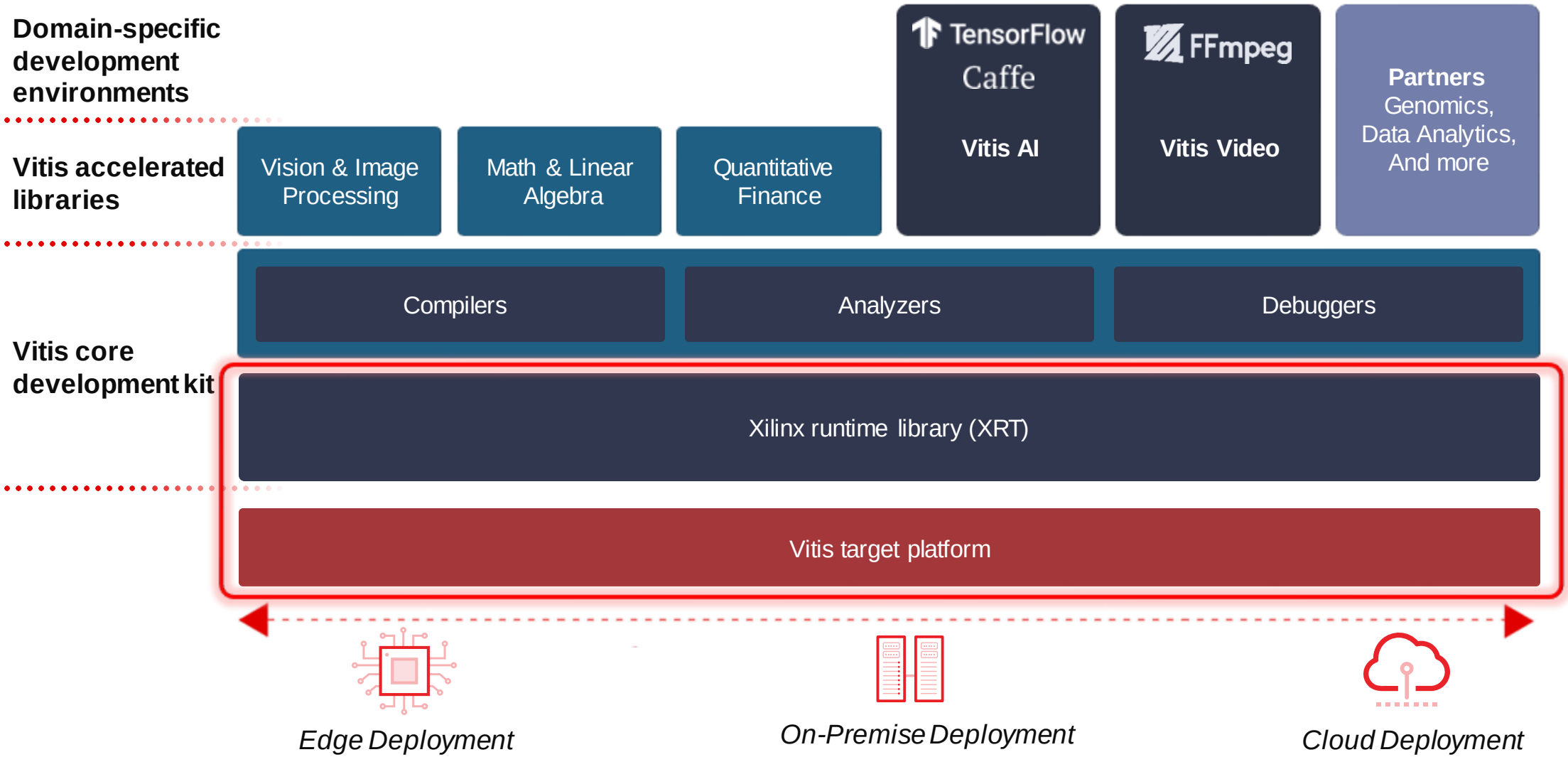
XILINX

XRT Essentials

Dec. 2020

Xilinx Adapt Conf

Vitis Unified Software Platform

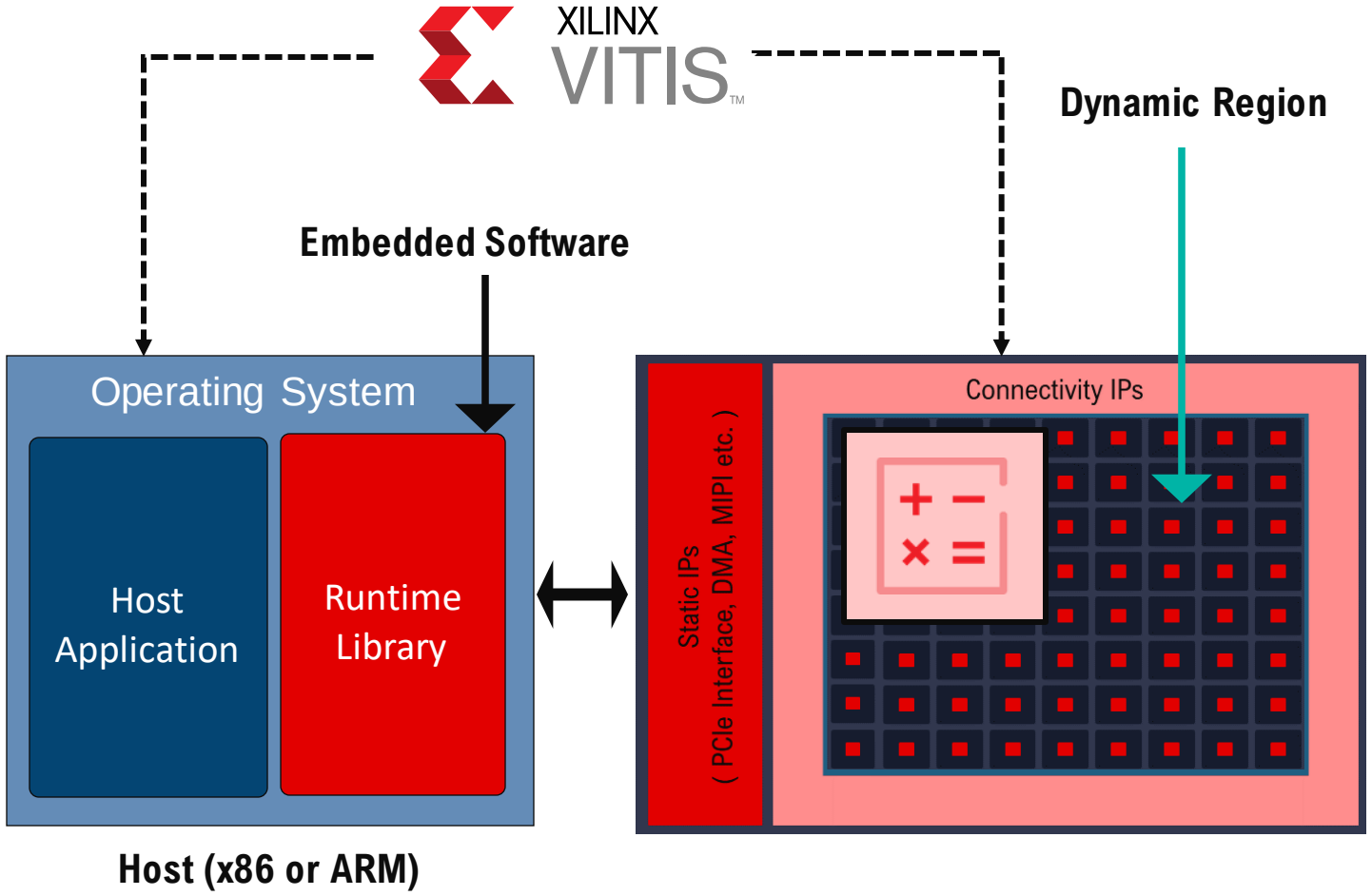


XILINX.

Vitis Target Platform

Base Hardware, Software Architecture

Ready-to-Use Vitis Target Platforms
OR
Build Your Own Embedded Platforms

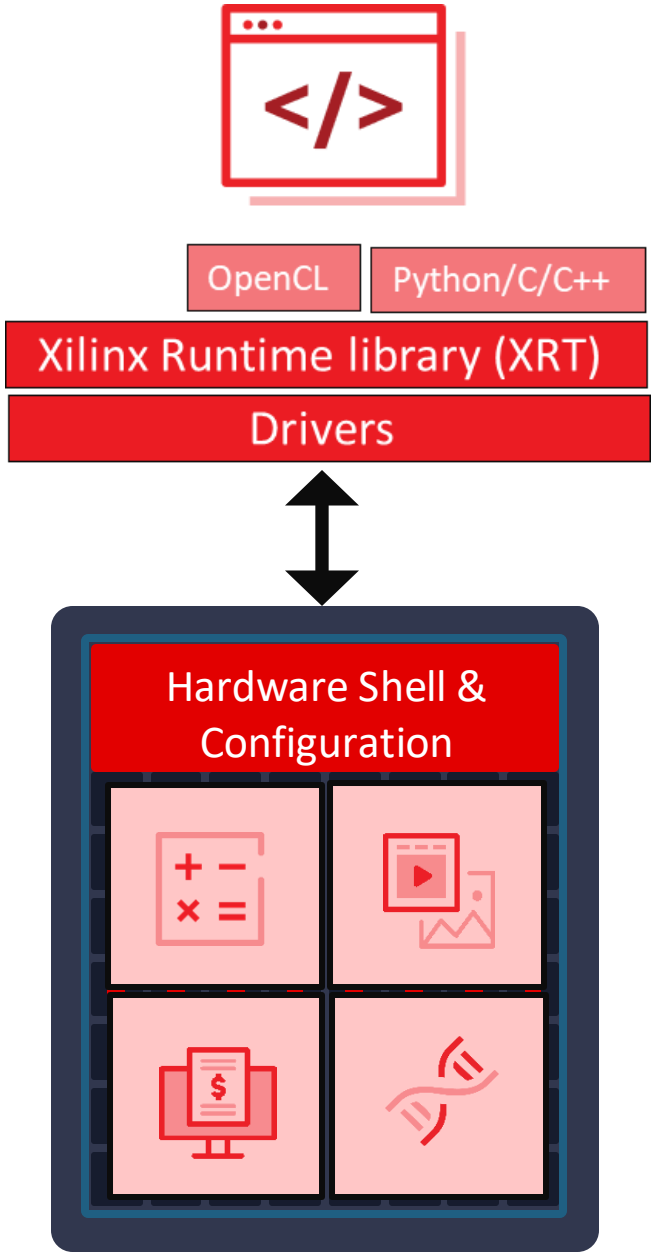


XILINX

Xilinx Runtime library (XRT)

- **Standardized, Open-Source Software Interface**
  - Between Host Application and Accelerators on Device
  - Use low-level APIs or OpenCL, Python bindings

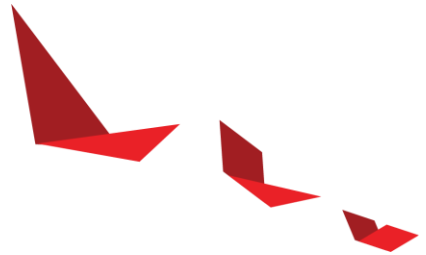
Open Source, Available on GitHub: https://github.com/Xilinx/XRT



- **Responsible For**
  - FPGA Image Download
    - *Accelerator Binaries*
  - Memory Management
    - *Data Movement between Host and Device*
  - Execution Management
    - *Trigger, Sequence & Synchronize Computations*
  - Board Management
    - *Board Recovery, Power Measurement etc.*

What You'll Learn

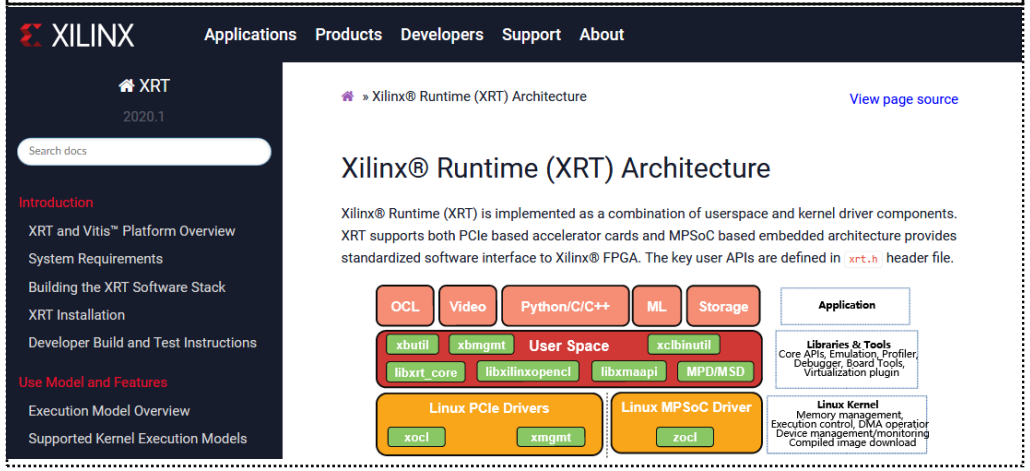
The Agenda



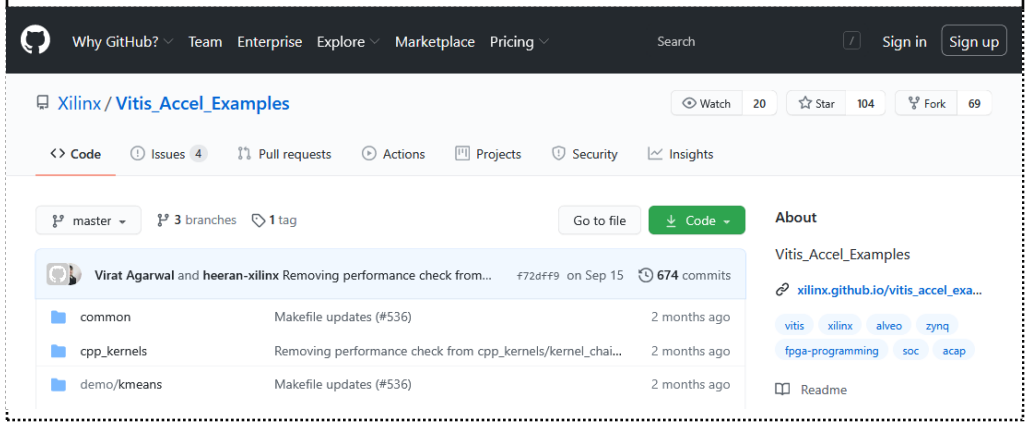
- Section 1: What is the basic Vitis app workflow?
- Section 2: How to optimize for performance?
- Section 3: Advanced Features of XRT

Documents, Examples, Tutorials

https://xilinx.github.io/XRT

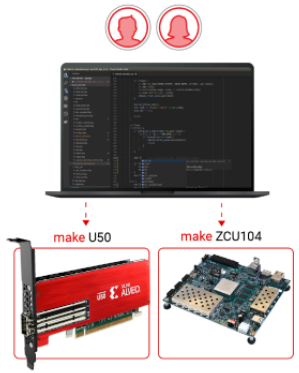


https://github.com/Xilinx/vitis_accel_examples



https://github.com/Xilinx/Vitis-In-Depth-Tutorial

The Vitis In-Depth Tutorials takes users through the design methodology and programming model for deploying accelerated application on all Xilinx platforms.



