ, sg_a);  
coyote_thread.invoke(coyote::CoyoteOper::LOCAL_READ, sg_b);  
coyote_thread.invoke(coyote::CoyoteOper::LOCAL_WRITE, sg_c);  
while (  
    coyote_thread.checkCompleted(coyote::CoyoteOper::LOCAL_WRITE) != 1 ||  
    coyote_thread.checkCompleted(coyote::CoyoteOper::LOCAL_READ) != 2  
) {}
```

Example 2: Live demo

```
bramhorst@hacc-box-03:~/coyote-asplos/examples/02_hls_vadd/sw/build$ bin/test -s 10000
```

```
-- Validation: HLS vector addition
```

```
-----  
Vector elements: 10000
```

```
-- Validation passed!
```

```
-----  
bramhorst@hacc-box-03:~/coyote-asplos/examples/02_hls_vadd/sw/build$ █
```

Example 3: RDMA networking

- ❑ RDMA works on the notion of *Queue Pairs (QPs)*, which uniquely identify a connection between each node.
- ❑ Each QP contains many details about the connection, but the most important ones are:
 - ❑ IP address of the two nodes (each node stores its own and the remote IP)
 - ❑ Virtual address of the RDMA buffer → recall, in RDMA data is written directly to a target buffer and its virtual address must be known.
 - ❑ Unique queue pair number (QPN)

Example 3: Implementing the software

- 1 Create a *Coyote thread* and allocate memory (like in the 1st example)
- 2 Initialize RDMA, by simply calling the **initRDMA()** function, which exchanges the information with a remote node (specified by IP address in 3rd parameter), and then allocates a buffer which can be used for RDMA operation

```
int *mem = (int *) coyote_thread.initRDMA(max_size, coyote::DEF_PORT, server_ip.c_str());
```

Virtual address of the allocated buffer

Buffer size

Connection port

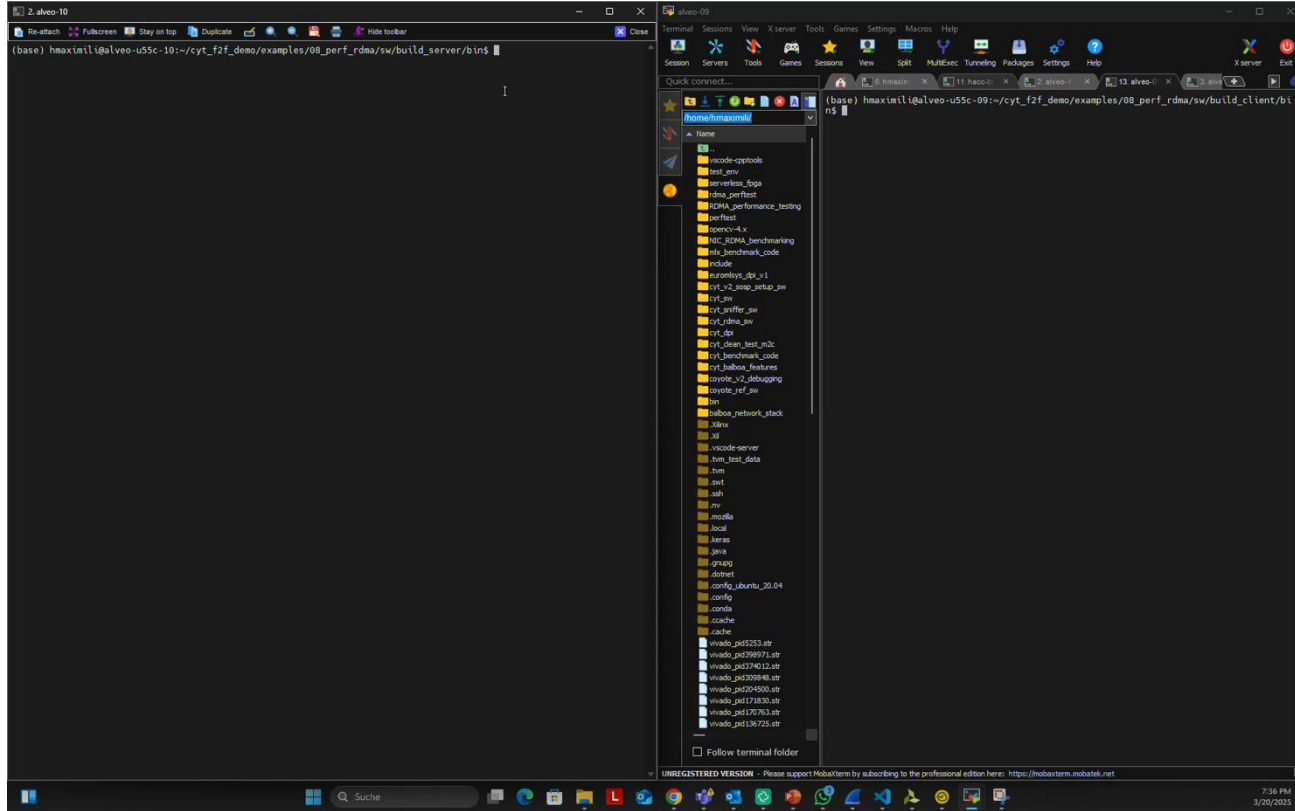
IP address of remote node

Example 3: Implementing the software

- 3 Execute RDMA operation on one node; can either be a RDMA_READ or RDMA_WRITE

