Kria Robotics Stack (KRS)

The **Kria Robotics Stack (KRS)** is a ROS 2 superset for industry, an integrated set of robot libraries and utilities to accelerate the development, maintenance and commercialization of industrial-grade robotic solutions while using adaptive computing.

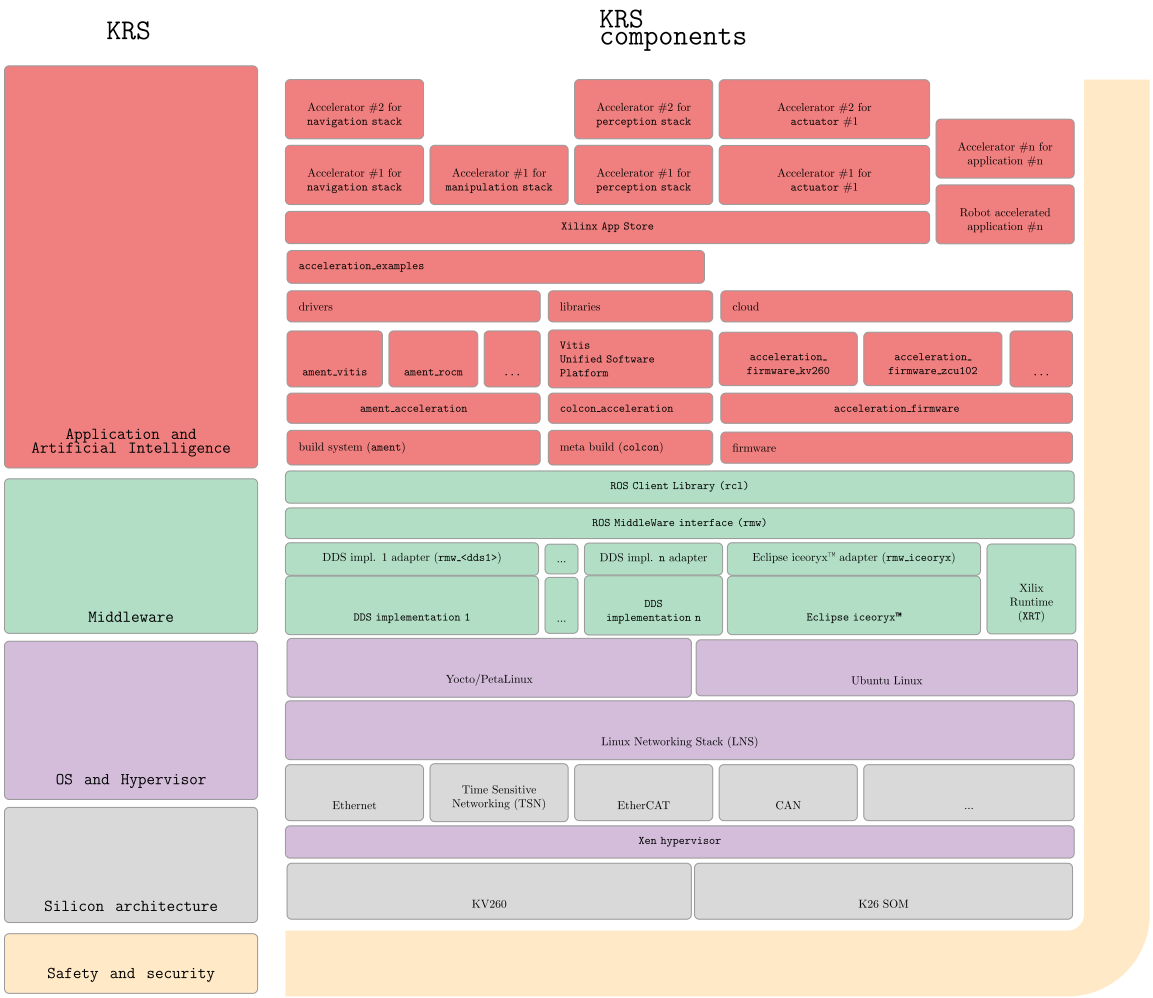
Alpha release

KRS is still on **alpha** release. Correspondingly, the documentation provided here is not intended for production environments and should be used only for evaluation purposes.

Stay tuned for upcoming official releases.

Learn ROS 2 first

KRS builds on top of ROS 2. If you're not familiar with ROS 2, **learn this first with its official documentation**.



KRS is around ROS 2, for roboticists and hardware acceleration, it leverages the Kria SOM portfolio to deliver low latency (real-fast), determinism (predictable), real-time (on time), security and high throughput to robotics. In a nutshell, KRS brings to ROS 2 roboticists the capability to create custom secure compute architectures, easily.