Introduction

Start here! Learn the basics of the Vitis programming model by putting together your very first application. No experience necessary!

Machine Learning Tutorial

Learn how to use Vitis, Vitis-AI, and the Vitis accelerated libraries to implement a fully end-to-end accelerated application using purely software-defined flows - no hardware expertise required.

Use Vitis-AI to configure Xilinx hardware using the Tensorflow framework. Vitis-AI allows the user to quantize, compile, and deploy an inference model in a matter of minutes.

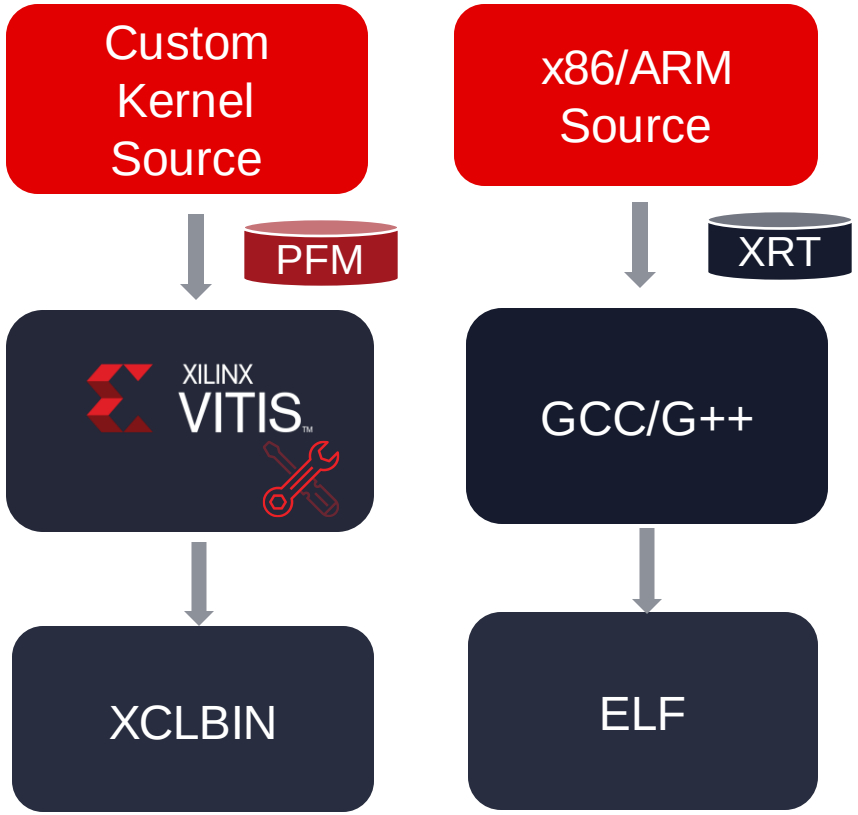
Acceleration Tutorial

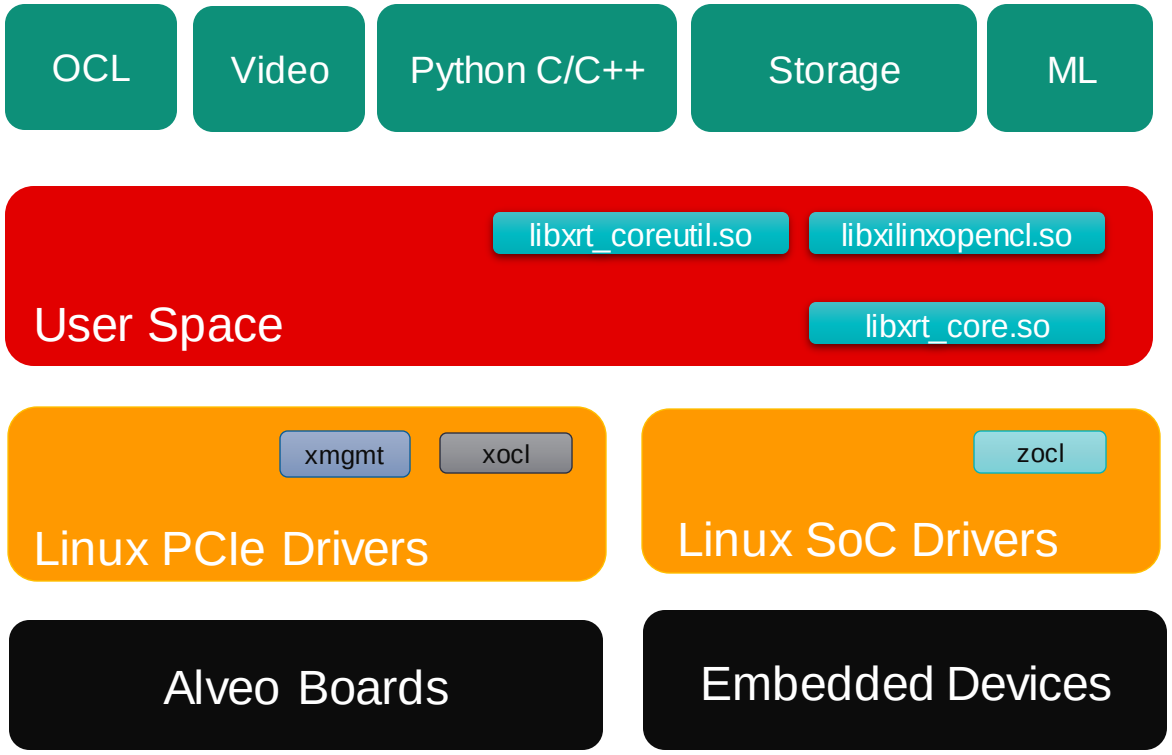
Learn how to use the Vitis core development kit to build, analyze, and optimize an accelerated algorithm developed in C++, OpenCL, and even low-level hardware description languages (HDLs) like Verilog and VHDL.

Section 1
Basic Workflow

Building Binaries - FPGA and Host

Common for all Devices





Programming Model

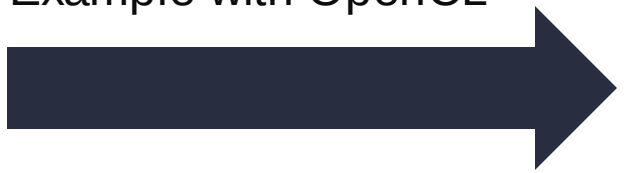
Vector Addition Example

- Normal Local Programming

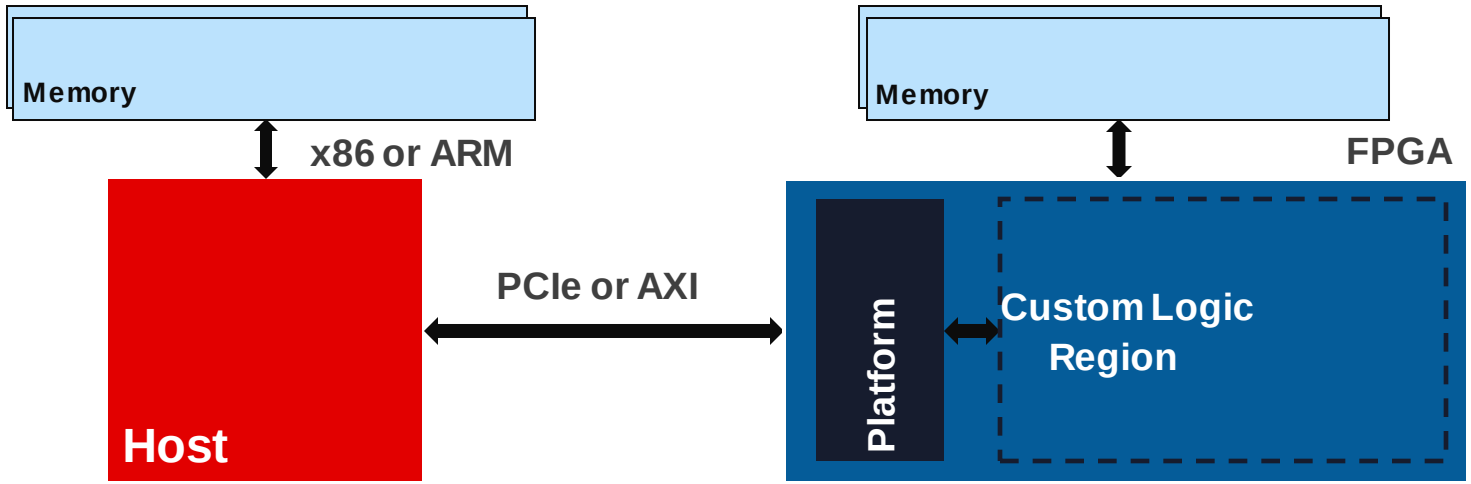
```
// Vector addition example
int a[10], b[10], c[10];

// Initialize a and b

// Vector Addition
for (int i = 0; i<10; i++) {
    c[i] = a[i] + b[i];
}
```

- Heterogenous Architecture Programming

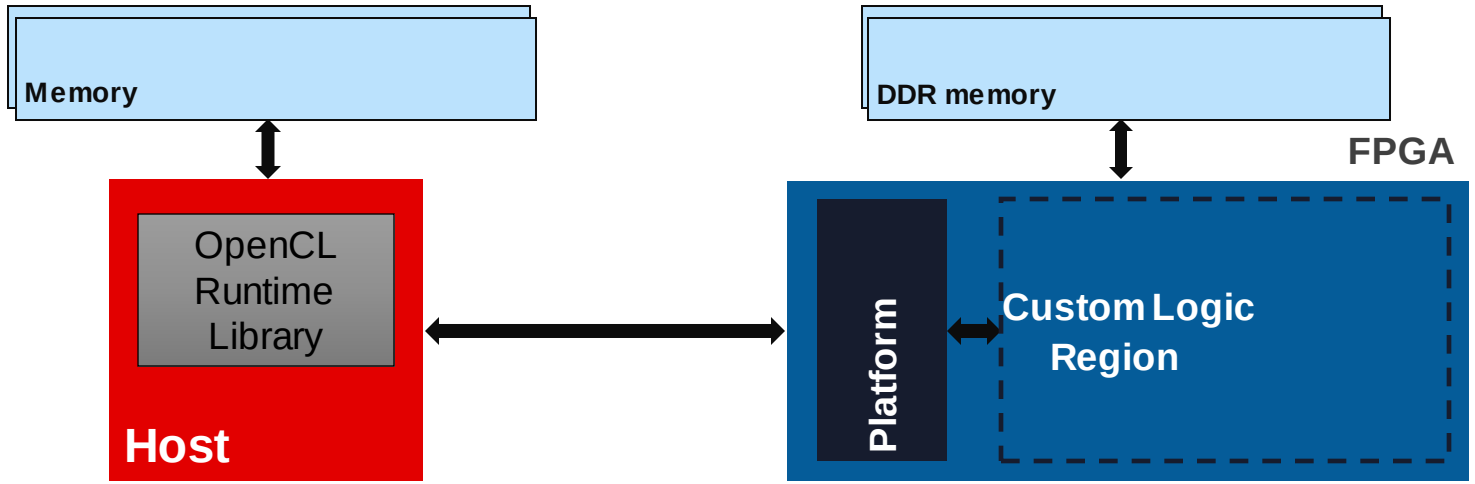
Example with OpenCL



1. Power Up

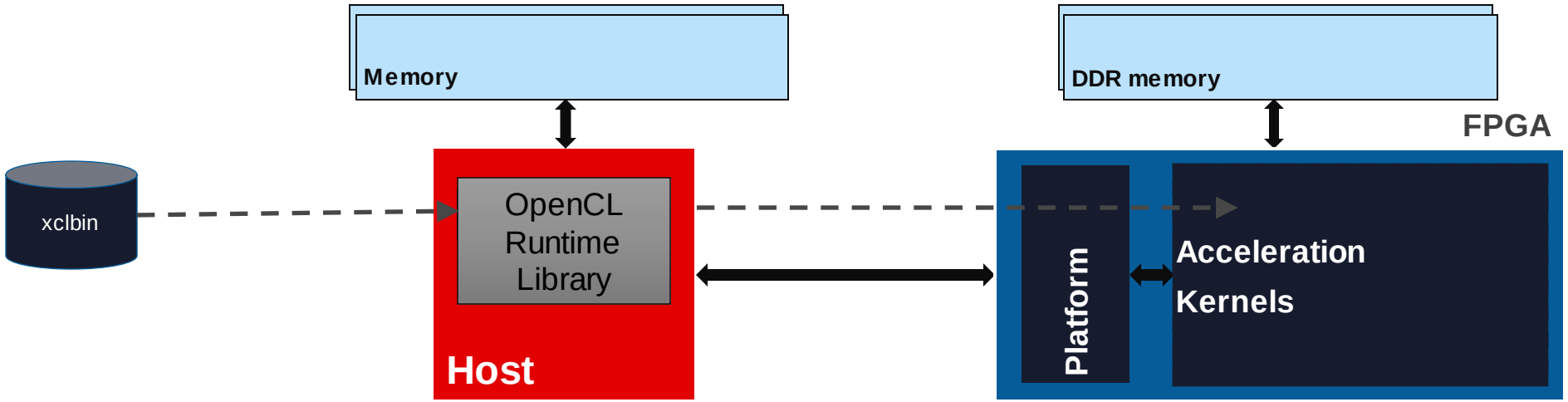


2. Runtime Initialization




```
// Create OpenCL Device (consider it as FPGA)
cl::Device device;

// Search available devices and assign to device.
// ... Omitted code block

// Create OpenCL context (consider it as Vitis Platform)
cl::Context context(device); // context -> Vitis Platform

// Create OpenCL command queue (to send commands to the FPGA)
cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);
```

