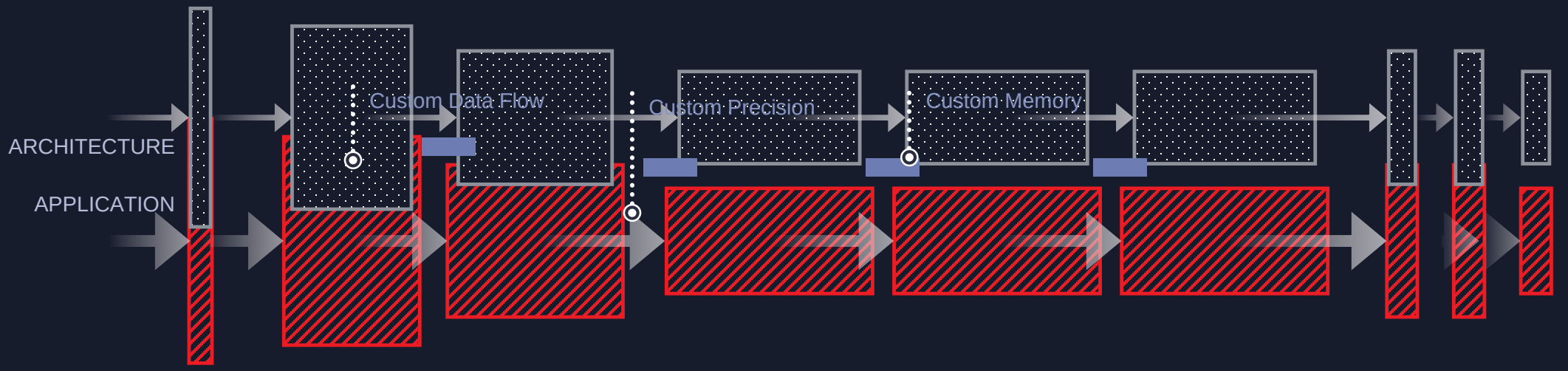
Bring Your Applications to Life with Vitis

Rob Armstrong
Director, Software Acceleration and AI Technical Marketing
19 November 2019

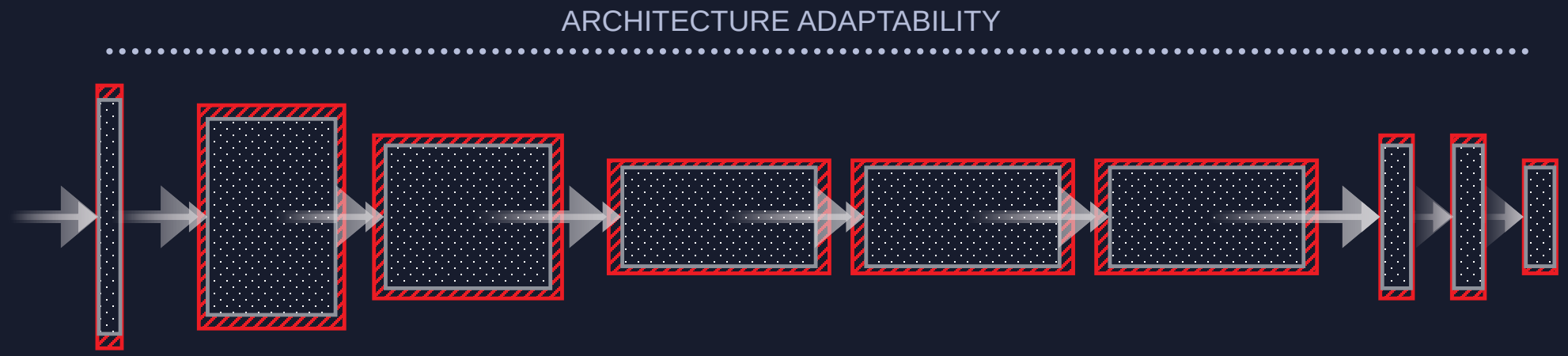
XILINX.

Domain Specific Architecture

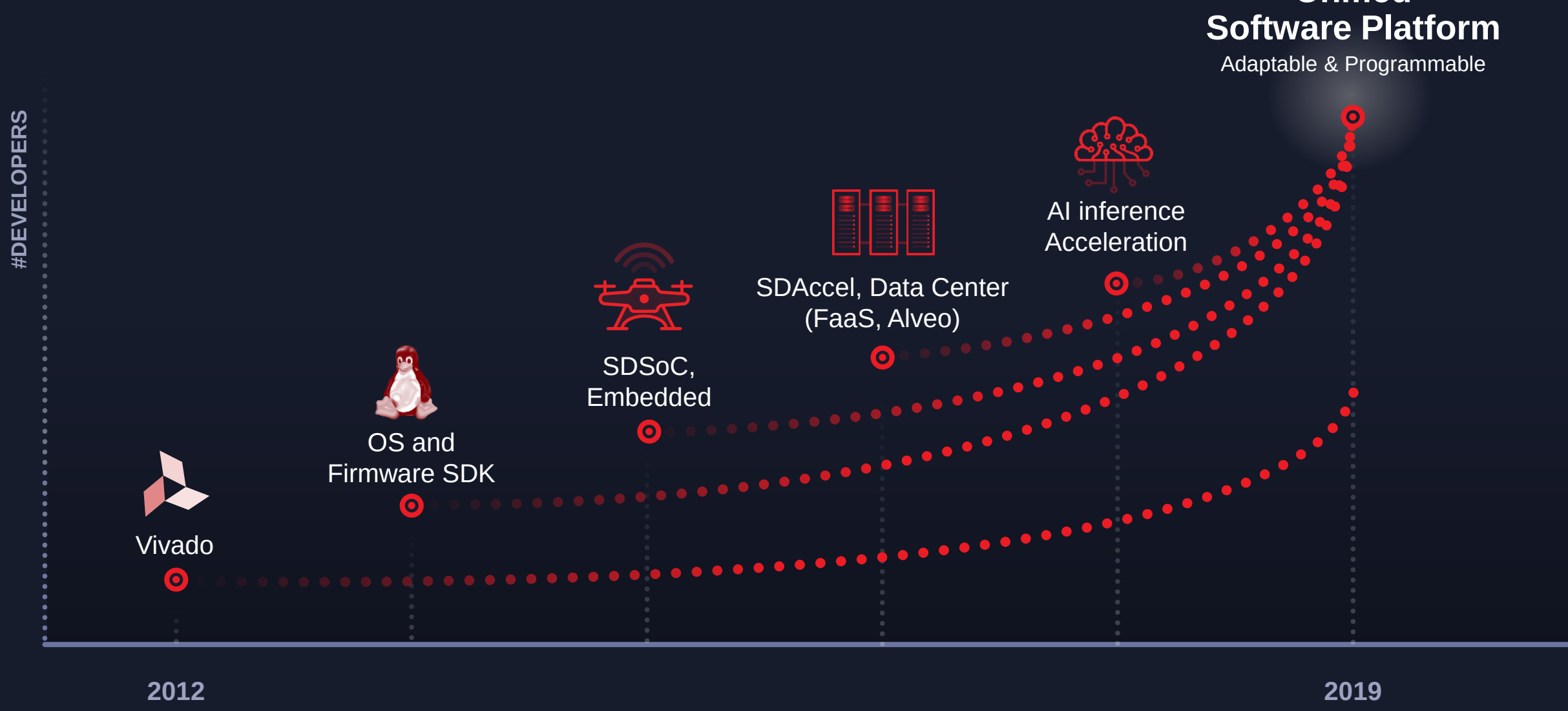


XILINX.