It does so by tightly integrating itself with ROS (lingua franca amongst roboticists) and by leveraging a combination of modern C++ and High-Level Synthesis (HLS), together with reference development boards and design architectures roboticists can use to kick-start their projects. Altogether, KRS supports Kria SOMs with an accelerated path to production in robotics.

ROS 2-centric experience

ROS is to roboticists what Linux is to most computer scientists and software developers. It helps roboticists build robot applications. With ROS 2, the capabilities to produce robot behaviours become production ready, with the potential to impact many industries. Opposed to reinventing the wheel with new robotics platforms that overload the space with replicas or forks of libraries and/or similar robotics simulators, Xilinx's approach with KRS meets the ROS robotics community interests and builds on top of ROS 2, together with its tightly connected robotics simulator, Gazebo.

To connect Xilinx's hardware acceleration technology with the ROS 2 world in a way that encourages package maintainers to benefit from it, Xilinx has created a series of extensions to the ROS 2 build system (`ament`) and meta build tools (`colcon`) that minimize the effort required from ROS 2 package maintainers. This architecture is based on three pillars.

Pillar I - Extensions to ament build system

The first pillar represents extensions of the `ament` ROS 2 build system. These CMake extensions help achieve the objective of simplifying the creation of acceleration kernels. By providing an experience and a syntax similar to other ROS 2 libraries targeting CPUs, maintainers will be able to integrate kernels into their packages easily. The `ament_acceleration` ROS 2 package abstracts the build system extensions from technology-specific frameworks and software platforms. This allows to easily support hardware acceleration across FPGAs and GPUs while using the same syntax, simplifying the work of maintainers. The code listing below provides an example that instructs the `CMakeLists.txt` file of a ROS 2 package to build a `vadd` kernel with the corresponding sources and configuration:

```
acceleration_kernel(
  NAME vadd
  FILE src/vadd.cpp
  INCLUDE
    include
  TARGET kv260
)
```

Under the hood, each specialization of `ament_acceleration` should rely on the corresponding technology libraries to enable it. For example, `ament_vitis` relies on Vitis Unified Software Platform and on the Xilinx Runtime (XRT) library to generate acceleration kernels and facilitate OpenCL communication between the application code and the kernels. Vitis and XRT are completely hidden from the robotics engineer, simplifying the creation of kernels through simple CMake macros. The same kernel expressed with `ament_vitis` is presented below:

```
    vitis_acceleration_kernel(
      NAME vadd
      FILE src/vadd.cpp
      CONFIG src/kv260.cfg
      INCLUDE
        include
      TYPE
        sw_emu
        # hw_emu
        # hw
      PACKAGE
    )
```

While `ament_acceleration` CMake macros are preferred and will be encouraged, maintainers are free to choose among the CMake macros available. After all, it'll be hard to define a generic set of macros that fits all use cases across technologies.

Through `ament_acceleration` and technology-specific specializations (like `ament_vitis`), the ROS 2 build system is automatically enhanced to support producing acceleration kernel and related artifacts as part of the build process when invoking `colcon build`. To facilitate the work of maintainers, this additional functionality is configurable through `mixins` that can be added to the build step of a ROS 2 workspace, triggering all the hardware acceleration logic only when appropriate (e.g. when `--mixin kv260` is appended to the build effort, it'll trigger the build of kernels targeting the KV260 hardware solution). For a reference implementation of these enhacements, refer to `ament_vitis`.

Pillar II - Extensions to colcon build tools

The second pillar extends the `colcon` ROS 2 meta built tools to integrate hardware acceleration flows into the ROS 2 CLI tooling and allow to manage it. Examples of these extensions include emulation capabilities to speed-up the development process and/or facilitate it without access to the real hardware, or raw image production tools, which are convenient when packing together acceleration kernels for embedded targets. These extensions are implemented by the `colcon_acceleration` ROS 2 package. On a preliminary implementation, these extensions are provided the following `colcon acceleration` subverbs:

```
    colcon acceleration subverbs:
      board                 Report the board supported in the deployed
firmware
      emulation             Manage hardware emulation capabilities
      hls                   Vitis HLS capabilities management extension
      hypervisor            Configure the Xen hypervisor
      linux                 Configure the Linux kernel
      list                  List supported firmware for hardware acceleration
      mkinitramfs           Creates compressed cpio initramfs (ramdisks)
from raw image
      mount                 Mount raw images
      package               Packages workspace with kernels for distribution
```

```
      platform              Report the platform enabled in the deployed
firmware
      select                Select an existing firmware and default to it.
      umount                Umount raw images
      v++                   Vitis v++ compiler wrapper
      version               Report version of the tool
```

Using the `list` and `select` subverbs, it's possible to inspect and configure the different hardware acceleration solutions. The rest of the subverbs allow to manage hardware acceleration artifacts and platforms in a simplified manner.