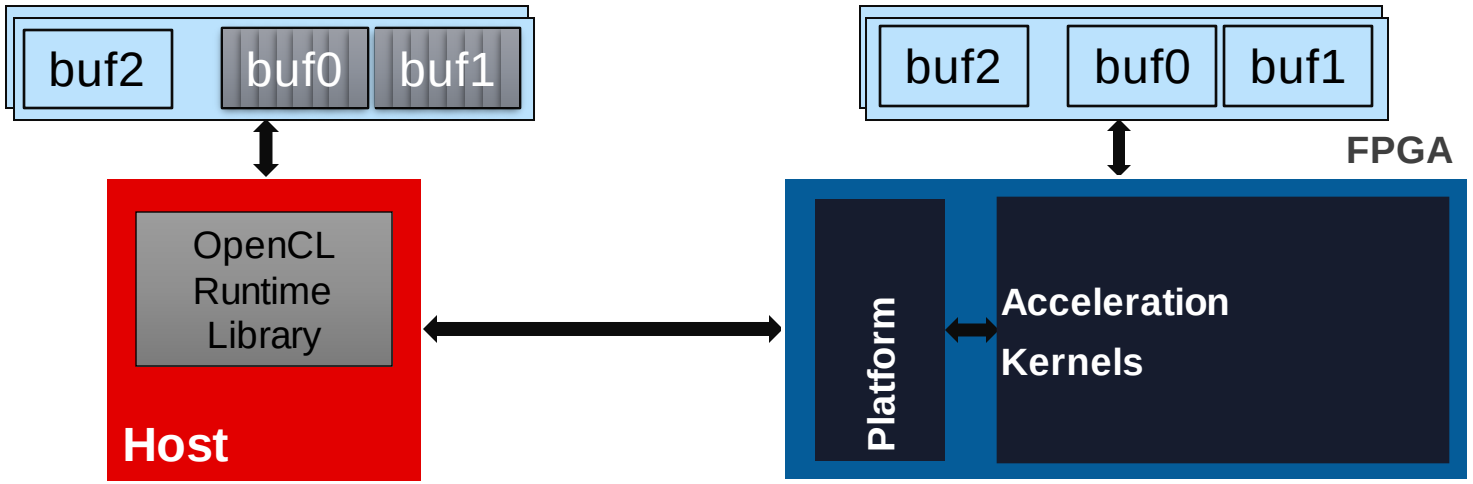
3. Device Configuration




```
cl::Program::Binaries bins;

// Load xclbin to bins
// ... Code procedure omitted

cl::Program program(context, devices, bins);
```

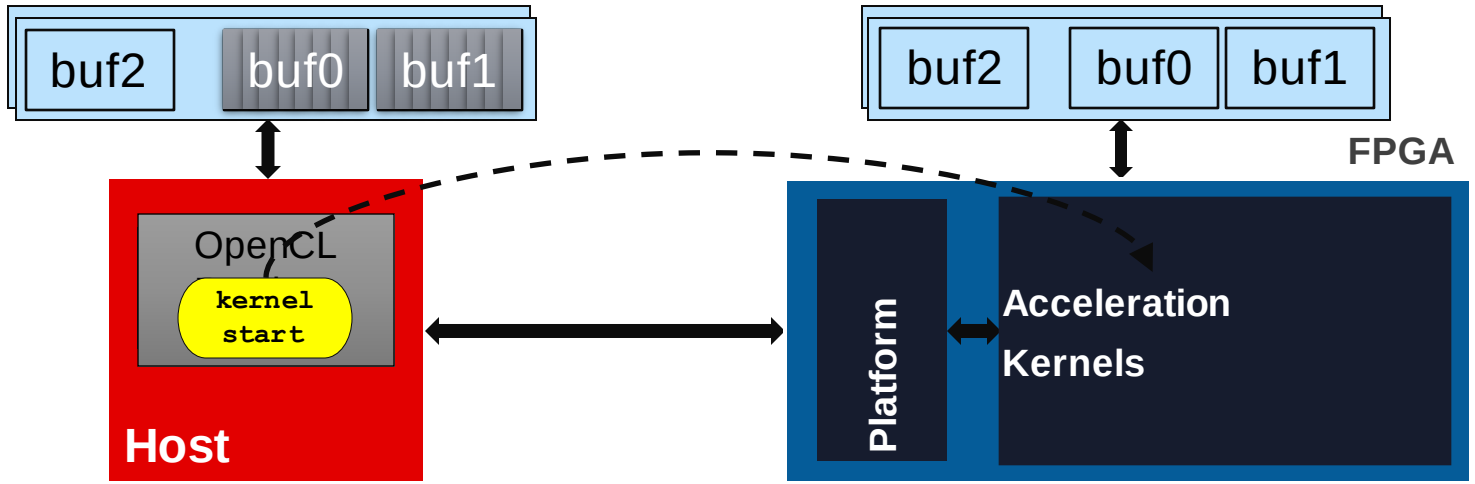

XILINX.

4. Buffer Allocation



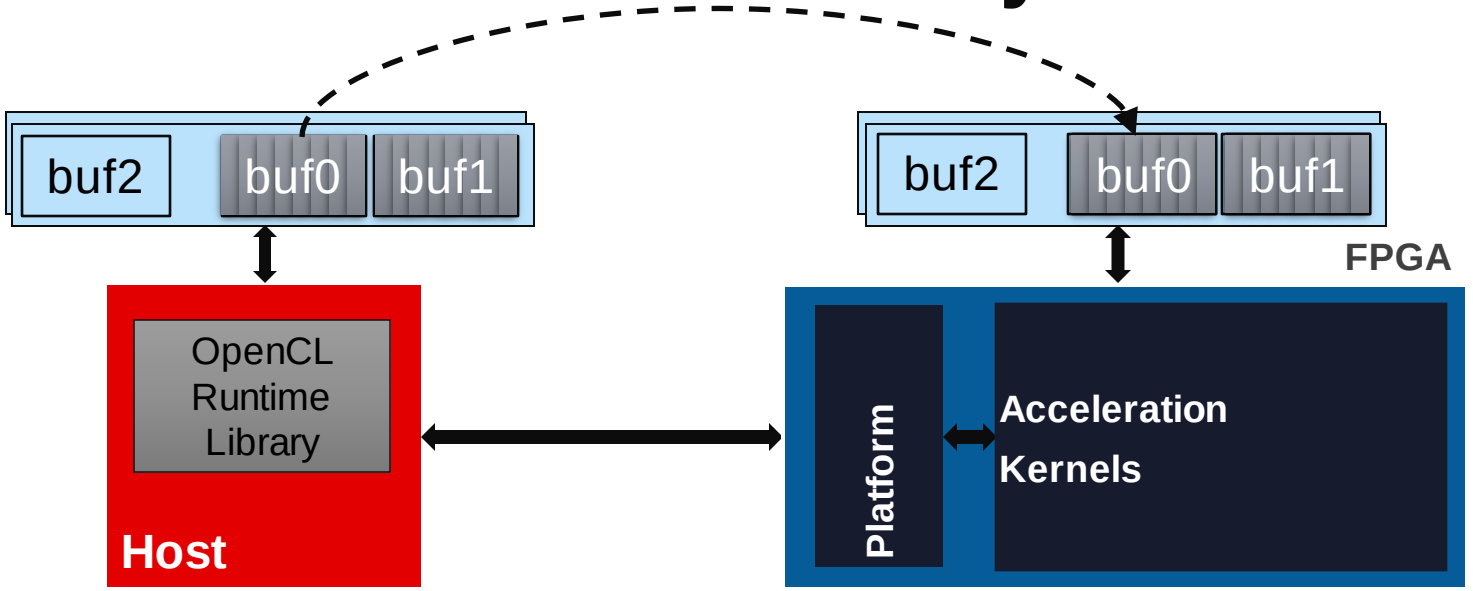

```
cl::Buffer buffer_a(context, CL_MEM_READ_ONLY, size_in_bytes);
cl::Buffer buffer_b(context, CL_MEM_READ_ONLY, size_in_bytes);
cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY, size_in_bytes);
```

5. Setup Kernel Arguments



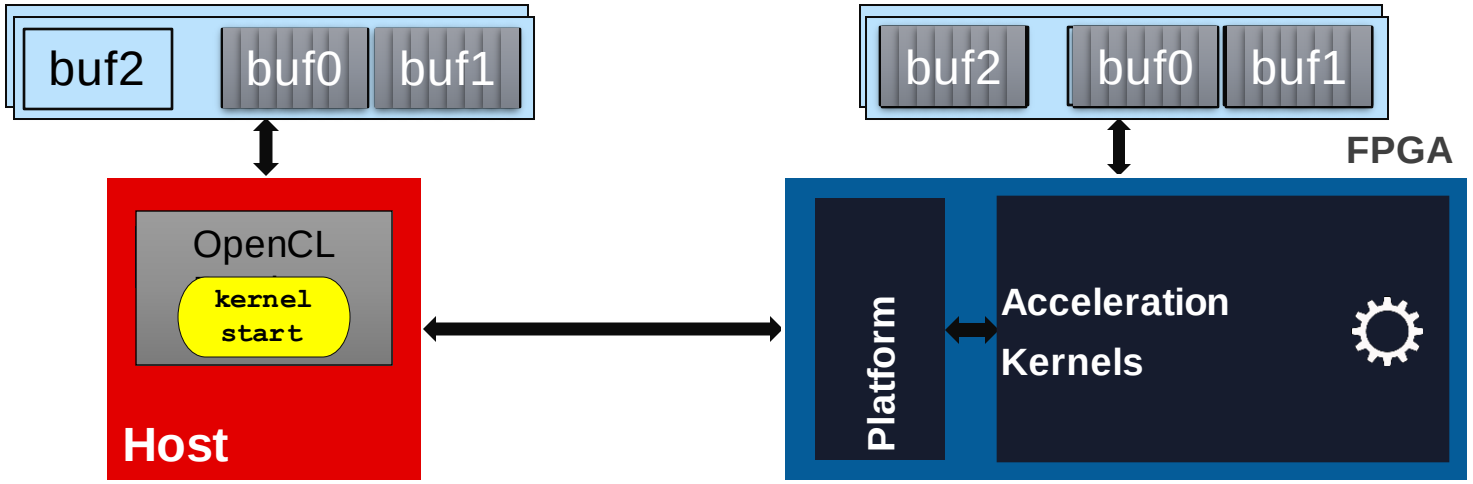

```
cl::Kernel krnl_vector_add(program,"krnl_vadd");
krnl_vector_add.setArg(0, buffer_a);
krnl_vector_add.setArg(1, buffer_b);
krnl_vector_add.setArg(2, buffer_result);
krnl_vector_add.setArg(3, DATA_SIZE);
```

6. Writing Buffers to FPGA Memory



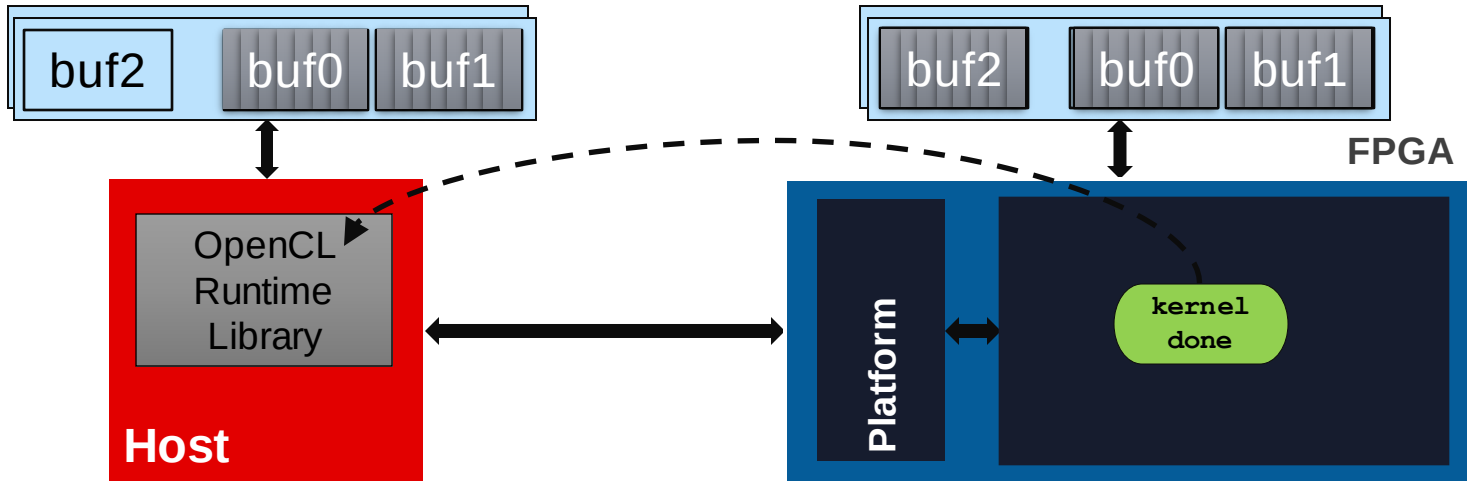

```
q.enqueueMigrateMemObjects({buffer_a,buffer_b},0/* 0 means from host*/);
```