Domain Specific Architecture



Platform Transformation



Introducing Vitis, Unified Software Platform

XILINX

XILINX VITIS™

Unified Software Platform

Software & AI

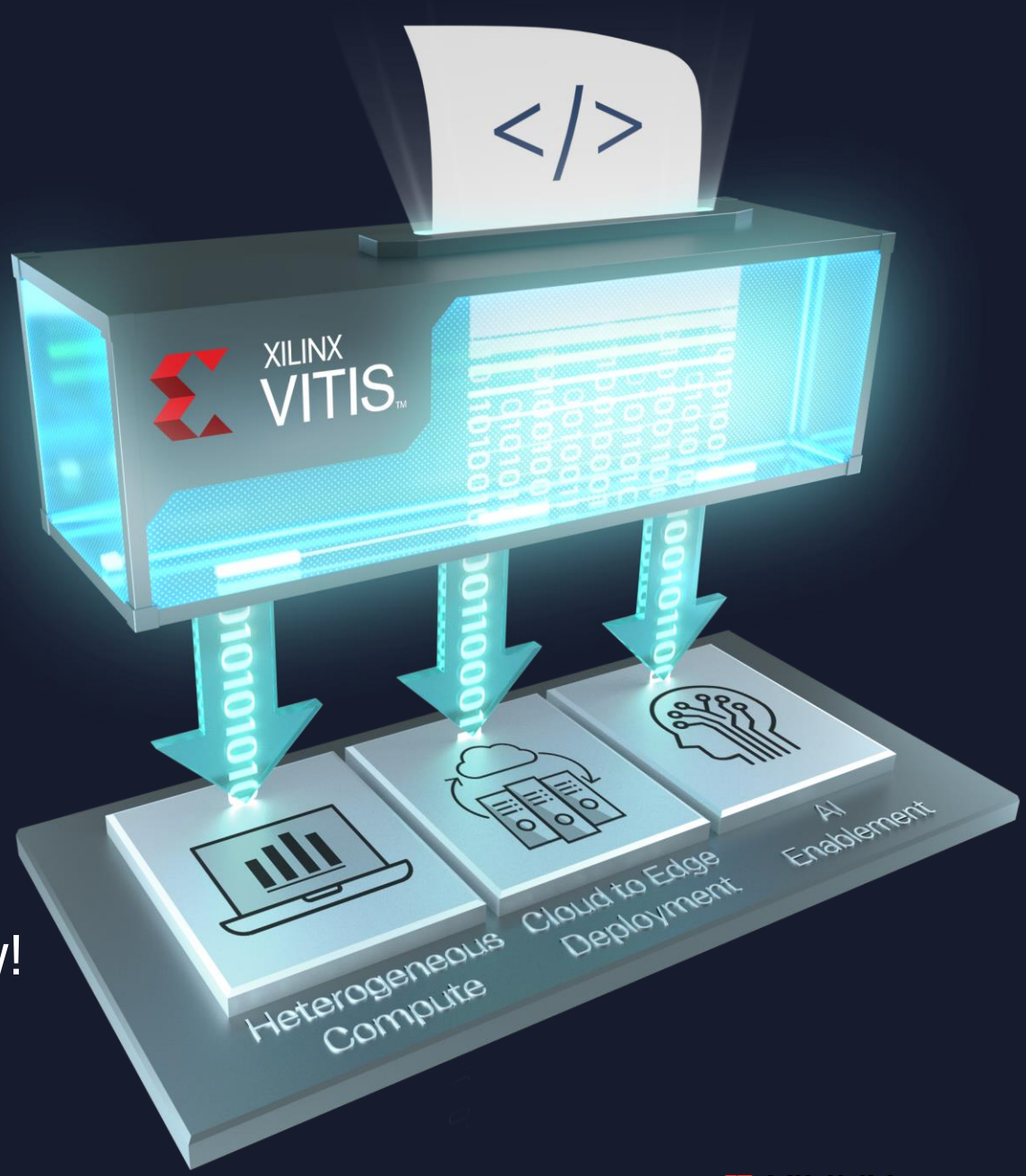
Adaptive computing

Edge to Cloud

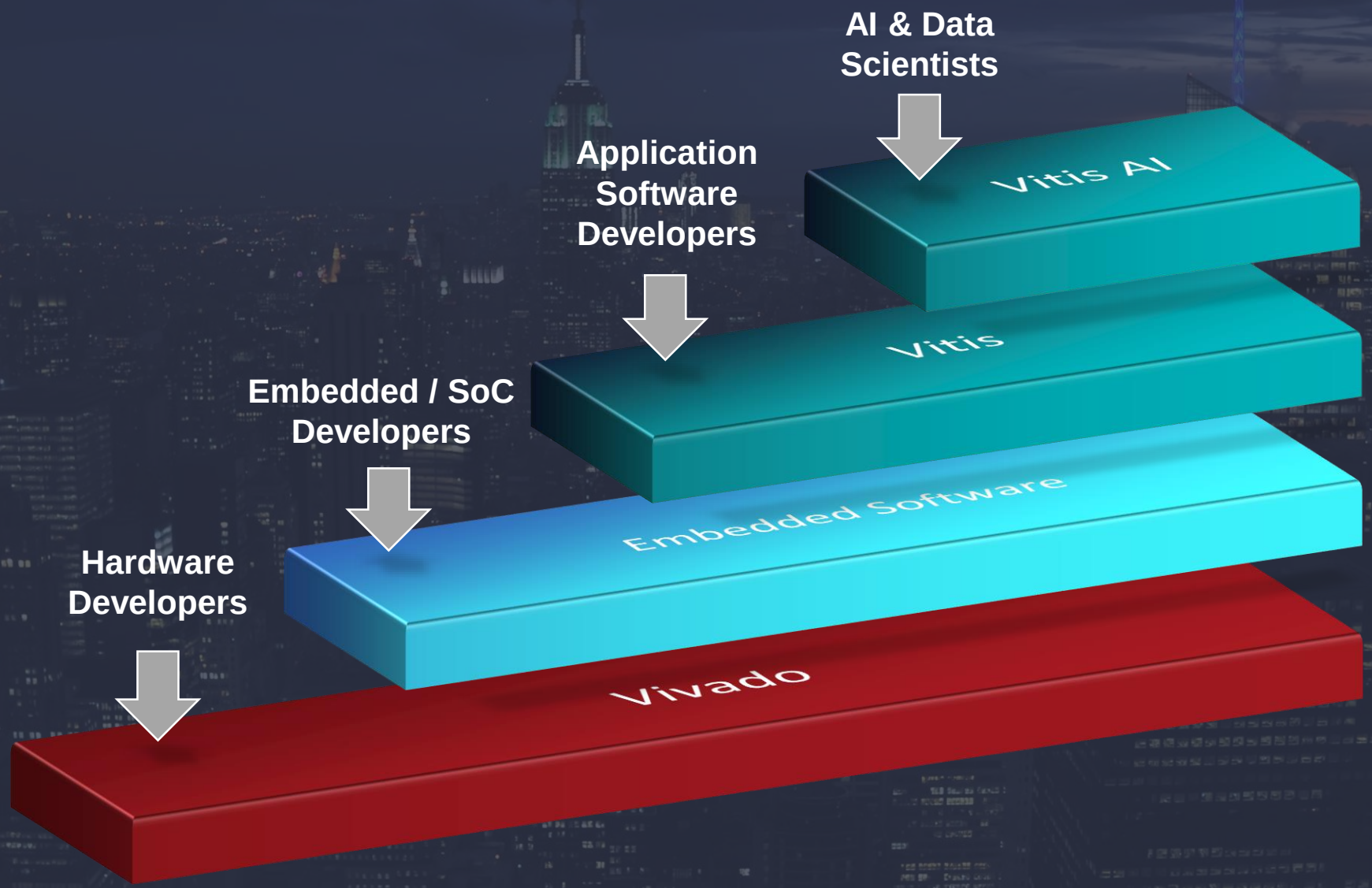
Free!

Available now!