In turn, the list of subverbs will improve and grow to cover other technology solutions.

Pillar III - a firmware layer for portability

The third pillar is firmware. Represented by the abstract `acceleration_firmware` package, it is meant to provide firmware artifacts for each supported technology so that the kernels can be compiled against them, simplifying the process for consumers and maintainers, and further aligning with the ROS typical development flow.

Each ROS 2 workspace can have one or multiple firmware packages deployed. The selection of the active firmware in the workspace is performed by the `colcon acceleration select` subverb (pillarII). To get a technology solution aligned with this REP's architecture, each vendor should provide and maintain an `acceleration_firmware_<solution>` package specialization that delivers the corresponding artifacts in line with its supported categories_ and capabilities_. Firmware artifacts should be deployed at `<ros2_workspace_path>/acceleration/firmware/<solution>` and be ready to be used by the ROS 2 build system extensions at (pillarI) . For a reference implementation of specialized vendor firmware package, see `acceleration_firmware_kv260` .

By splitting vendors across packages with a common front-end (`acceleration_firmware`), consumers and maintainers can easily switch between hardware acceleration solutions and

Real-time ROS 2

Real-time is an end-to-end characteristic of a robotic system. A ROS 2 application running in a scalar processor (i.e. a CPU) suffers from different sources of indeterminism. The picture above depicts these across the OSI stack. For a robot to respond in a deterministic manner while exchanging information inter-process, intra-process or intra-network while using ROS 2, all layers involved across the OSI stack must respond deterministically. It's not possible to guarantee real-time in ROS 2 unless all overlays and underlays are equally time bounded. Correspondingly, **for a real-time ROS 2 interaction, all of its layers must also be real-time**. In the case of ROS 2 running in CPUs, for each one of these levels, the sources of indeterminism need to be addressed.

FPGAs allow to design robot circuitry that provides deterministic responses. It possible to design hard real-time robotic systems relying solely on the FPGA however when interfacing with ROS 2 often running on the scalar processors (CPUs), the determinism is often compromised. The goal of KRS is to provide mechanisms to mitigate all these indeterminism issues in the scalar processors through a modular approach. Customers can then prioritize and use selected modules to remove the desired sources of indeterminism, adapting time mitigations to each use case.

In general, solutions to real-time problems in CPUs fall into two big categories: a) setting the correct priority in the corresponding abstraction and b) applying Quality of Service (QoS) techniques. Each layer has its own QoS methods. In OSI Layer 2 we have the well-known QoS techniques specified in the existing IEEE 802.1Q standards, and new techniques such as the Time Sensitive Networking (TSN) standards. For the Linux Network Stack (OSI Layers 3 and 4), traffic control allows to configure QoS methods. Similarly, from the Linux kernel all the way up to the application libraries, each layer needs to be configured for bounded maximum latencies in order for the robotic system to deliver real-time capabilities. The sections below analyze briefly the indeterminism sources one by one and provide a short discussion around the solutions integrated within KRS:

①	**Link layer congestion bottlenecks**: real-time problems related to Data Link layer bottlenecks (OSI L2) are a relevant source of indeterminism in robotic systems. The most common scenario is when different traffic sources are present in the same network segment with different orders of priority. For example, congestion problems may arise when various depth sensors generating point clouds of the environment are transmitted over switched Ethernet networks, mixed together with higher priority traffic (e.g. that commands actuators). This may lead to undesired indeterminism or even to packet drops for time critical packages. KRS addresses these problems by providing time sensitive link-layer components such as TSN, which enables deterministic real-time communications over Ethernet. Another source of indeterminism in the Data Link layer comes while interoperating between different communication buses. For example, relaying the sensing data perceived over Ethernet to an EtherCAT fieldbus, or passing annotated data from Ethernet to CAN. The interfacing between these networks in scalar processors typically happens at higher layers (e.g. OSI L7) which besides consuming many additional processing cycles, exposes data to the additional indeterminism of upper abstraction layers. By leveraging the FPGA, KRS provides capabilities to create specialized circuitry to bridge communications between these networks while operating on the data, everything while at the Data Link layer.
②	**Linux Networking Stack (LNS) queues congestion**: the real-time problems in the Linux Network Stack (LNS) arise when networking resources (e.g. queues) shared between ROS 2 publishers and subscribers in the same device cause congestion issues, leading to undesired latencies. These delays become specially apparent when considering the threads processing the networking messages, which might be competing for CPU time (further enhancing the real-time issues at ③). KRS leverages the Linux kernel Traffic Control (`tc`) capabilities to mitigate the indeterminism of the LNS.
③	**Linux kernel indeterminism**: this source accounts for the real-time problems related with the indeterminism inside the Operating System (OS) or more specifically, the lack of timely preemptions at the kernel level, including its drivers. As an example, the lack of preemption capabilities in a kernel driver may leads to communication interactions that suffer delays because CPU cycles arrive late when processing, delivering or retrieving messages. KRS addresses these real-time issues by a) offering a fully preemptible Linux kernel (`PREEMPT_RT`) and b) by maintaining a Xilinx-specific Linux kernel fork with drivers optimized for determinism and preemption.