7. Running the Accelerators



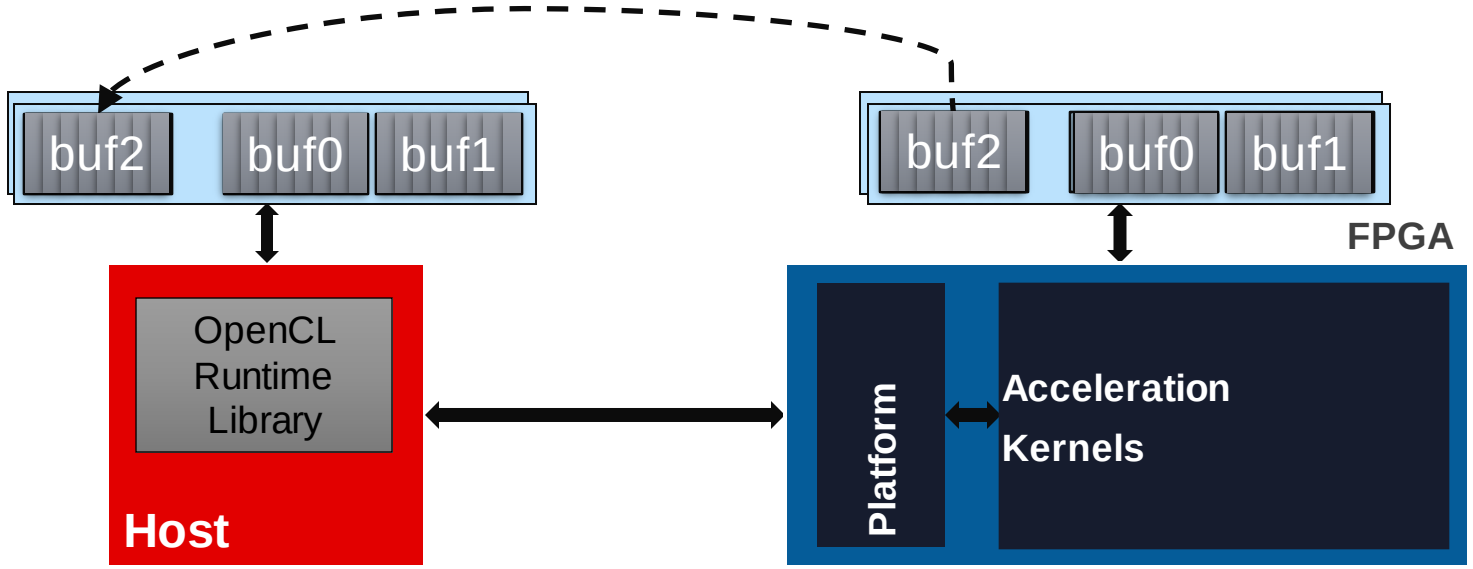

```
q.enqueueTask(krnl_vector_add);
```

7. Running the Accelerators (Cont'd)



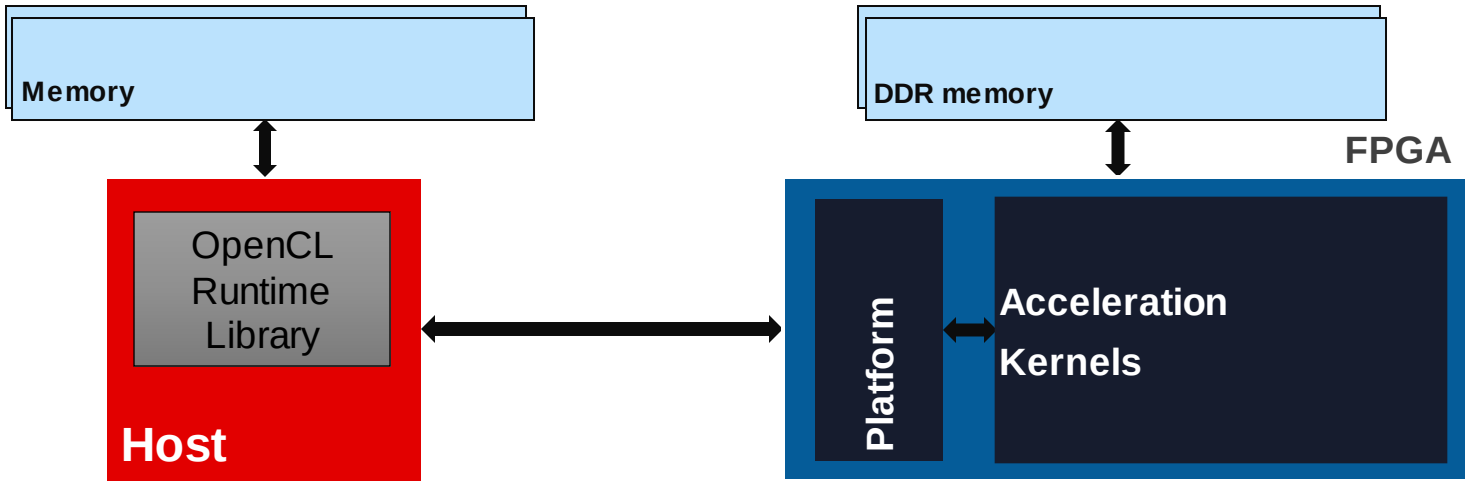

```
// Alternative out-of-order queue events control
cl::Event event_sp;
q.enqueueTask(krnl_vector_add, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

8. Reading Buffers from FPGA Mem




```
q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST);

// wait for data moving finish
q.finish();
```

9. Clean Up




```
q.enqueueUnmapMemObject(buffer_a , ptr_a);
q.enqueueUnmapMemObject(buffer_b , ptr_b);
q.enqueueUnmapMemObject(buffer_result , ptr_result);
q.finish();
```

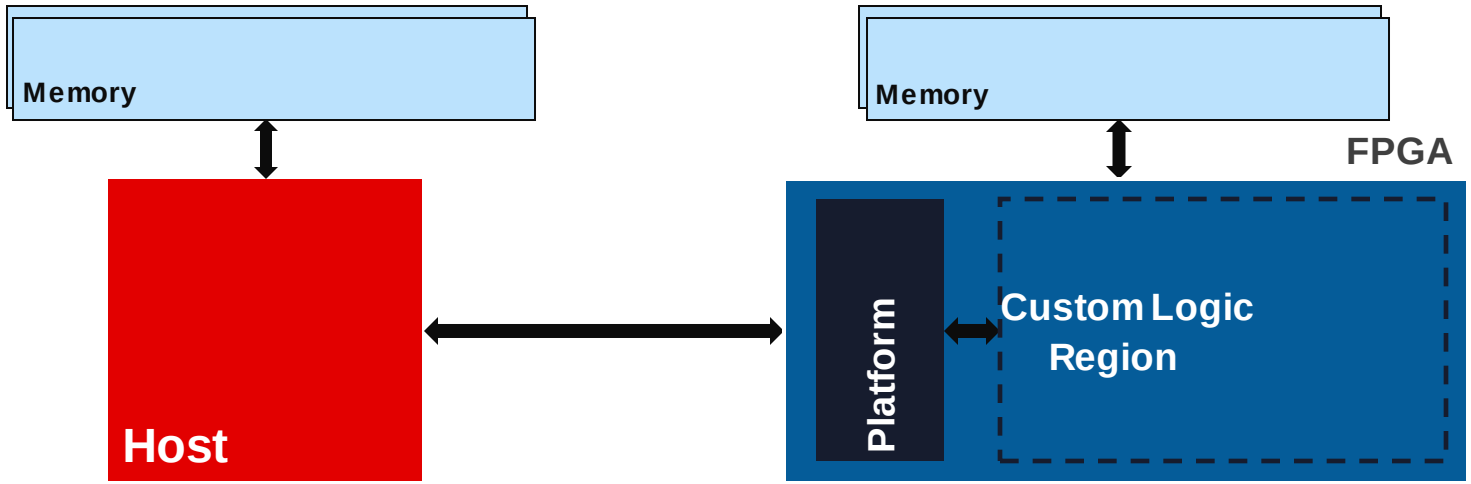
XRT API Groups

https://xilinx.github.io/XRT/master/html/xrt_native_apis.html

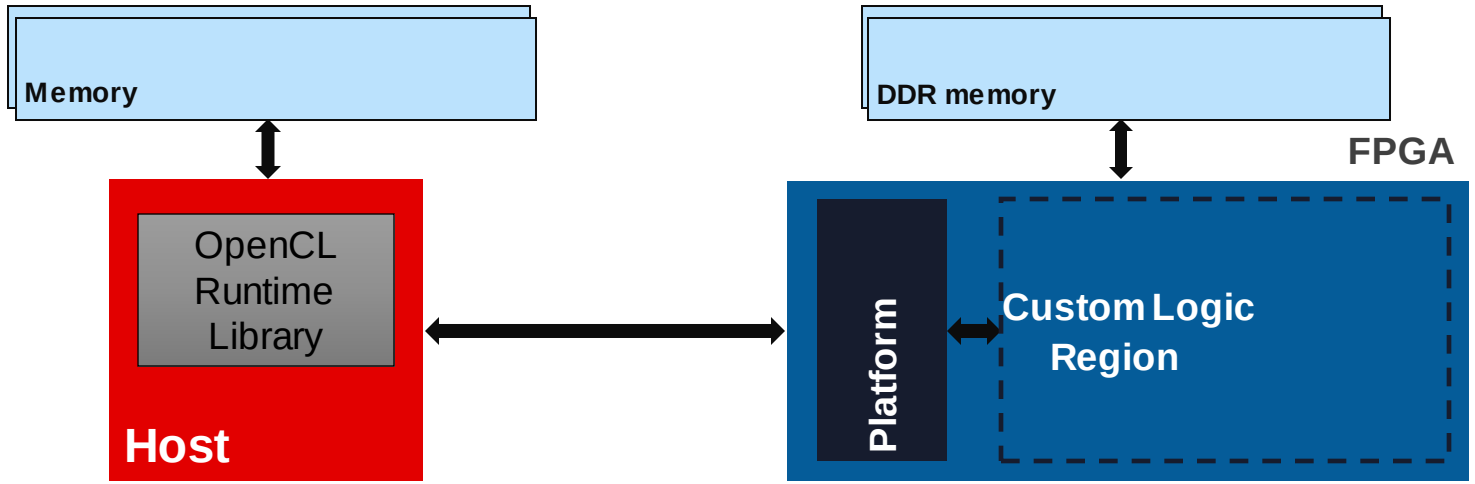
API Groups	OpenCL API	XRT API
Device and Context *Open a device*	cl::Device cl::Context	xrt::device
XCLBIN *Download XCLBIN*	cl::Program	xrt::xclbin
Kernel *Set kernel args, kernel handler for execution, etc*	cl::Kernel	xrt::kernel
Buffer *Allocate, deallocate buffer, data transfer, buffer property setting*	cl::Buffer	xrt::bo
Execution *Run, wait*	cl::CommandQueue	xrt::run

	`krnl_vector_add.setArg(0,buffer_a);`
	`krnl_vector_add.setArg(1,buffer_b);`
	`krnl_vector_add.setArg(2,buffer_result);`
	`krnl_vector_add.setArg(3,DATA_SIZE);`
	`q.enqueueTask(krnl_vector_add);`
		`kernel(a, b, result, DATA_SIZE);`

1. Power Up



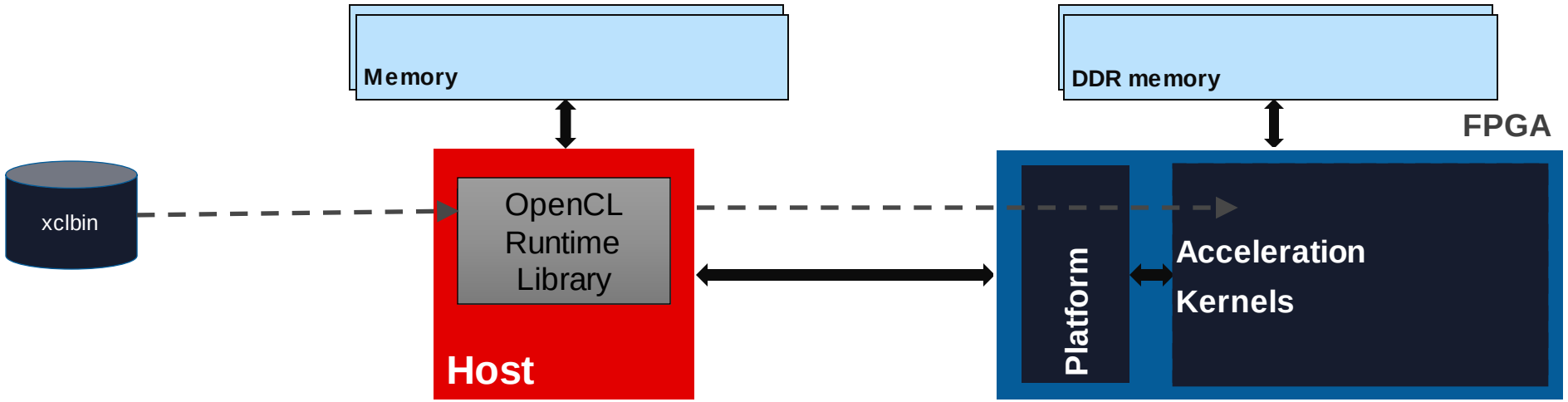
2. Runtime Initialization




```
// Create OpenCL Device (consider it as FPGA)
cl::Device device;

// Search available devices and assign to device.
// ... Omitted code block

// Create OpenCL context (consider it as Vitis Platform)
cl::Context context(device); // context -> Vitis Platform

// Create OpenCL command queue (to send commands to the FPGA)
cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);
```