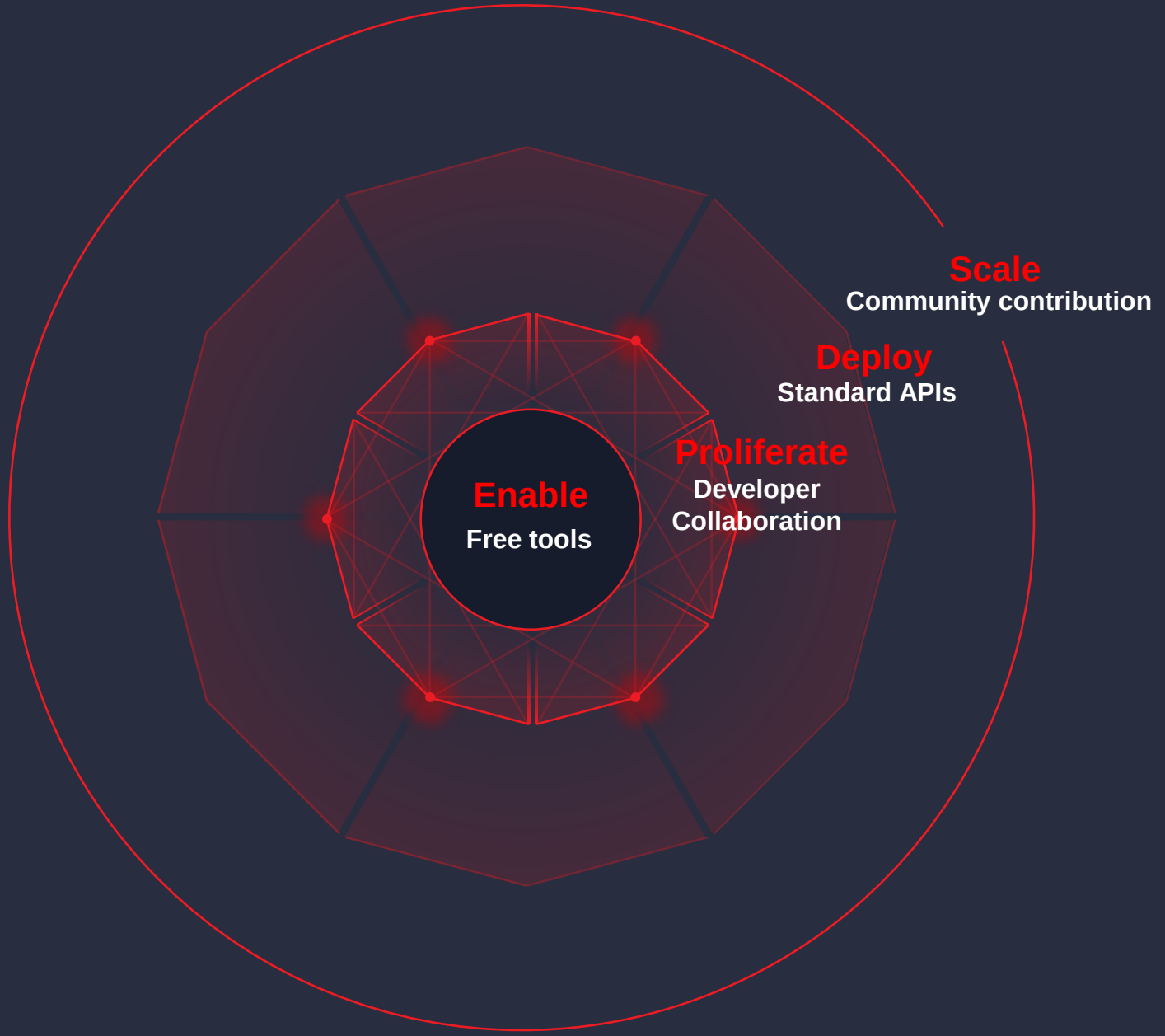
Standards based



Development Platforms for ALL Developers



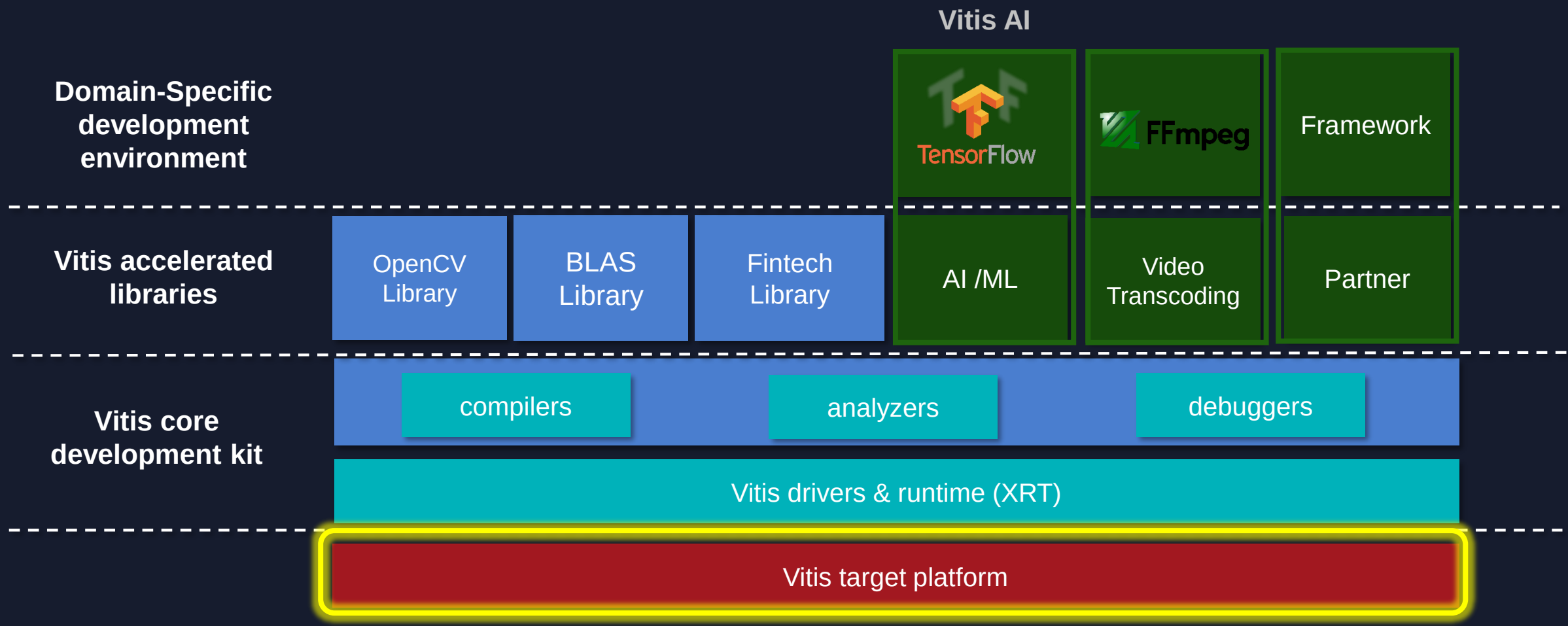
OPEN SOURCE


Scale
Community contribution
Deploy
Standard APIs
Proliferate
Developer
Collaboration
Enable
Free tools

Open Source

			Today	Future
 Communities	**Scale**	Available	Vitis AI Open Source Compression Algorithms | Quantization Format | Pytorch & TF Plugins	Kubernetes & Docker Plugins
 Libraries	**Deploy**	Standard	Vitis Libraries Video | Fintech | AI Models | Math | BLAS | OpenCV	Integration with Upstream Projects
 Runtime & System SW	**Proliferate**	Open	Vitis Runtime Open Source OpenAMP Linaro Foundation	Vitis Compiler Frontends
 Libraries	**Enable**	Free	Vitis Tools Free Vivado Images in Clouds	Select Vivado modules as APT packages

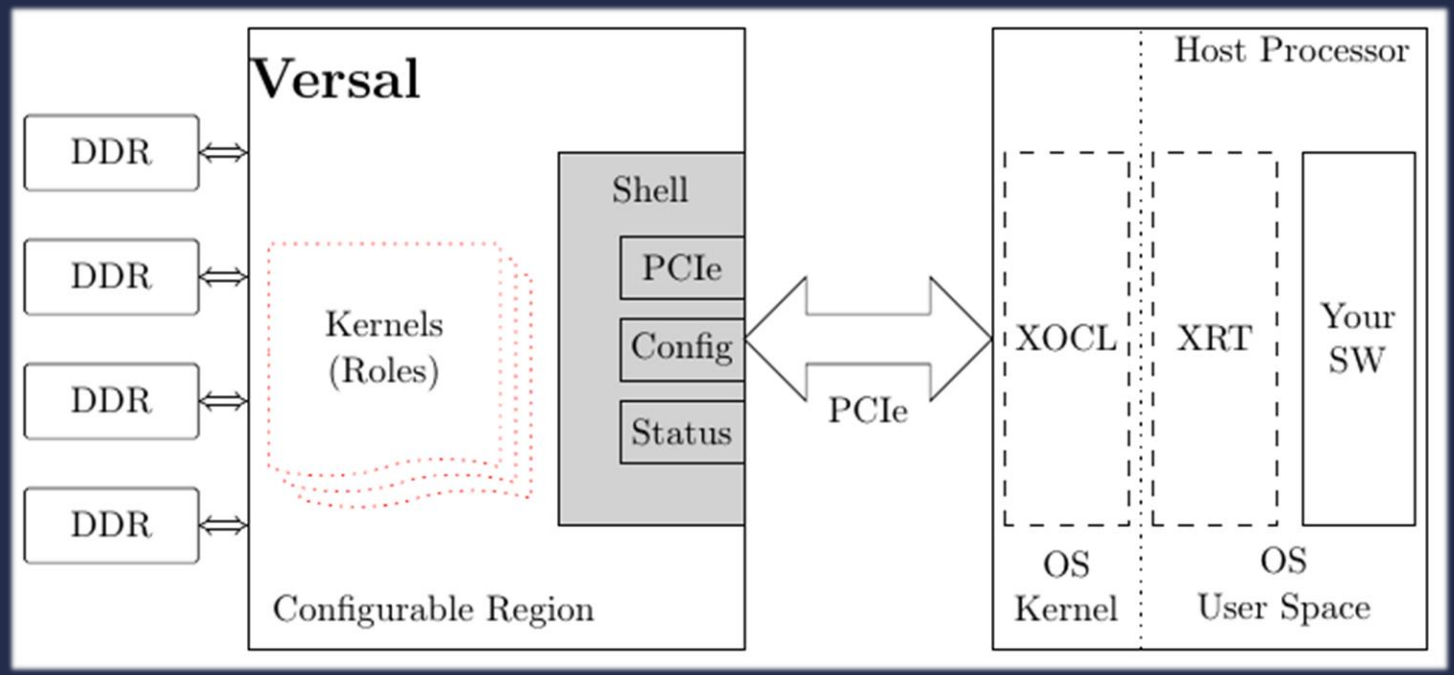
Vitis: Unified Software Platform



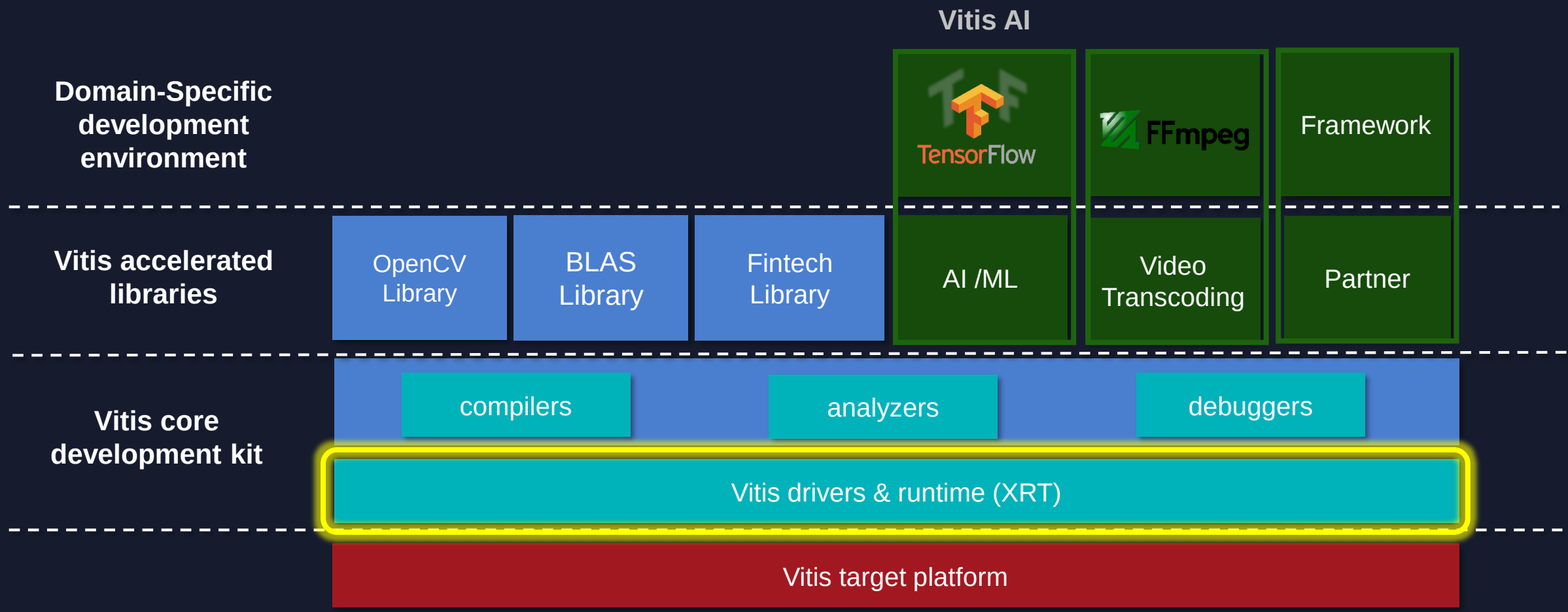
XILINX

Shell-Based Development Architecture

- Acceleration is performed in the context of a *platform*
  - A pre-configured system containing I/O, status monitoring, and lifecycle management functionality