```
coyote::Coyote0per coyote_operation = operation ? coyote::Coyote0per::REMOTE_RDMA_WRITE : coyote::Coyote0per::REMOTE_RDMA_READ;
for (int i = 0; i < transfers; i++) {
    coyote_thread.invoke(coyote_operation, sg);
}
```

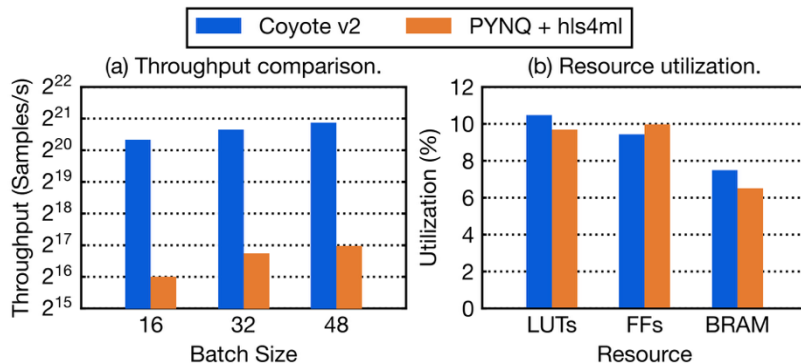
Example 3: Live Demo (writes)



The screenshot displays a desktop environment with two windows. The left window is a terminal titled '2, alive-10' with the command prompt '(base) hmaximil@alive-u55c-10:~/cyt_f2f_demo/examples/08_perf_rdma/sw/build_server/bin\$'. The right window is a file explorer titled 'alive-09' showing the file system structure of the user's home directory. The file explorer shows a list of folders including 'hmaximil', 'hcode-cppitools', 'hcode-gov', 'serverless-fpga', 'rdma_perf_test', 'RDMA_performance_testing', 'perf-test', 'spency-4-x', 'h3C_RDMA_benchmarking', 'h3c_benchmark_code', 'include', 'kubernetes_db_v1', 'cyt_v2_setup_setup_sw', 'cyt_sw', 'cyt_swfilter_sw', 'cyt_rdma_sw', 'cyt_09', 'cyt_clean_test_m3c', 'cyt_benchmark_code', 'cyt_balboa_features', 'koyote_v2_debugging', 'koyote_rst_sw', 'bin', 'balboa_network_stack', 'libra', 'xi', 'xscod-server', 'xim_test_data', 'xim', 'xint', 'xsh', 'xsv', 'xtools', 'local', 'karma', 'kava', 'kgroup', 'dotnet', 'confg_bunbu_20.04', 'confg', 'conda', 'cache', 'hcode', 'hwado_pd52523.str', 'hwado_pd398971.str', 'hwado_pd1790111.str', 'hwado_pd2020948.str', 'hwado_pd204500.str', 'hwado_pd171830.str', 'hwado_pd170763.str', and 'hwado_pd182725.str'. The terminal window is currently empty, and the file explorer is showing the contents of the 'hmaximil' directory.

Putting it all together: CoyoteAccelerator for hls4ml

- We integrated Coyote as a backend with hls4ml, an open-source framework for real-time inference of neural network on FPGAs
- Full control, synthesis and deployment in Python



```
# Load TensorFlow/Keras model and dataset
model = load_model('sample_keras_model.h5')
X = np.load('sample_data.npy')

# Create hls4ml model targetting Coyote backend
hls_config = config_from_keras_model(keras_model)
hls_model = convert_from_keras_model(
    keras_model, hls_config=hls_config,
    output_dir='/path/to/output/dir',
    backend='CoyoteAccelerator', clock_period=4,
    input_data_tb='sample_data.npy',
    output_data_tb=f'sample_labels.npy'
)

# Compile and run software emulation
hls_model.compile()
pred_emu = hls_model.predict(X)

# Start hardware synthesis
hls_model.build()

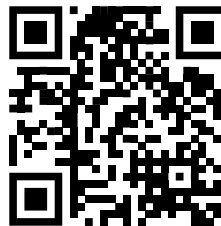
# Once done, create an "Overlay" of the vFPGA
overlay = CoyoteOverlay('/path/to/output/dir')
overlay.program_fpga()

# Run inference on hardware
pred_fpga = overlay.predict(X, (1,), BATCH_SIZE)
```

Conclusions

- ❑ Coyote v2 aims to become the „Linux of FPGAs“: a one-stop solution for all forms of application acceleration on reconfigurable hardware.
- ❑ For this purpose, it implements the following:
 - ❑ Versatile user interfaces in both hardware and software
 - ❑ Support for standard RoCE v2 networking at 100G
 - ❑ Multi-tenancy and run-time reconfiguration
 - ❑ FPGA – GPU DMA

Coyote v2 & BALBOA (arXiv pre-prints)



Coyote v2 (GitHub)



CoyoteAccelerator
for hls4ml (GitHub)



Acknowledgments

We would like to thank AMD for the continuous support and collaboration in the development of various projects, including Coyote v2, ACCL, BALBOA etc. Additionally, we would like to thank AMD for the generous donation of the Heterogenous Acceleration Compute Cluster (HACC), which was used for the development and testing of Coyote v2.