```
// Create XRT Device
auto device = xrt::device(0);
```

3. Device Configuration

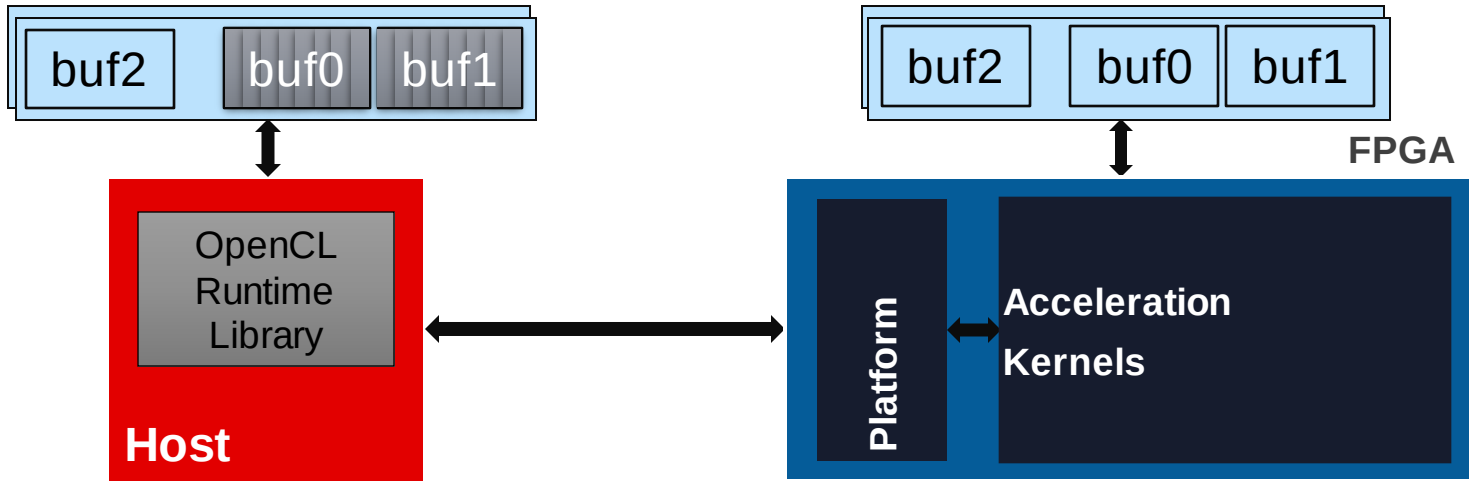



```
cl::Program::Binaries bins;

// Load xclbin to bins
// ... Code procedure omitted

cl::Program program(context, devices, bins);
```

```
auto uuid = device.load_xclbin("binary_container_1.xclbin");

auto kernel = xrt::kernel(device,uuid,"krnl_vadd");
```


XILINX.

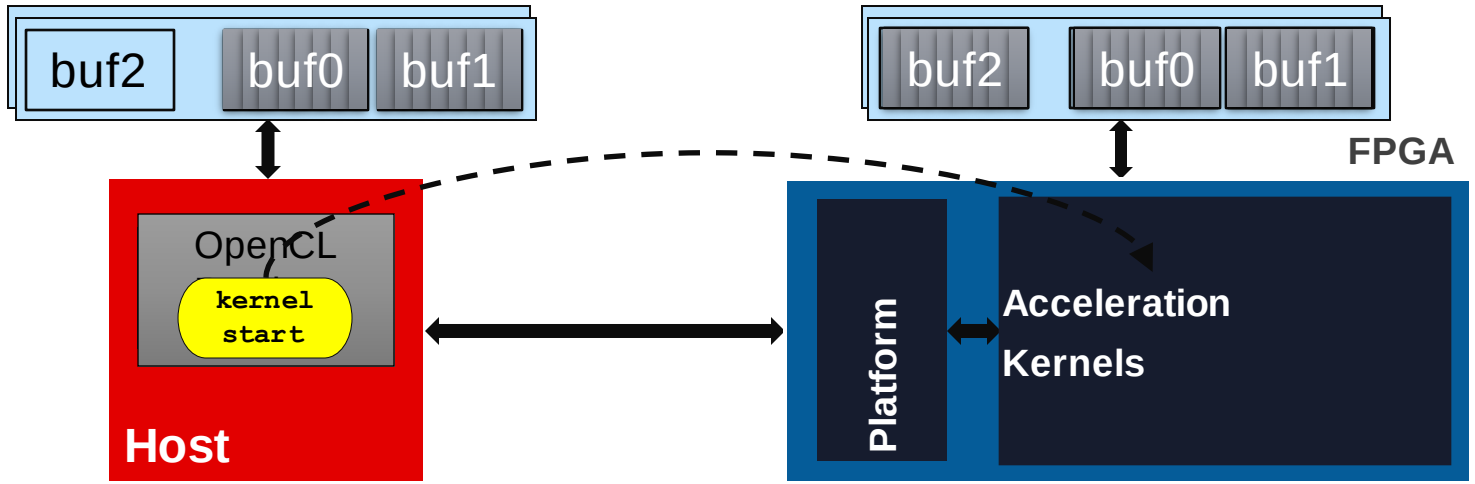
4. Buffer Allocation




```
cl::Buffer buffer_a(context, CL_MEM_READ_ONLY, size_in_bytes);
cl::Buffer buffer_b(context, CL_MEM_READ_ONLY, size_in_bytes);
cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY,
size_in_bytes);
```

```
auto a = xrt::bo(device, size_in_bytes, XCL_BO_FLAGS_NONE,
kernel.group_id(0));
auto b = xrt::bo(device, size_in_bytes, XCL_BO_FLAGS_NONE,
kernel.group_id(1));
auto result = xrt::bo(device, size_in_bytes, XCL_BO_FLAGS_NONE,
kernel.group_id(2));
```

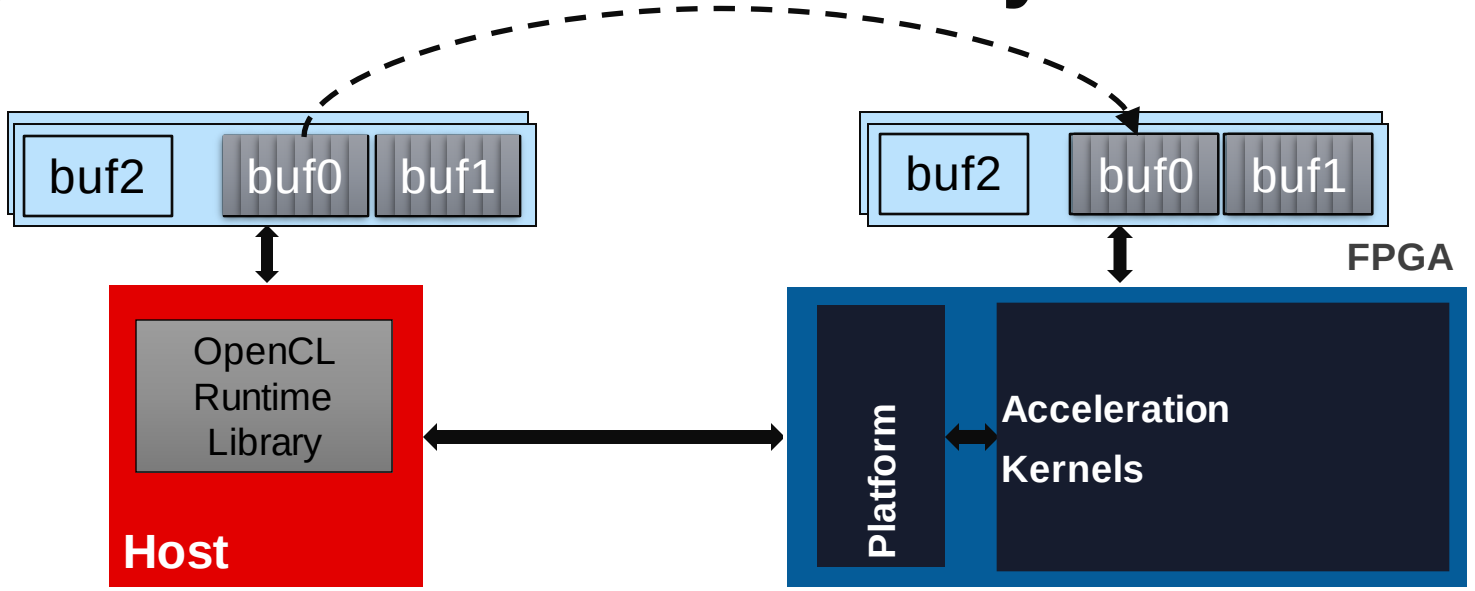
5. Setup Kernel Arguments




```
cl::Kernel krnl_vector_add(program,"krnl_vadd");
krnl_vector_add.setArg(0, buffer_a);
krnl_vector_add.setArg(1, buffer_b);
krnl_vector_add.setArg(2, buffer_result);
krnl_vector_add.setArg(3, DATA_SIZE);
```

```
// Kernel arguments can set with execution commands
```

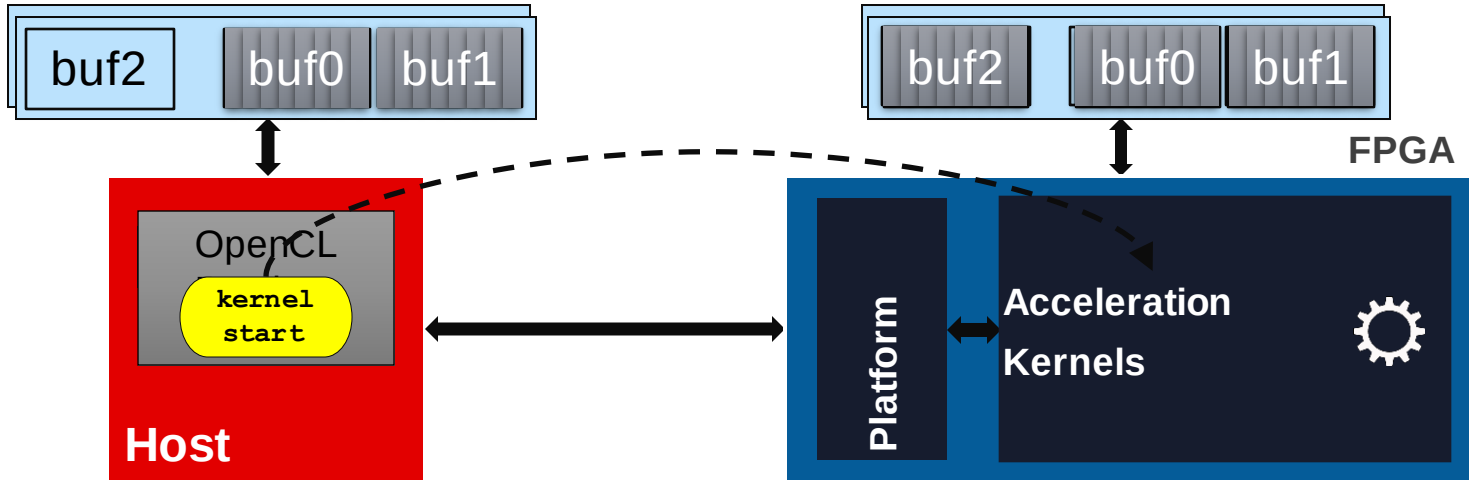
6. Writing Buffers to FPGA Memory




```
q.enqueueMigrateMemObjects({buffer_a,buffer_b},\
                           0/* 0 means from host*/);
```

```
a.sync(XCL_BO_SYNC_BO_TO_DEVICE,size_in_bytes,0);
b.sync(XCL_BO_SYNC_BO_TO_DEVICE,size_in_bytes,0);
```

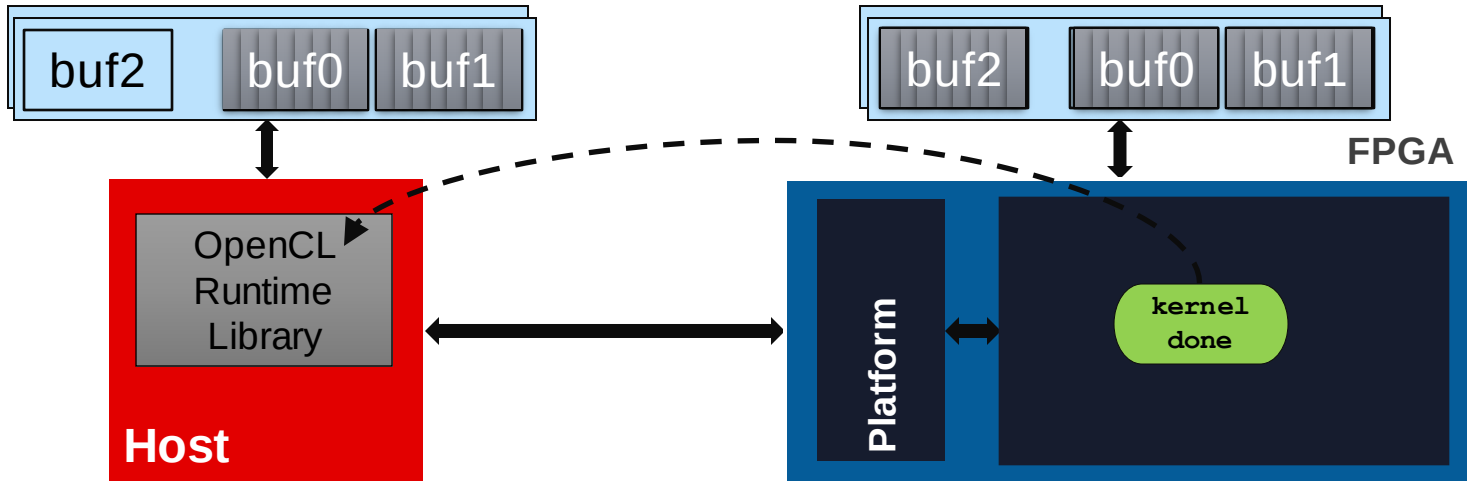
7. Running the Accelerators