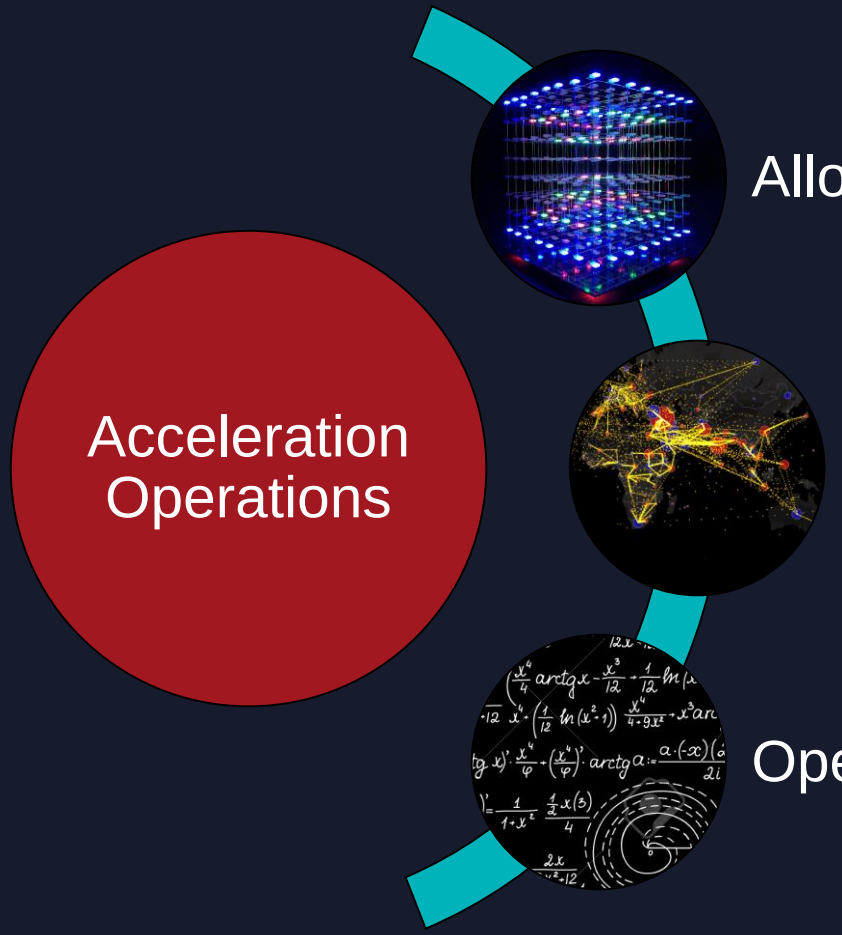


Vitis: Unified Software Platform



XILINX

Three Fundamental Operations

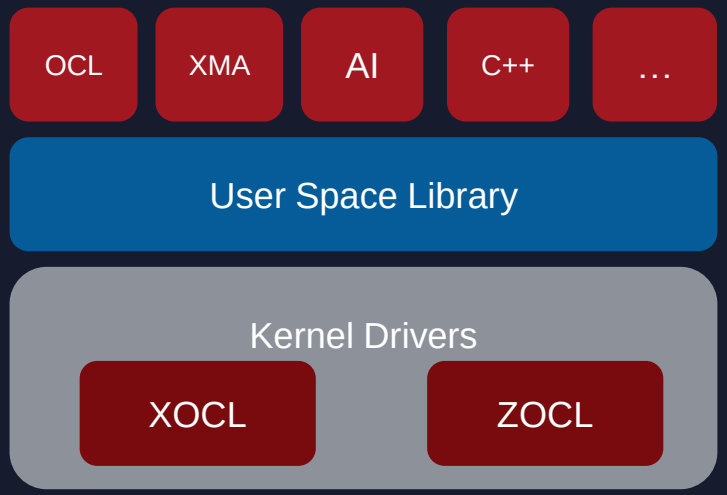



```
sort(order.begin(), order.end(), [](const std::pair<int, float> &ls, const std::pair<int, float> &rs) {
    return ls.second > rs.second;
});

std::vector<int> keep;
std::vector<bool> exist_box(count, true);
for (size_t _i = 0; _i < count; ++_i) {
    size_t i = order[_i].first;
    float x1, y1, x2, y2, w, h, iarea, jarea, inter, ovr;
    if (!exist_box[i])
        continue;
    keep.push_back(i);
    for (size_t _j = _i + 1; _j < count; ++_j) {
        size_t j = order[_j].first;
        if (!exist_box[j])
            continue;
        x1    = std::max(box[i][0], box[j][0]);
        y1    = std::max(box[i][1], box[j][1]);
        x2    = std::min(box[i][2], box[j][2]);
        y2    = std::min(box[i][3], box[j][3]);
        w     = std::max(float(0.0), x2 - x1 + 1);
        h     = std::max(float(0.0), y2 - y1 + 1);
        iarea = (box[i][2] - box[i][0] + 1) * (box[i][3] - box[i][1] + 1);
        jarea = (box[j][2] - box[j][0] + 1) * (box[j][3] - box[j][1] + 1);
        inter = w * h;
        ovr   = inter / (iarea + jarea - inter);
        if (ovr >= nms)
            exist_box[j] = false;
    }
}

std::vector<std::vector<float>> result;
result.reserve(keep.size());
```

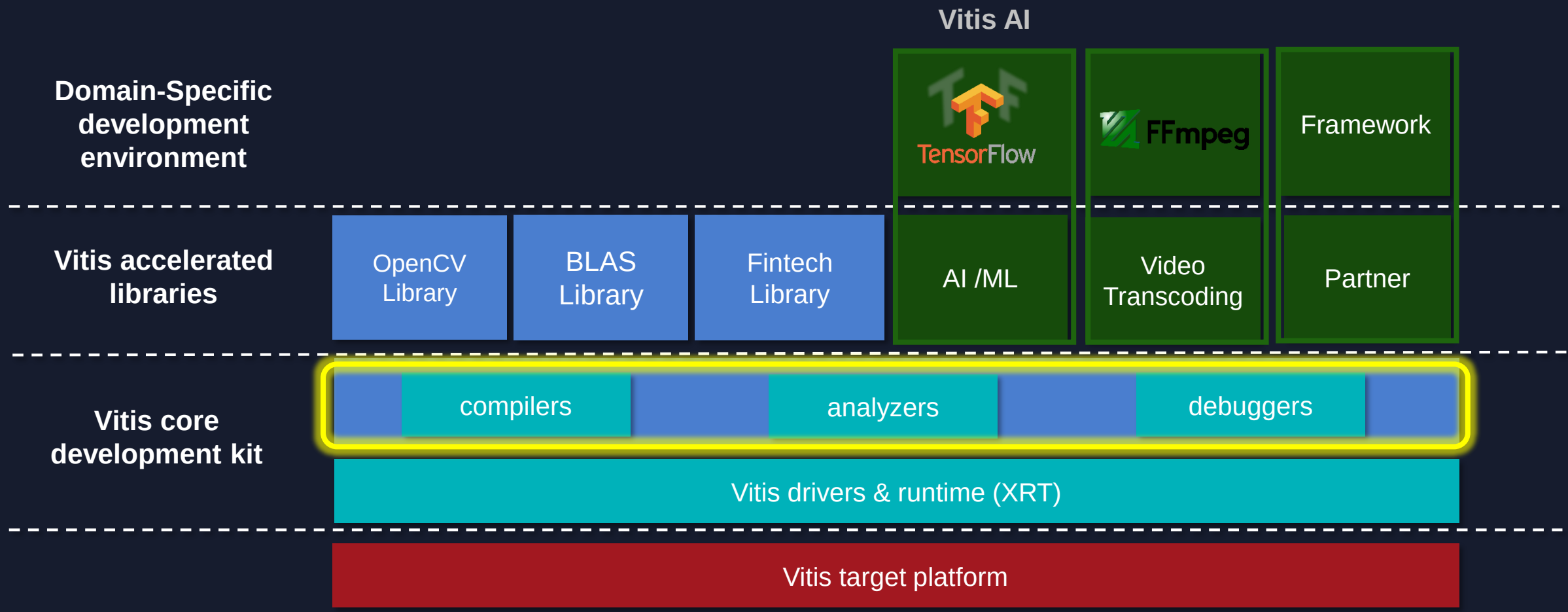
XILINX

The Xilinx Runtime (XRT)

- XRT is a combination of a runtime software layer and kernel driver allowing software to dynamically interact with FPGA and ACAP-based accelerators
- XRT has been in use for several years targeting PCIe-based accelerators such as the Alveo boards
  - Has now being extended to support shared-memory embedded systems
- On top of the low-level APIs and drivers, Xilinx has built OpenCL API wrappers, media frameworks, and domain-specific acceleration APIs
- These libraries and APIs interoperate with higher-level software frameworks and libraries

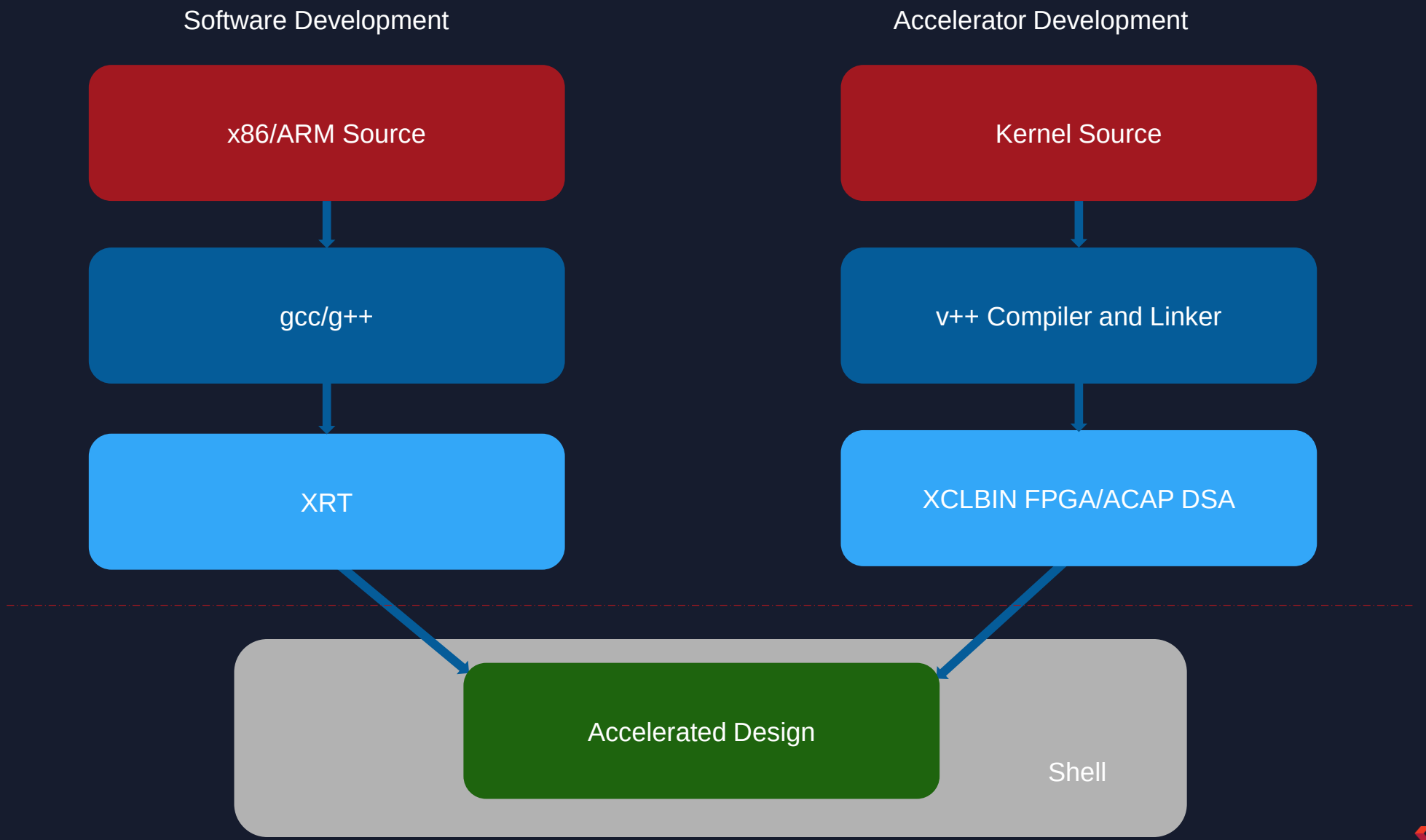


Vitis: Unified Software Platform



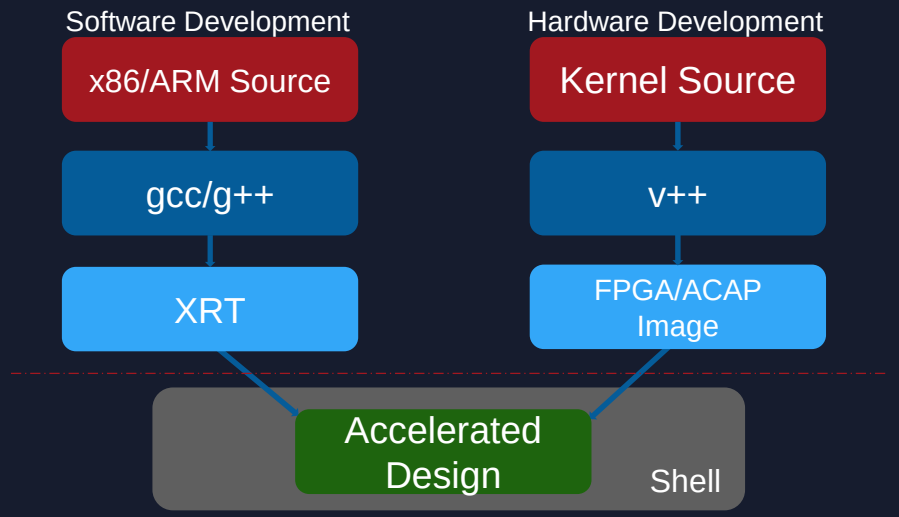
XILINX

Vitis: Accelerated Application Flow

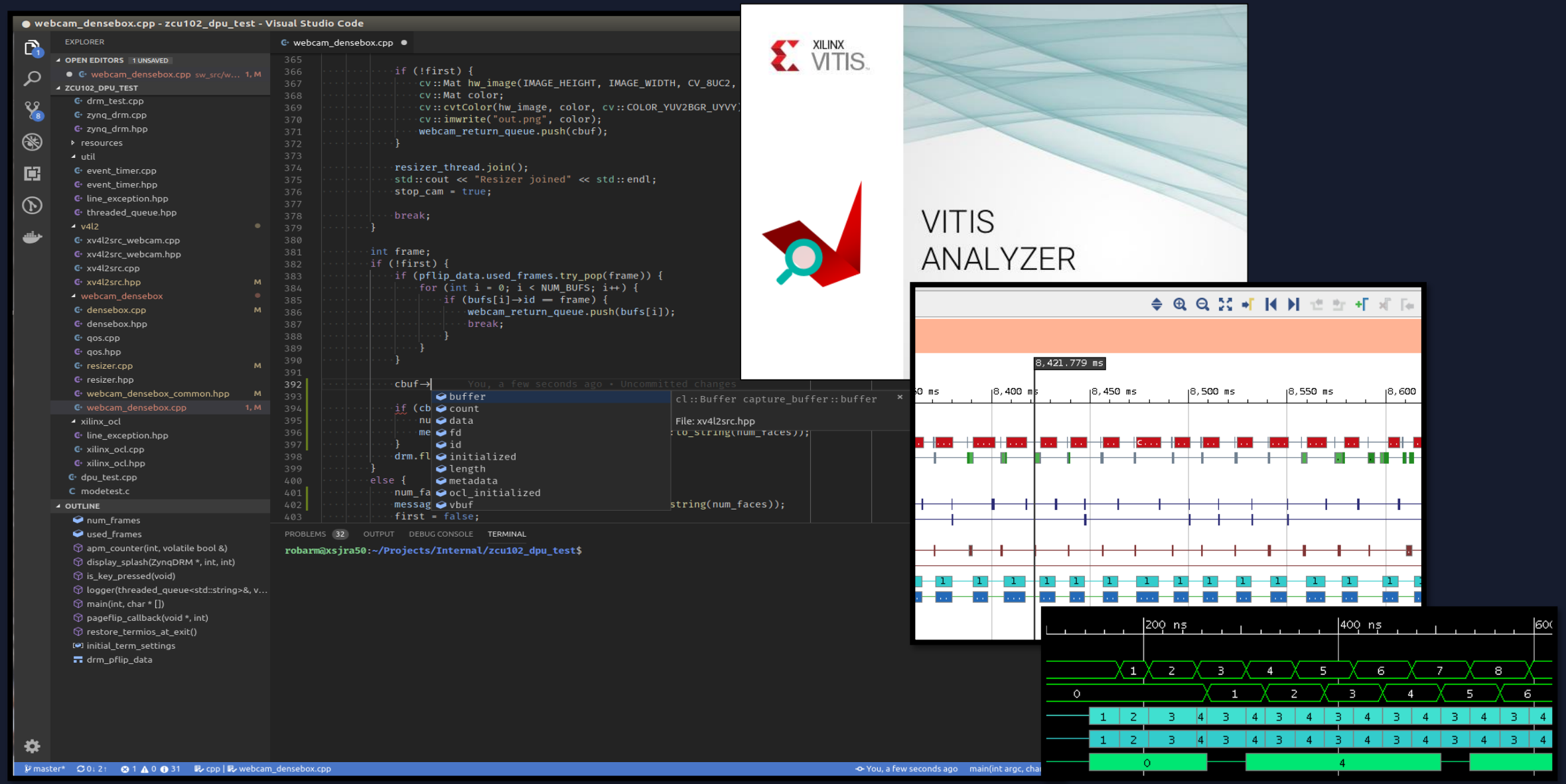


General Vitis Workflow

- Vitis has a clearly defined hardware and software build flow
- Hardware is *composed* in C/C++ using high-level compilers
  - Software determines acceleration *contents*, calling pre-optimized libraries, RTL functions, or translating software directly into hardware
  - Command line options to the compiler and linker define system *composition* and *connectivity*
- The software build flow is identical to a "pure" software build flow
  - Interactions with the hardware happen via the XRT API