```
q.enqueueTask(krnl_vector_add);
```

```
auto r = kernel(a,b,result, DATA_SIZE);
```

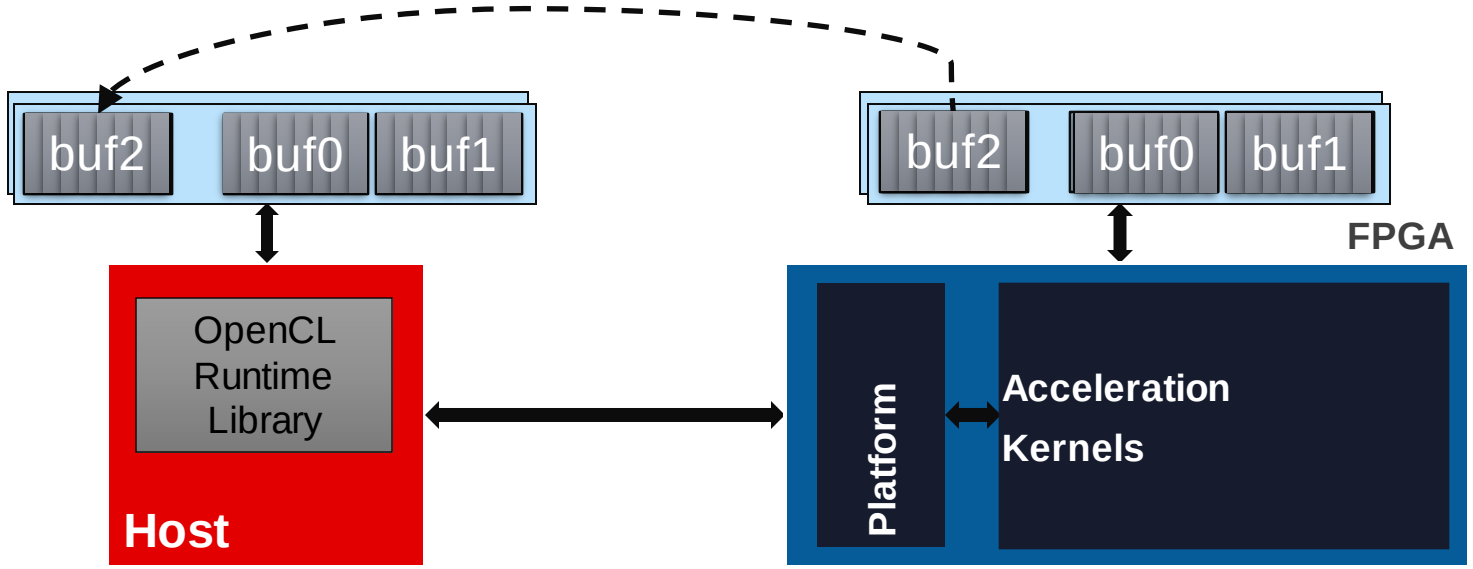
7. Running the Accelerators (Cont'd)




```
// Alternative out-of-order queue events control
cl::Event event_sp;
q.enqueueTask(krnl_vector_add, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

```
r.wait();
```

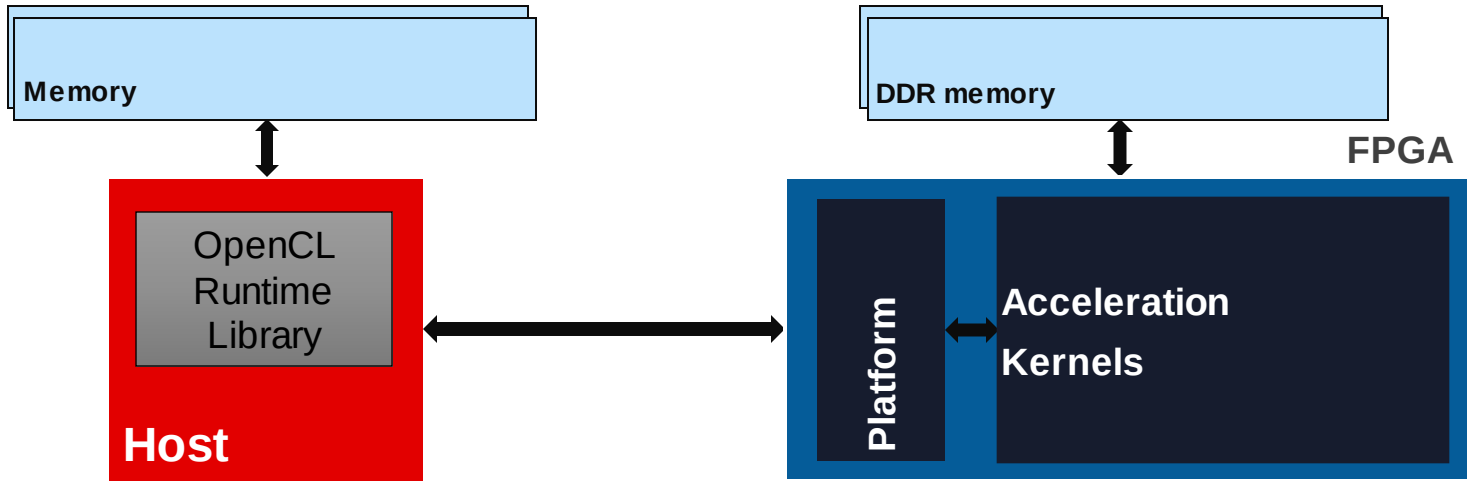
8. Reading Buffers from FPGA Mem




```
q.enqueueMigrateMemObjects({buffer_result},\
                           CL_MIGRATE_MEM_OBJECT_HOST);

// wait for data moving finish
q.finish();
```

```
result.sync(XCL_BO_SYNC_BO_FROM_DEVICE,size_in_bytes,0);
```

9. Clean Up

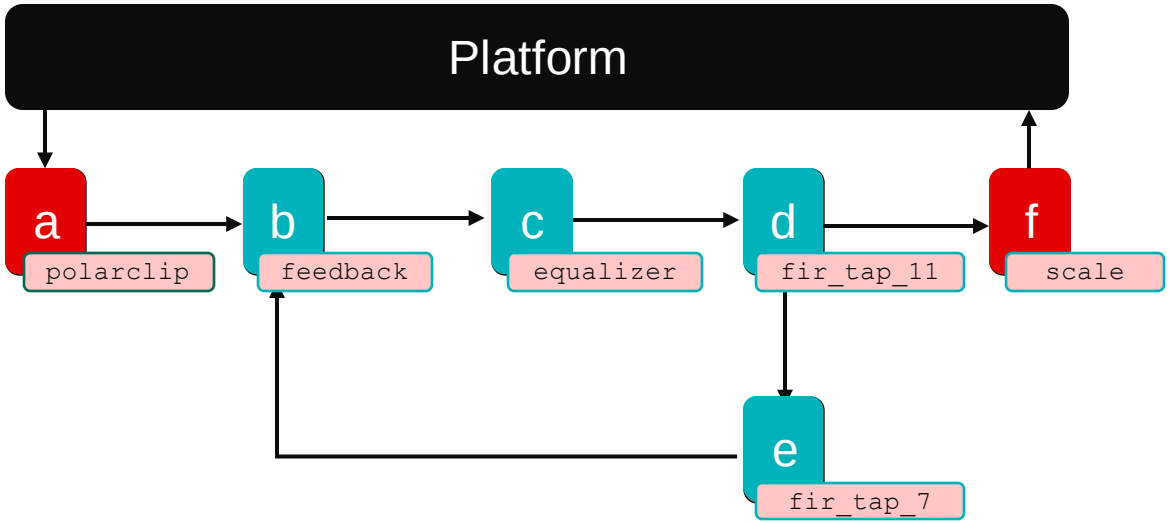



```
q.enqueueUnmapMemObject(buffer_a , ptr_a);
q.enqueueUnmapMemObject(buffer_b , ptr_b);
q.enqueueUnmapMemObject(buffer_result , ptr_result);
q.finish();
```

```
// XRT C++ API does BO deconstruction automatically
```

AI Engine Graph Control

- Opening and Closing
- Resetting
- Execution
- Measure Clock Cycles
- Runtime Parameter (RTP) control
- DMA operation
- Error Reporting




```
xrtGraphOpen(device, xclbin_uuid, "graph_name");

// start from reset state
xrtGraphReset(graphHandle);

// run the graph for 3 iteration
xrtGraphRun(graphHandle, 3);

// Wait till the graph is done
xrtGraphWait(graphHandle,0);
// Use xrtGraphWait if you want to execute the graph again

xrtGraphRun(graphHandle,5);
xrtGraphEnd(graphHandle,0);
// Use xrtGraphEnd if you are done with the graph execution
```

https://xilinx.github.io/XRT/master/html/xrt_native_apis.html

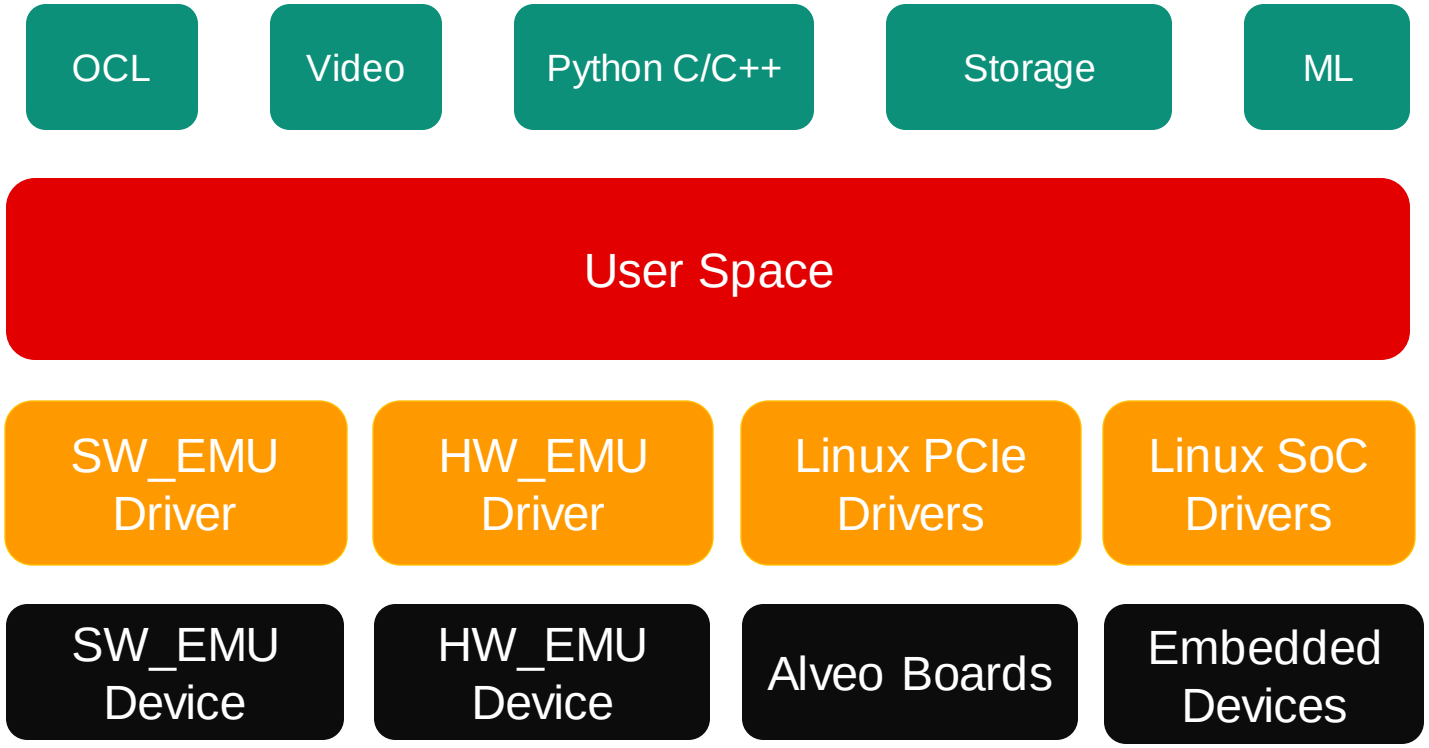
Debugging

- Function Verification
  - SW Emu
- Performance Verification
  - HW Emu
- Profiling
- Read/Write Kernel Reg

```
read_data = kernel.read_register(READ_OFFSET);
kernel.write_register(WRITE_OFFSET,write_data);
```

- Runtime Control Switches
  - xrt.ini

https://xilinx.github.io/XRT/master/html/xrt_ini.html



Board Management

- Management Features *xbmgmt*
  - Program flash
  - Advanced feature configuration

https://xilinx.github.io/XRT/master/html/xbmgmt.html

https://xilinx.github.io/XRT/master/html/xbmgmt2.html

- Utilities *xbutil*
  - Scan Alveo cards on the system
  - Report platform version and XRT version
  - Query kernel info on board
  - Program xclbin
  - Reset board
  - Validate the Alveo board