Familiar SW Environment

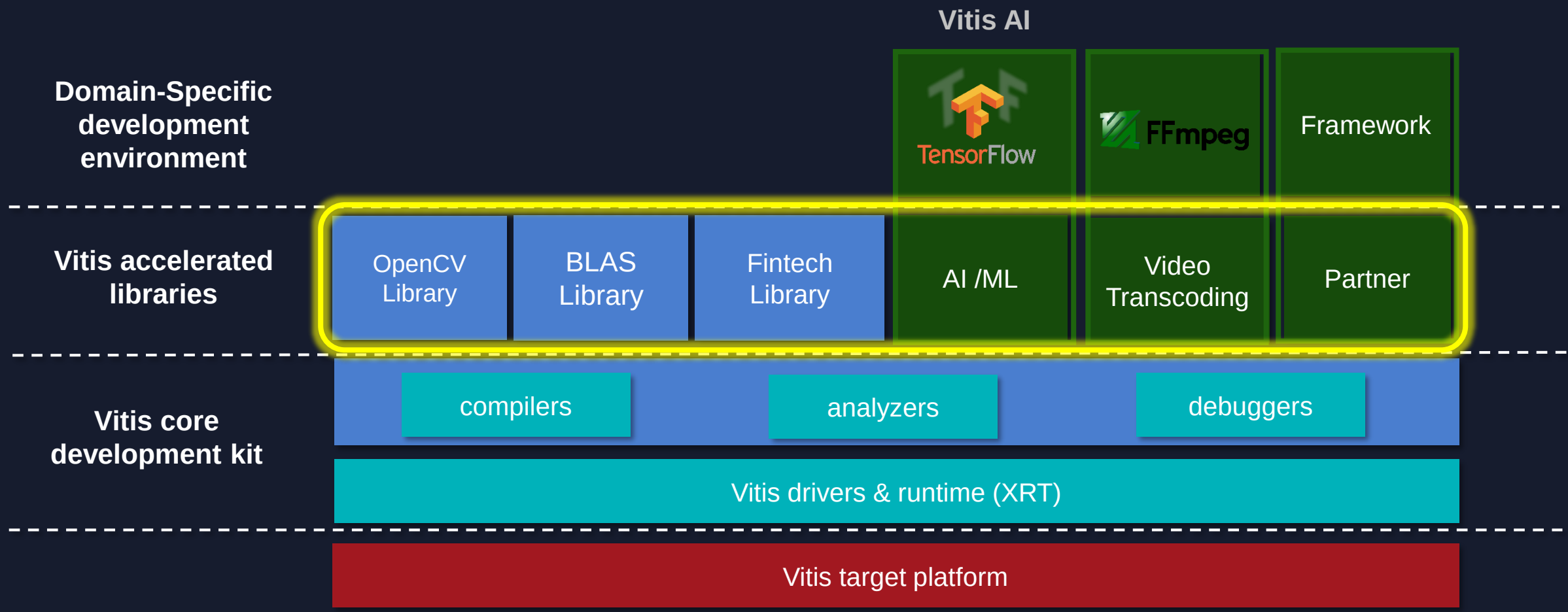


XILINX

Developing Accelerators

- In the software tools, Xilinx collectively refer to the accelerators placed into the FPGA as “kernels”
- Kernels can be developed using any method you choose:
  - Via high-level synthesis with C, C++, and OpenCL
  - From tools such as Model Composer, MATLAB, and Simulink
  - From RTL
- The acceleration tools will link these together into reconfigurable binaries to be loaded onto the FPGA at-will to perform acceleration
- Xilinx software tools also provide extensive emulation support, enabling system-level verification and fast debugging of designs

Vitis: Unified Software Platform

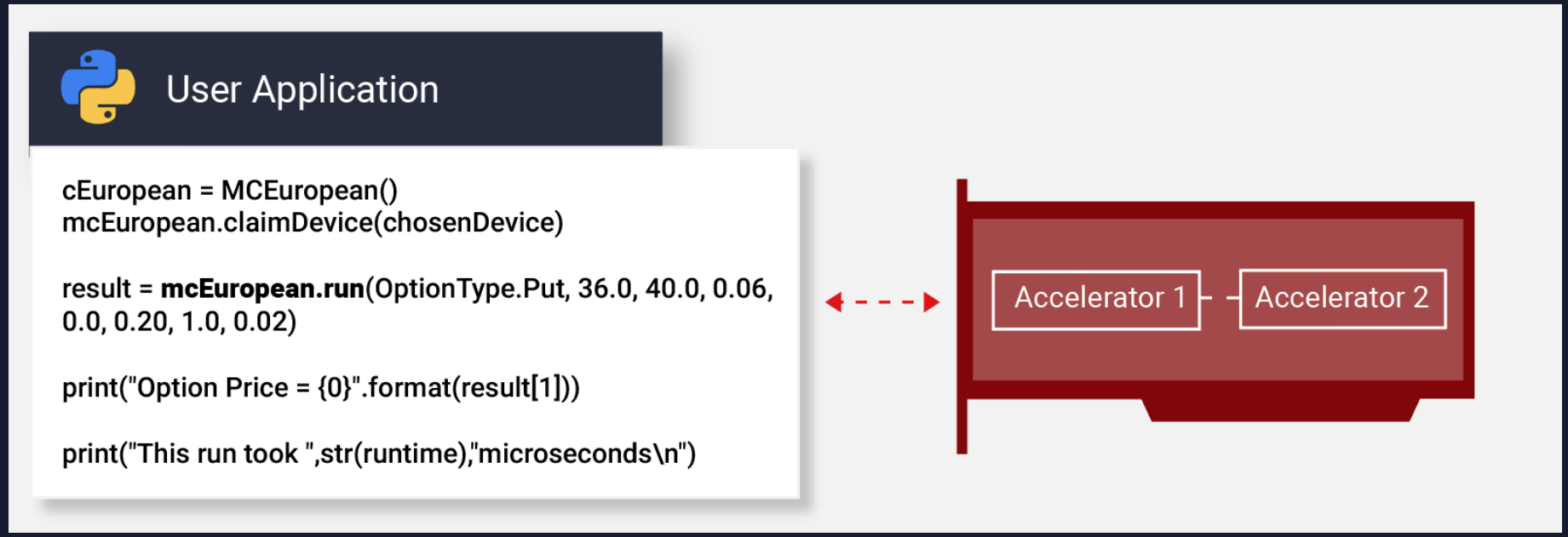


Vitis Accelerated Libraries – What?

- Open-Source, performance-optimized libraries offering out-of-the-box acceleration.

Scalable

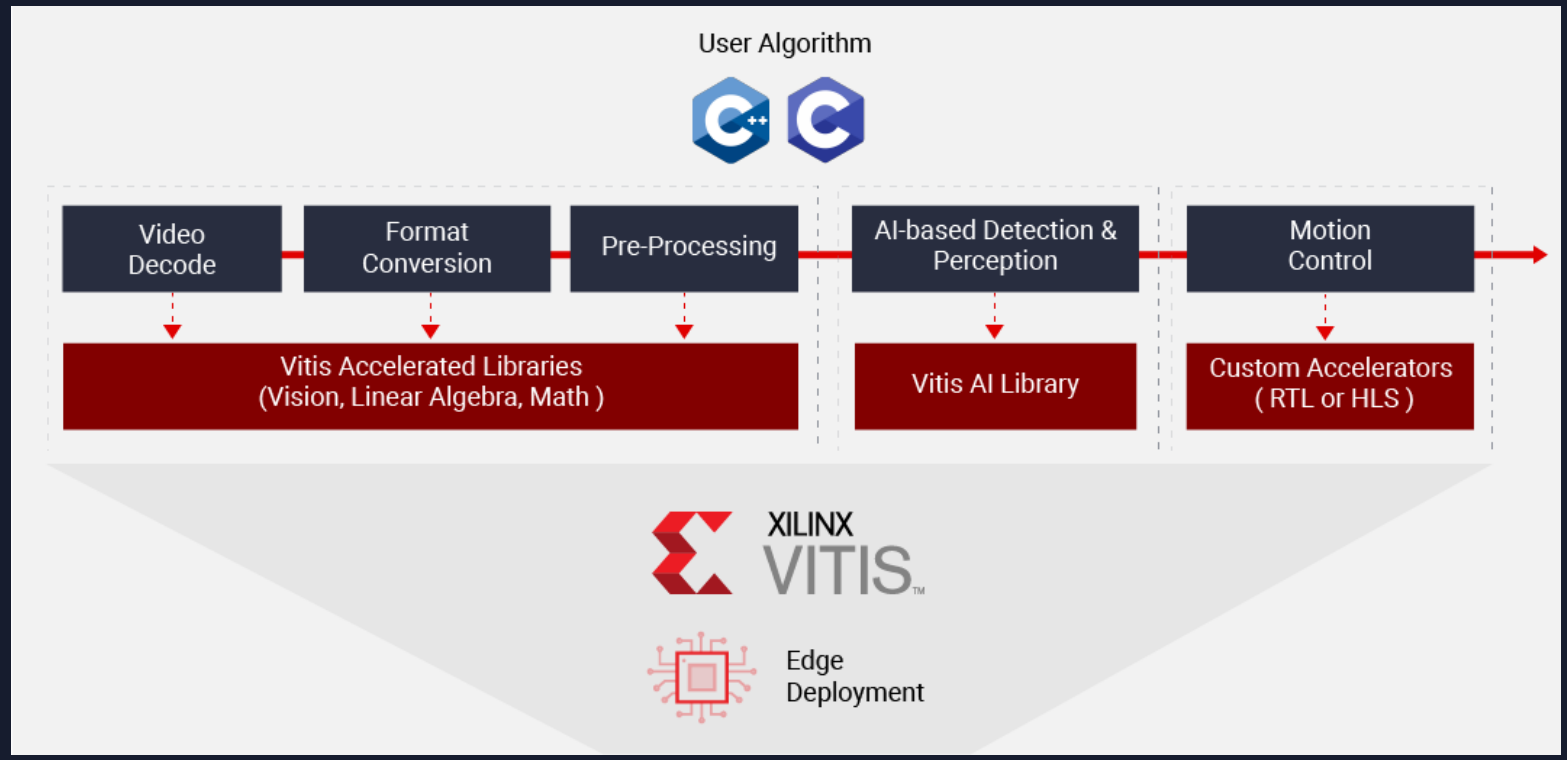
› Vitis Accelerated Libraries are accessible through GitHub and scalable across all Xilinx Platforms.