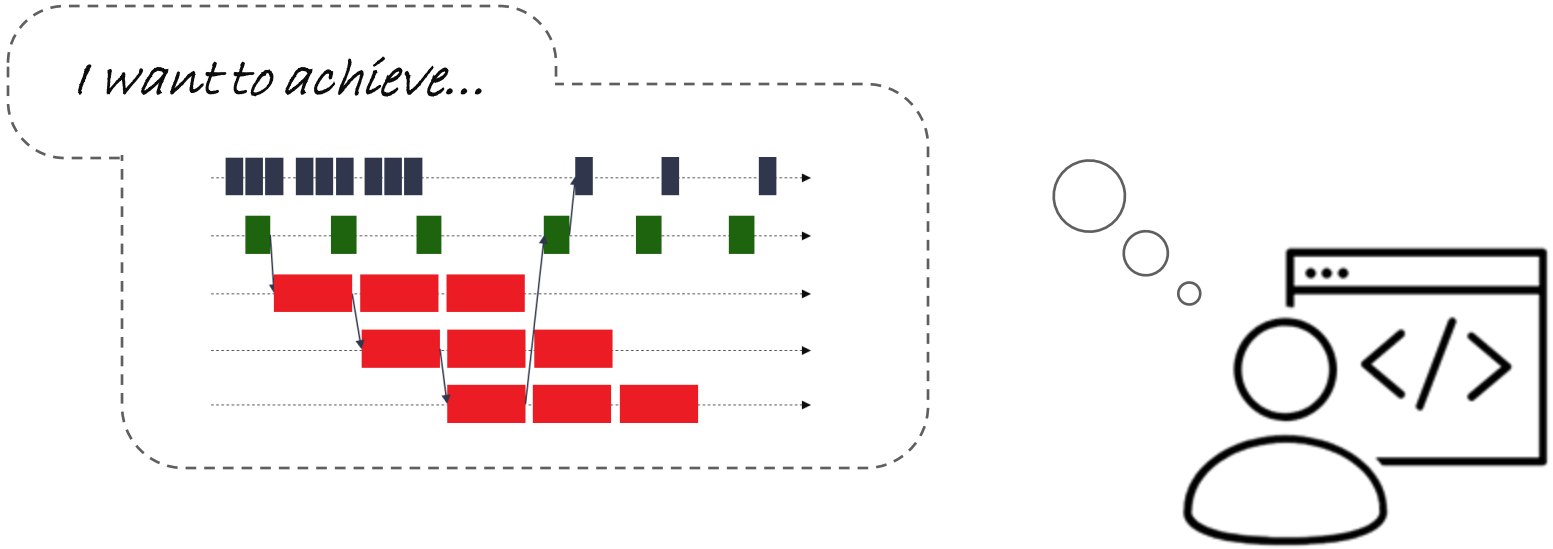
https://xilinx.github.io/XRT/master/html/xbutil.html

https://xilinx.github.io/XRT/master/html/xbutil2.html

Section 2
Performance Optimization

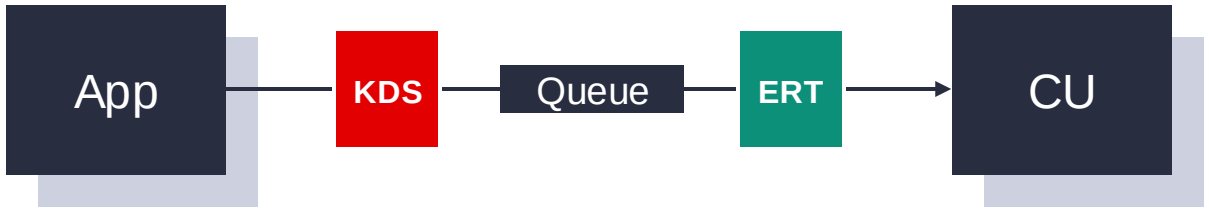
Conceptualize the Desired Application Timeline

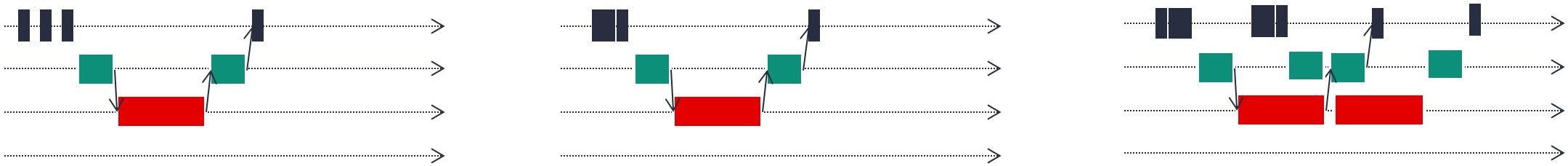
- Keep the kernels active, performing new computations as often as possible
- Optimize data transfers to and from the FPGA



Aggregate AXI Interactions

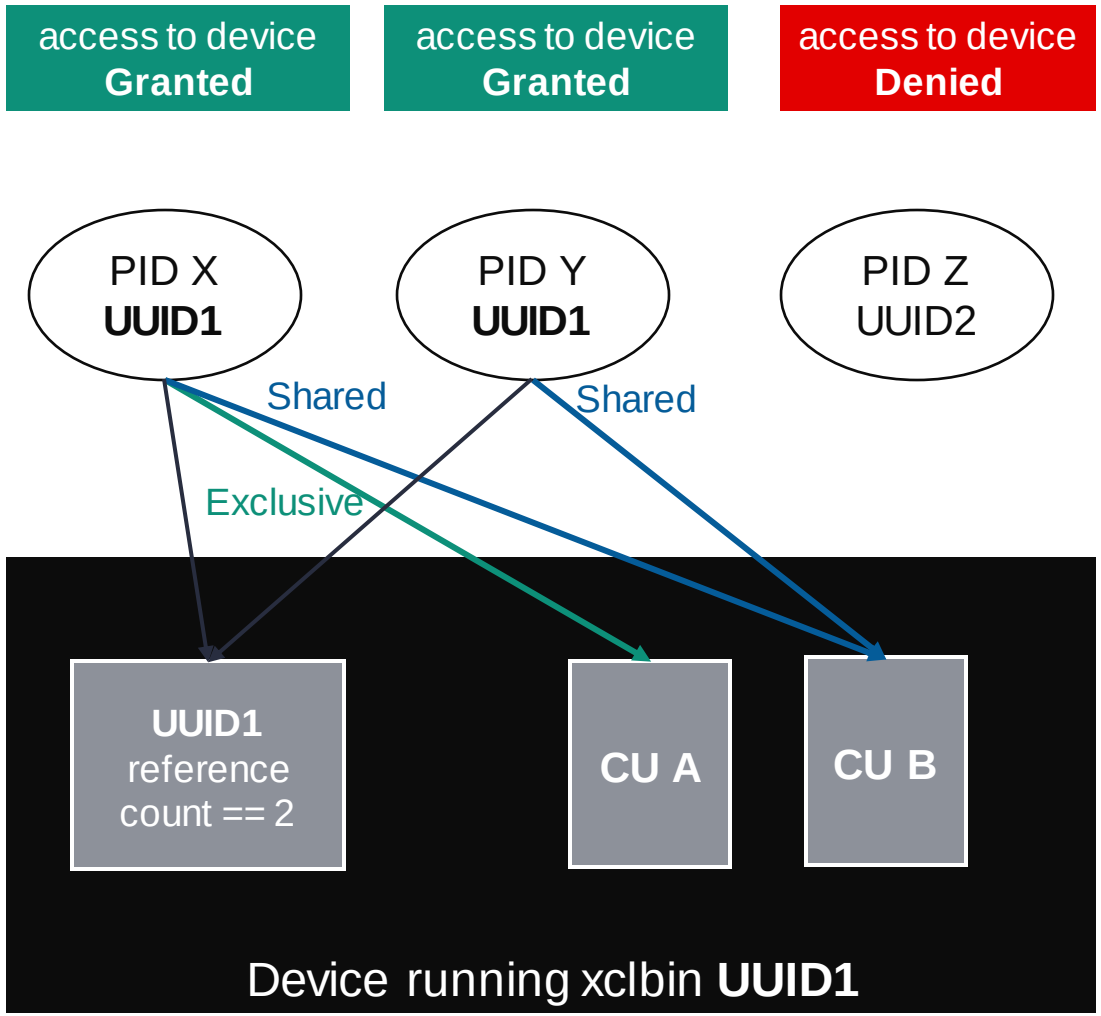
ERT for Async Execution





https://xilinx.github.io/XRT/master/html/execution-model.html

Multi-Processing




```
// XRT API
auto krnl = xrt::kernel(device, xclbin_uuid, name, \
            xrt::kernel::cu_access_mode::exclusive);

// xrt::kernel::cu_access_mode::shared
```

- Framework for multiple processes to time division multiplex a compute unit on a device concurrently
- Support for exclusive or shared context per compute unit
- Transparent locking of xclbins

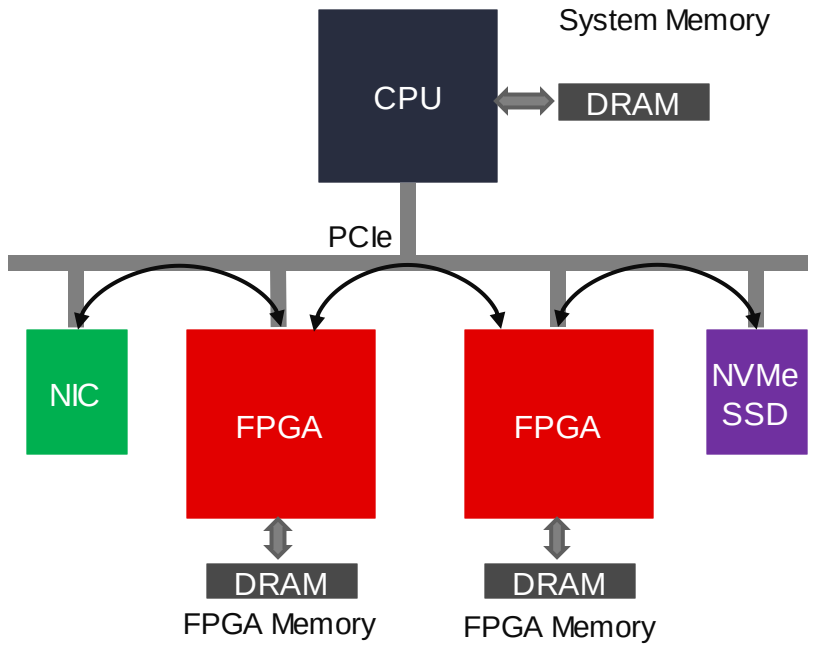
https://xilinx.github.io/XRT/master/html/multiprocess.html

Can FPGA Get/Send Data wo/ Going Through CPU?

Alveo Only

PCIe Peer-to-Peer (P2P)

- Direct data transfer between FPGAs or other PCIe devices like NVMe SSDs without using the host CPU memory

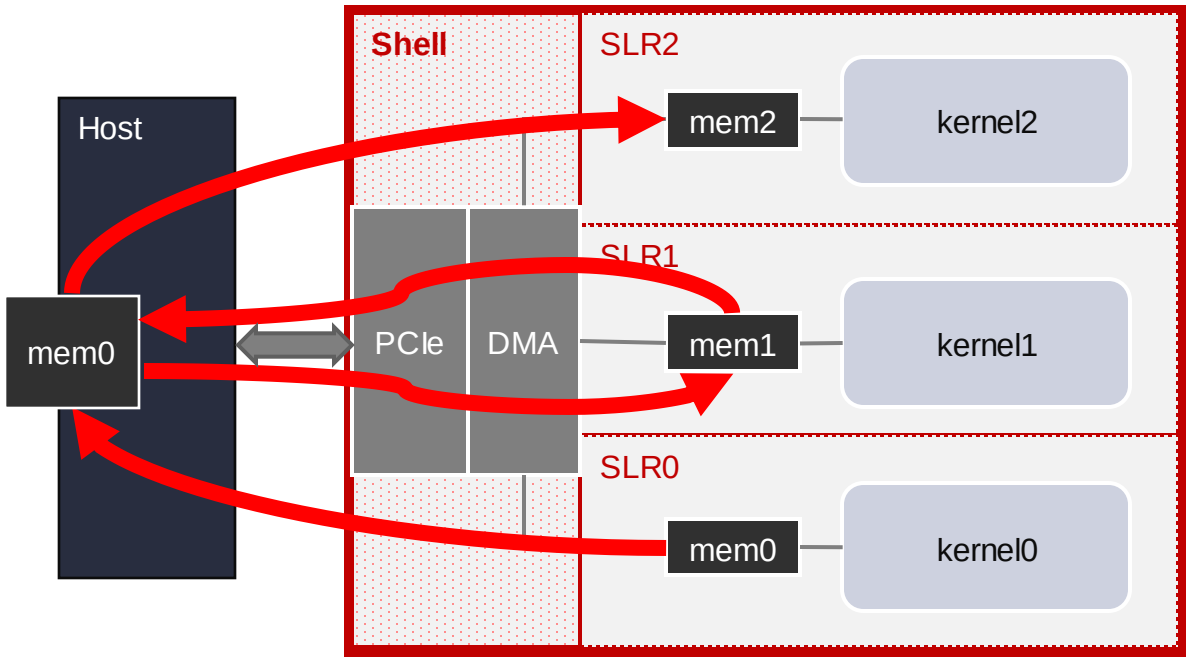


- P2P Simple -- Simple P2P Read/Write operations
  - https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/host/p2p_simple
- P2P Bandwidth -- Throughput data transfers
  - https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/host/p2p_bandwidth
- P2P FPGA
  - https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/host/p2p_fpga2fpga

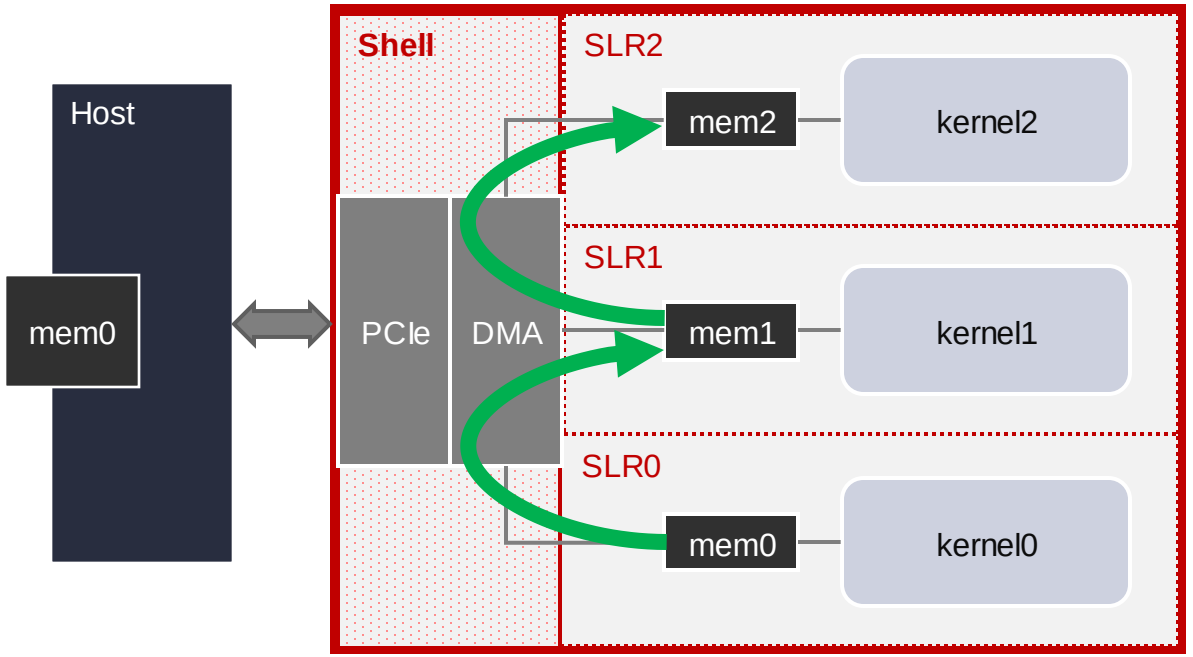
https://xilinx.github.io/XRT/master/html/p2p.html

FPGA Memory-to-Memory (M2M)

No M2M



M2M Enabled



- Efficient data transfer between kernels
  - Move data from local FPGA memories (DDR/PLRAM)
  - Low latency data transfer implemented on FPGA
  - Improves host memory and PCIe bandwidth efficiency

https://xilinx.github.io/XRT/master/html/m2m.html

OpenCL API

```
clEnqueueTask(queue,K1,0,nullptr,&events1);
clEnqueueCopyBuffer(queue,Buf2,Buf3,0,0,\
                    Buffer_size,1,&event1,&event2);
clEnqueueTask(queue,K2,1,&event2,nullptr);
```

XRT API

```
dst_buffer.copy(src_buffer, copy_size_in_bytes);
```

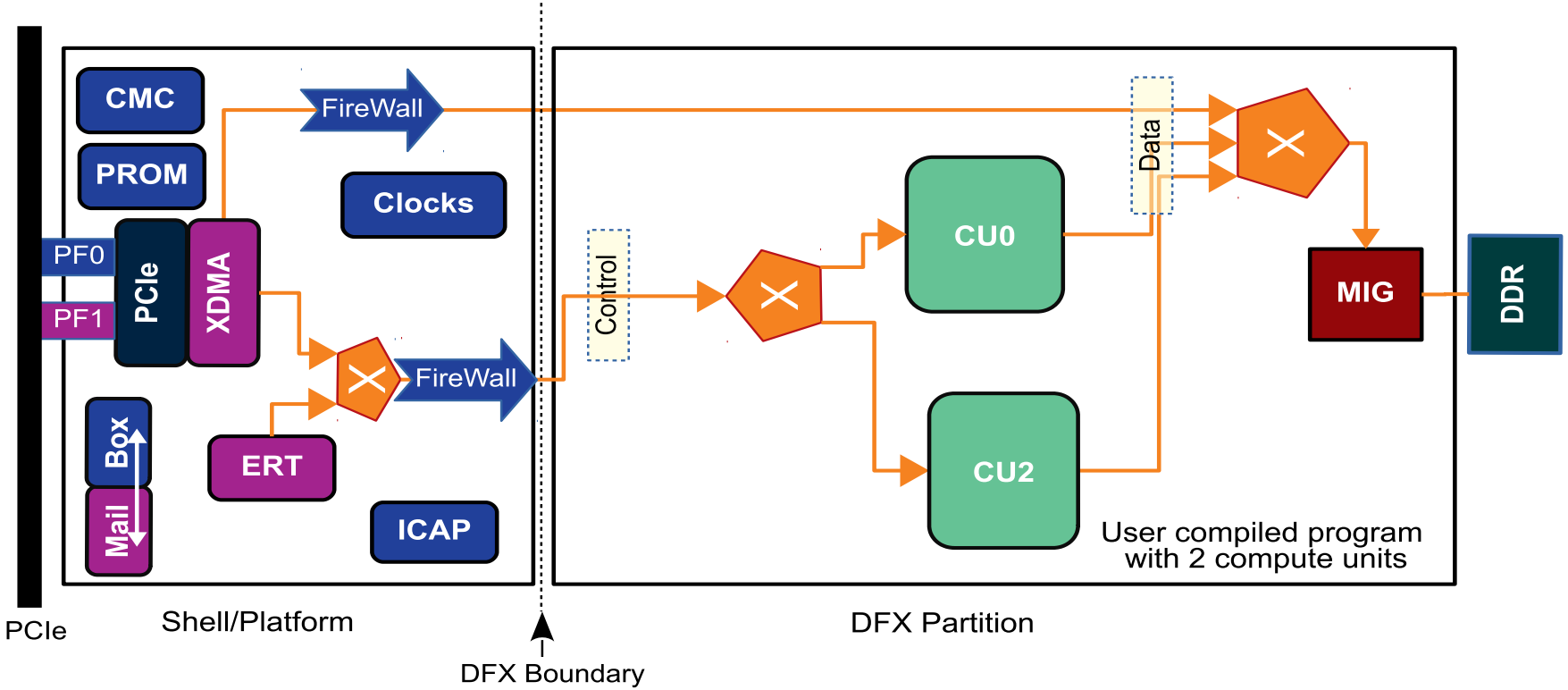

XILINX.

Section 3
Advanced Feature

How to Ensure Stability?

Firewall

Alveo Only



Alveo Architecture

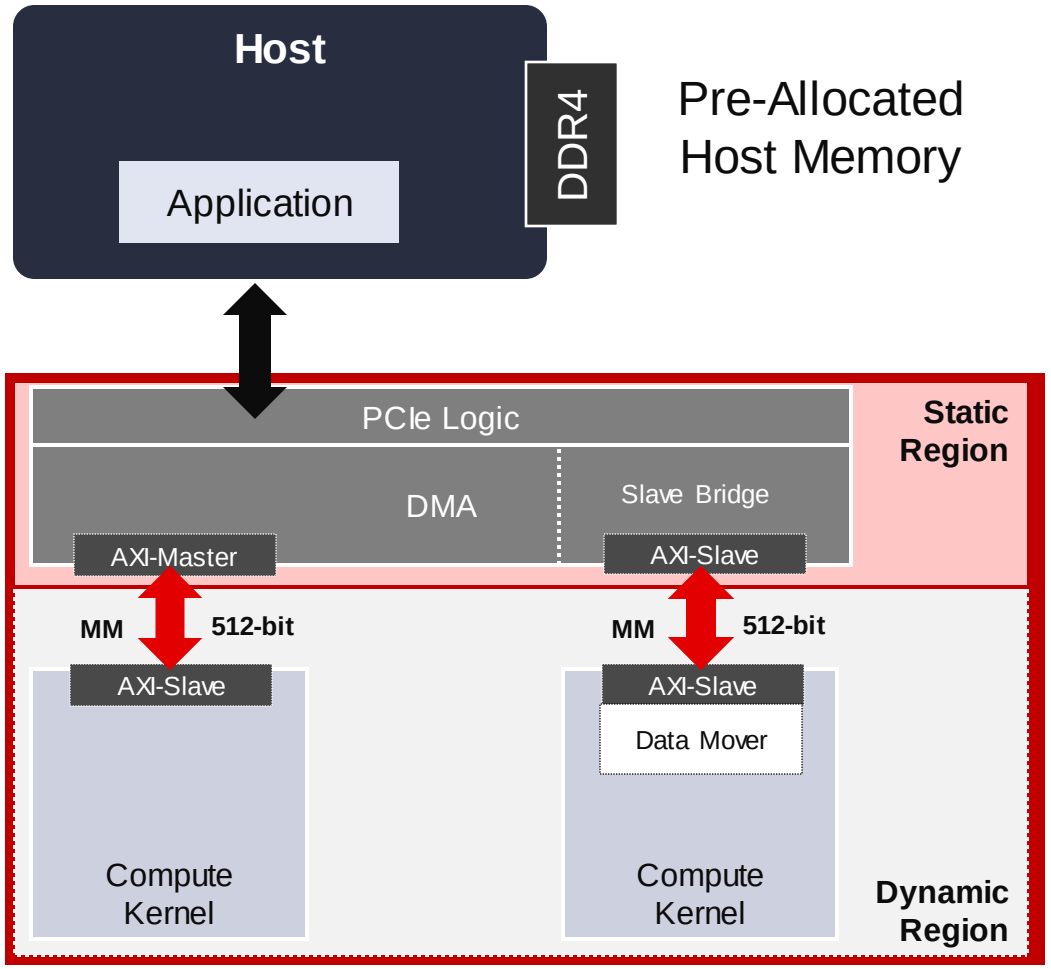
https://xilinx.github.io/XRT/master/html/security.html

XILINX.

Can Kernel Access Host Memory?

Slave Bridge

Alveo Only



- **Direct host memory access by the kernel**
  - Requires pre-allocated host memory

```
xbutil host_mem -d 0 --enable --size 1G
xbutil host_mem -d 1 --enable --size 4G
xbutil host_mem -d 2 --enable --size 16G
```

  - Host code requires a special buffer declaration

```
cl_mem_ext_ptr_t host_buffer_ext;
host_buffer_ext.flags = XCL_MEM_EXT_HOST_ONLY;
host_buffer_ext.obj = NULL;
host_buffer_ext.param = 0;

cl::Buffer buffer_in (context, (cl_mem_flags)(CL_MEM_READ_ONLY | CL_MEM_EXT_PTR_XILINX),
size_in_bytes, &host_buffer_ext);
cl::Buffer buffer_out(context, (cl_mem_flags)(CL_MEM_WRITE_ONLY| CL_MEM_EXT_PTR_XILINX),
size_in_bytes, &host_buffer_ext);
```

- **DMA bypass capability**
  - AXI-Slave 512-bit interface
  - User provides their data mover

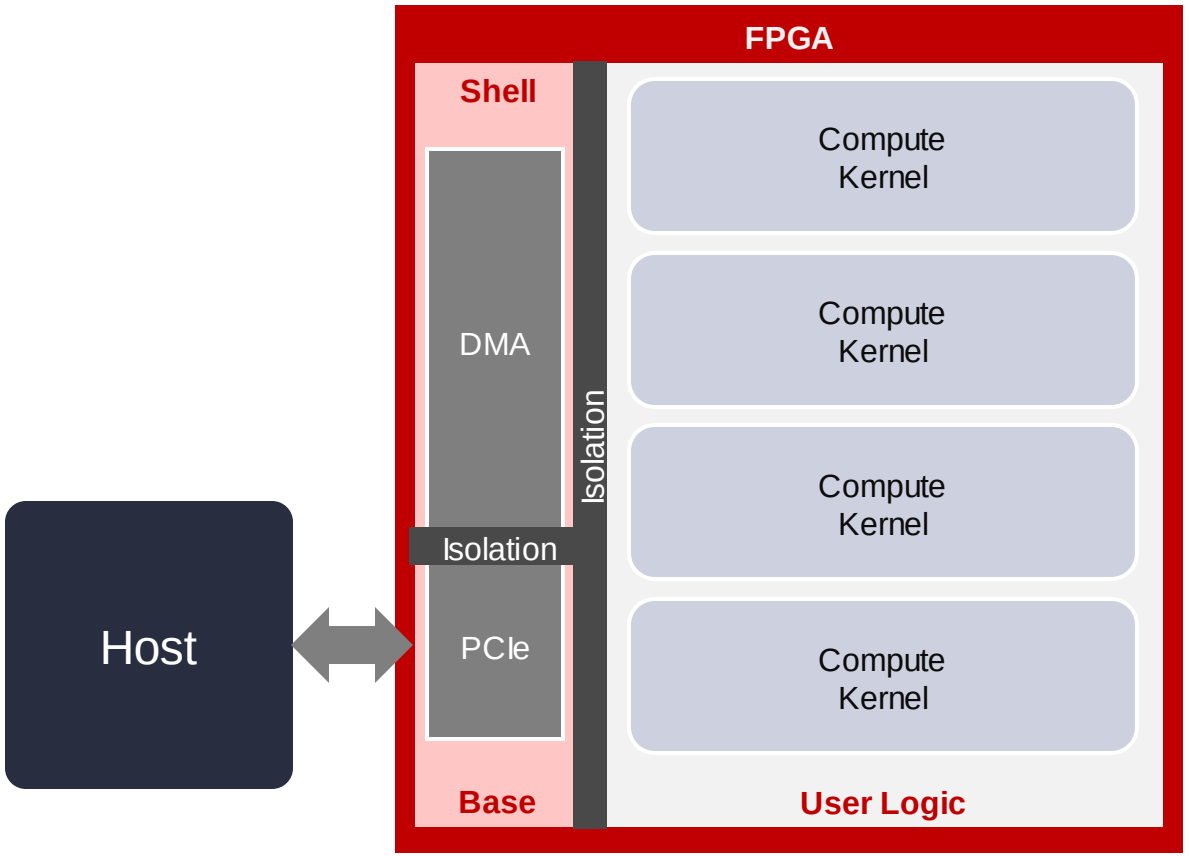
https://xilinx.github.io/XRT/master/html/sb.html

XILINX.

Can I Hot Update the DMA without Rebooting?

Dynamic Function eXchange w/ 2 Reconfiguration Partitions (DFX-2RP)

Alveo Only



- **Base**
  - PCIe functionality locked down
  - In-band FPGA partial reconfiguration
- **Shell**
  - Update DMA and utility functions w/o reboot
  - Potentially variable sized regions
- **User Logic**
  - Accelerator kernel functions
  - Applications loaded by Host at runtime

XILINX.

More on XRT

- Virtualization
  - Docker support
  - K8S support

https://github.com/Xilinx/FPGA_as_a_Service

- FaaS Support
  - MPD MSD

https://xilinx.github.io/XRT/master/html/cloud_vendor_support.html

- Self Test
  - The xbtest application performs a set of stress tests on Alveo cards.

https://www.xilinx.com/support/answers/75656.html

- Open-Source Porting Efforts
- Vitis Analyzer
- Vitis Accel Examples
- Vitis Methodology
  - UG1393
  - Vitis-In-Depth-Tutorial

Summary

- Use OpenCL API or XRT API for Alveo and SoC Heterogenous Acceleration Programming
- Achieve performance with advanced XRT and platform features and programming methodology.
- Check XRT references for more in-depth learning.

https://www.xilinx.com/xrt

Thank You