› Seamlessly deploy applications at the edge, on-premise, or in the cloud without having to reimplement your accelerators

Flexible

- Configure, modify, or use the library functions as-is.
- The Vitis accelerated libraries provide the flexibility needed to design custom application accelerators



XILINX

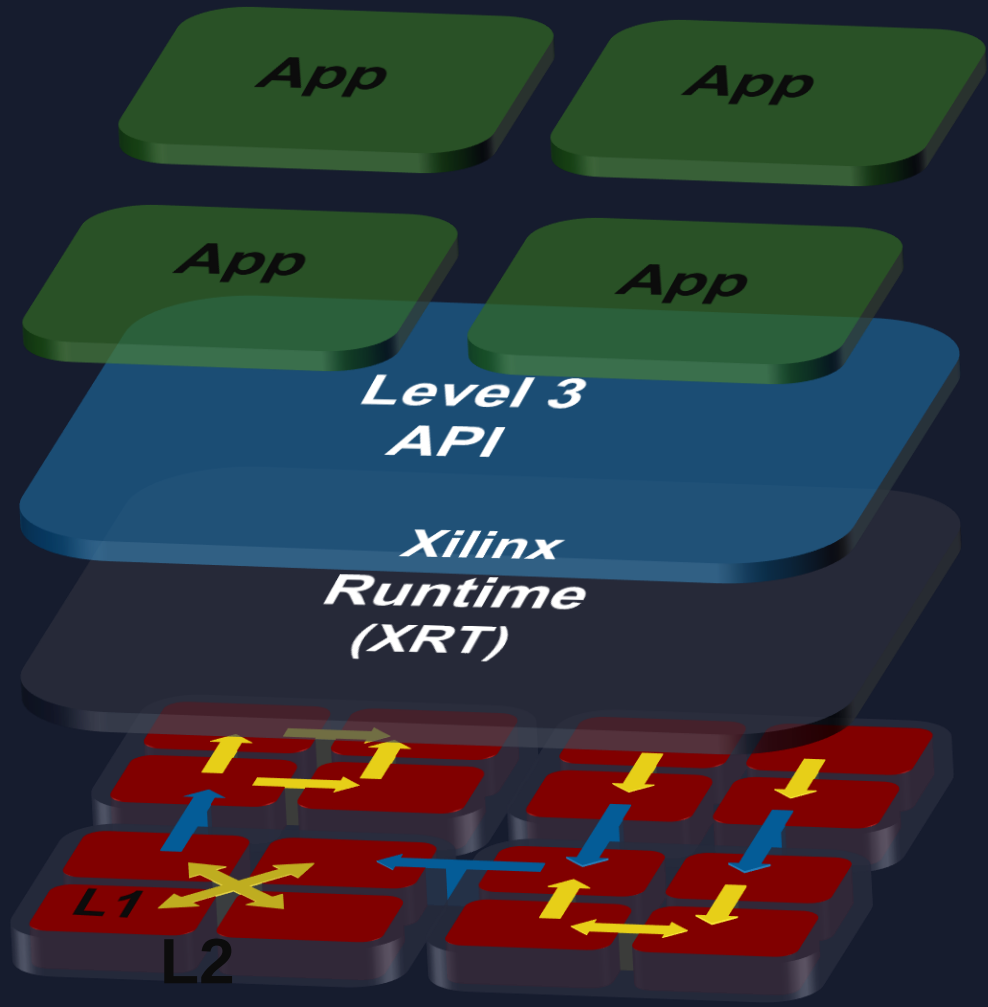
Vitis Accelerated Libraries Structure

- Level 1 – Vitis Library Primitives
  - Optimized functions used to build kernels
- Level 2 – Vitis Library Kernels
  - High-Level optimized kernels with required interfaces
- Level 3 – Vitis Library API
  - Software-API
  - Initialization and data transfers handled automatically

XILINX.

Vitis Accelerated Libraries Structure

- User can interact with the libraries at all three levels
- Modify primitives for a particular application or used them as templates for new ones.
- Customize or create new kernels.
- Combine existing and custom primitives and kernels to create new libraries
- Modify the library API to support new functions and system configurations



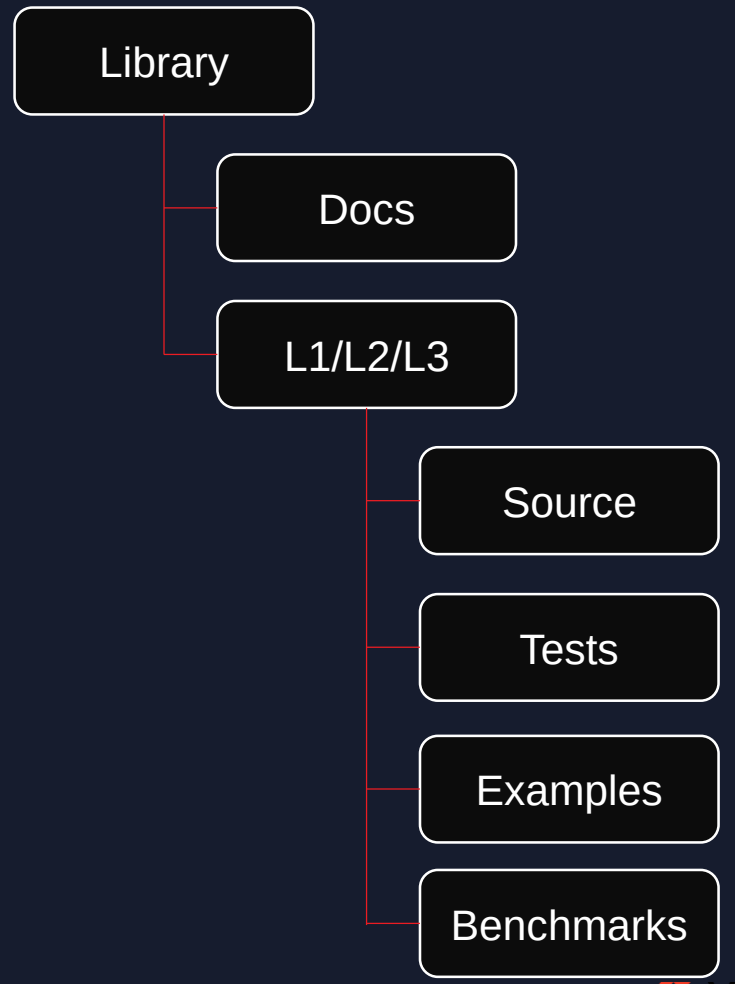
XILINX

Library Repository

- Vitis Accelerated Libraries are hosted on GitHub.

www.github.com/Xilinx/vitis_libraries

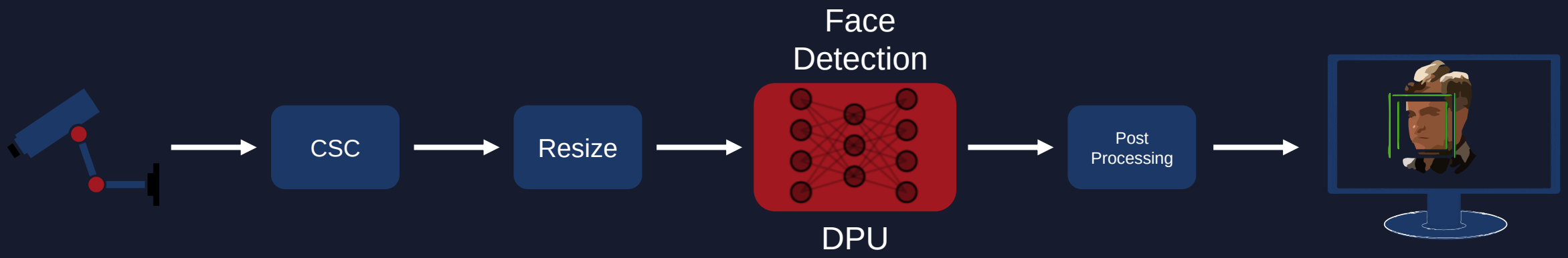
- The repository includes everything needed for development
  - Documentation
  - Primitive, kernel, and software source files
  - Tests at each library level
  - Examples
  - Demos
  - Benchmark information



Use Case: 1080P Face Detection

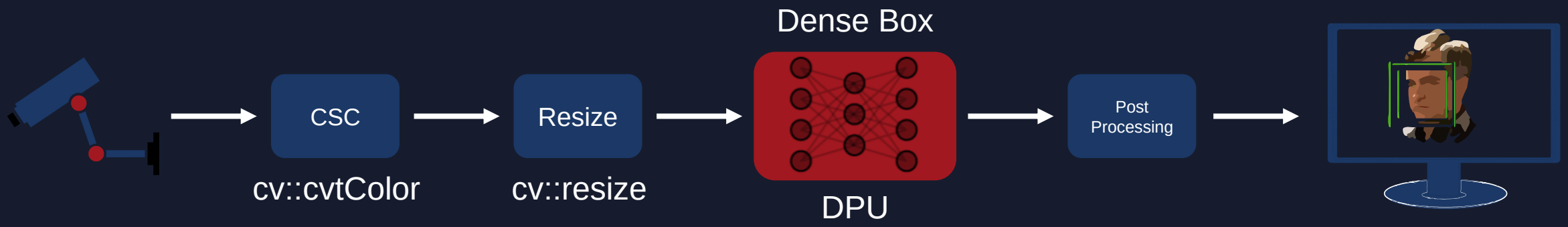
Design Overview

- Design captures 1080p60 camera data, runs a neural network inference, and displays the results on a monitor.



- The neural network has been trained to accept RGB images at 640x360
- The camera outputs UYVY video format.
- The design must convert the incoming images to RGB and resize them.

Base Performance



Operation	Time	Interpretation
Camera Capture	34.407 ms	Time required to receive one video frame and convert it to BGR
DPU Create Task	3.992 ms	Initial setup and initialization of the DPU
OpenCV Resize	16.356 ms	Resizing, 1920x1080 down to 640x360
DPU Set Input Image	2.238 ms	Copy input image into the DPU's buffers
DPU Run Task	12.443 ms	Actual ML processing time in the B1152F DPU
SoftMax	2.284 ms	SoftMax output layer, processed in software
NMS	0.022 ms	Non-maximal suppression, processed in software

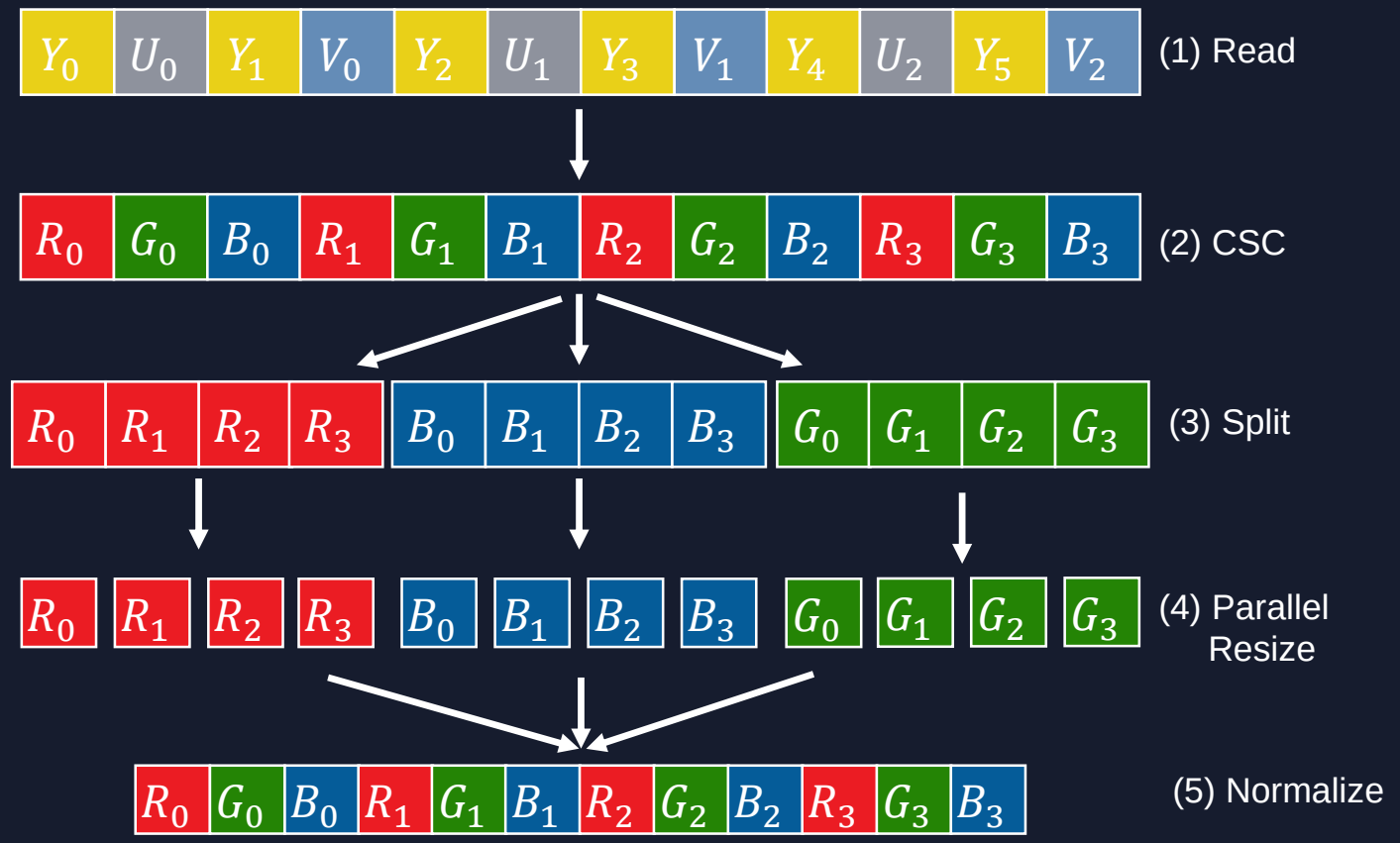
Performance: ~13 FPS

Pre-process Acceleration using Vitis Vision

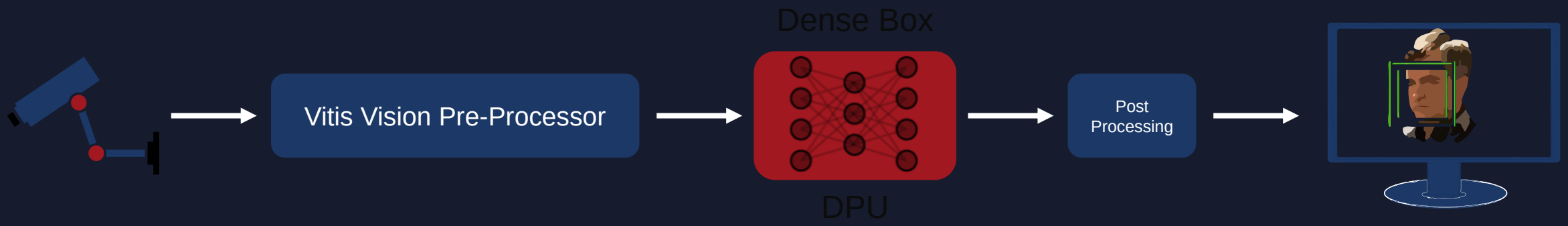
- Acceleration is done in completely in software using the Vitis Vision library
- cv::cvtColor() maps to xf::cv::uyvy2bgr()
- cv::resize() maps to xf::cv::resize()
- Data movement between the application and the accelerated functions as well as scheduling and configuration is handled by the Xilinx Runtime (XRT)

Pre-Processing Pipeline

```
void preProcess( ... )
{
        # Read in data
        xf::Array2xfMat(image_in, in_mat)

        # Color Space Converter
        xf::yuyb2bgr(in_mat, in_rgb)

        # Parallel Resize
        xf::resize(in_rgb, out_rgb)

        # Normalize and Resize Color Components
        normalize(out_mat, out_rgb)

        # Write out data
        xf::xfMat2Array(out_mat, image_out)

}
```



End-to-End Acceleration Performance



- With a clock rate of 300 MHz, processing one pixel per clock. We can process a whole 1080p frame in
- $1920 \times 1080 \times \frac{1}{300\ MHz} = 6.91\ ms$
- Can now easily increase our DPU efficiency and meet our FPS target of 60 fps

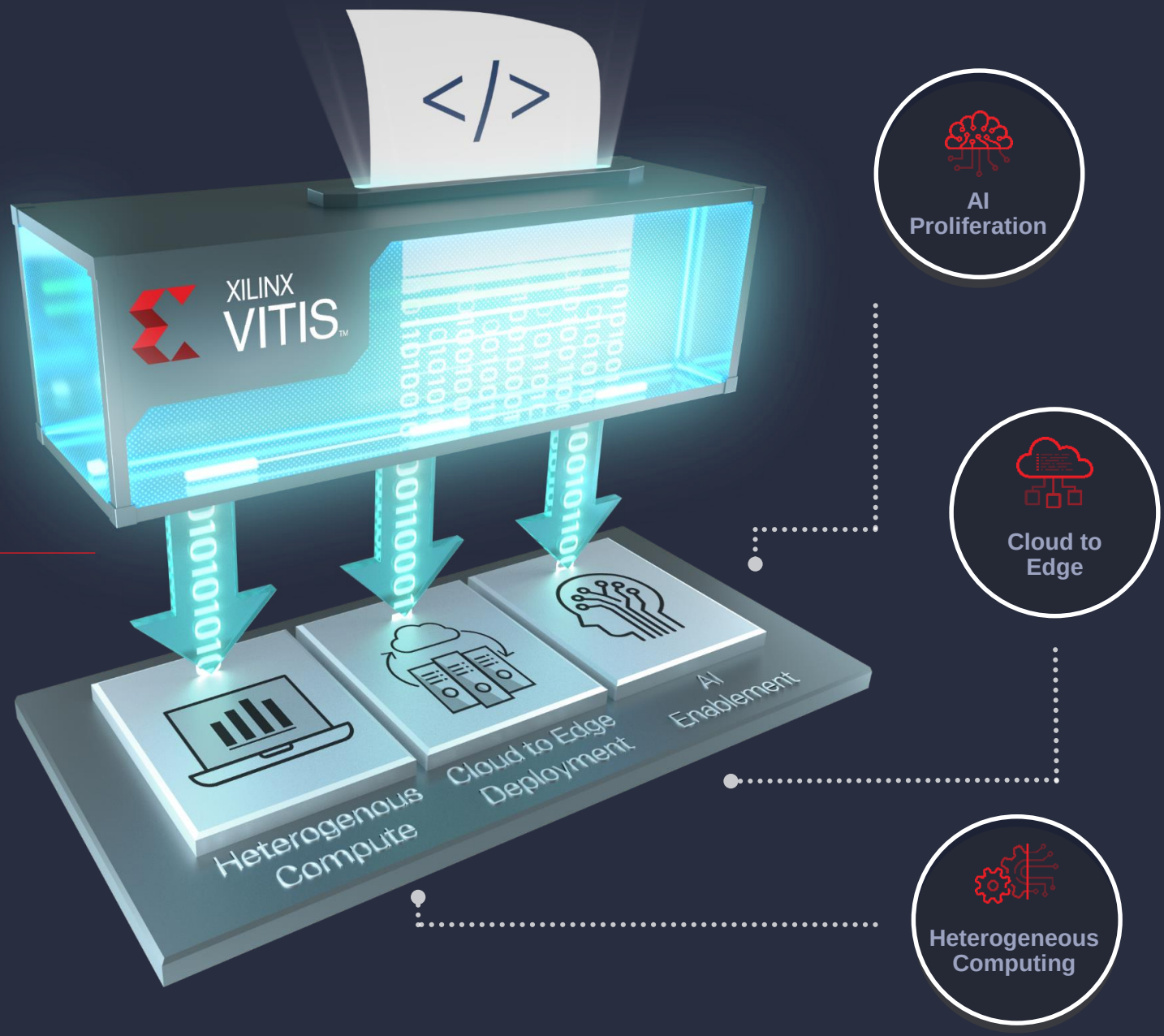
Performance: 60 FPS (~4.6x)

Summary

The Era of Software-Driven Architecture

- Available now
- Standards, Open

Free!



Call to Action

- Take a test drive! If you don’t have an Alveo card or other Xilinx board today, try Vitis in the cloud!
- Refer to our wide array of Vitis getting started examples here:
  - https://github.com/Xilinx/Vitis_Accel_Examples
- Check out our new Xilinx Developer Site!
  - Find tutorials, onboarding, application examples, and documentation to get started with Vitis immediately
  - https://developer.xilinx.com
- Download Vitis from Xilinx.com today!

Q & A

Adaptable.
Intelligent.

XILINX