④	**Last level cache (LLC) contention**: Another large source of real-time interference in multi-core scalar processors is the last level cache. In a multi-core CPUs, the last level cache (LLC) is shared by all cores and means that a noisy neighbor can have a drastic impact on the worst-case execution time (WCET) of software on other cores. Past research showed a slowdown factor of 340 times, swamping the theoretical 4 times speedup expected from a quad core multi-core processor. KRS addresses this problem by cache coloring at the Xen hypervisor level. Cache coloring consists of partitioning the L2 cache, allocating a cache entry for each Virtual Machine (VM) in the hypervisor. This approach allows real-time applications to run deterministic IRQs, lowering the time variance, reducing the effects of cache interference and allowing mixed critical workloads on a single SoC.
⑤	**Communication middleware indeterminism**: Often architected and programmed with a data connectivity and throughput mindset (as opposed to determinism), the communication middleware also represents a source of indeterminism. Real-time safe coding practices should be applied when implementing the communication middleware, and also when used in higher abstraction layers. In addition, it should also be considered that the communication middleware builds on top of L1-L4 of the OSI stack, which forces the middleware implementations to rely on deterministic lower layers for overall end-to-end real-time capabilities. KRS currently provides support for the most popular communication middlewares in the ROS 2 world. Future enhancements are planned to make these implementations rely on real-time deterministic lower layers.
⑥	**ROS 2 core layers indeterminism**: Similar to ④, the ROS 2 layers should be built using real-time safe primitives and real-time capable underlayers. KRS initially supports the official ROS 2 `rclcpp`, `rcl`, `rmw`, and `rmw_adapters`. Ongoing work is in progress to explore other architectures which push some of the ROS 2 abstractions to hardware.
⑦	**Robotic applications and libraries indeterminism**: Finally, at the application level, the robotics code including all its libraries also need to be real-time safe and use only the underlying primitives and layers that guarantee determinism.

In addition to mitigations for these problems, KRS also packs capabilities to facilitate creating systems for mixed criticality. The following figure shows two examples of how KRS extends `colcon` (through `colcon_acceleration`) to easily 1) select a fully preemtible Linux kernel and 2) a mixed critical setup in the same SoC involving the Xen hypervisor with three Virtual Machines (VMs).

Accelerated Apps for Robots

The ROS ecosystem brings together a worldwide community of thousands of roboticists developing robot applications while using ROS 2 abstractions. In a way, ROS is the common API roboticists use when building robot behaviours. The reference Software Development Kit (SDK) in robotics. With the advent of mixed source technology ecosystems in robotics, in the ROS world, there are already various examples of companies providing value around open source packages, while contributing back to the community. Xilinx attempts to bring those capabilities to every single ROS 2 package maintainer by introducing a mixed source ecosystem powered by KRS:

With KRS, we take a step further and propose a layer for commercialization of ROS 2 robot accelerated applications. By leveraging Xilinx experience using encryption and authentication to secure FPGA bitstreams, the Xilinx App Store proposes a managed, easy to use and secure infrastructure for digital rights management (DRM) whereto commercialize and discover global customers. Through the Xilinx App Store connection, KRS containerizes ROS 2 overlay workspaces into robot accelerated applications allowing roboticists to protect and monetize their accelerated ROS 2 packages.

Beyond extending the ROS 2 build system and tools, to faciliate the process of monetizing ROS 2 packages, KRS also facilitates additional tools and extensions to simplify the process of packaging ROS 2 overlay workspaces and shipping them into the Xilinx App Store. `colcon acceleration package` subverb above shows one of such tools.

Contributing back to ROS 2

Xilinx is a proud supporter of the ROS ecosystem, contributing to ROSCon and ROS-Industrial activities. While KRS focuses on adaptive computing, as part of Xilinx commitment with ROS 2, the ROS 2 Hardware Acceleration Working Group (HAWG) was created. With it, Xilinx is contributing back to the ROS 2 community and driving the creation, maintenance and testing of acceleration kernels on top of open standards (C++ and OpenCL) for optimized ROS 2 and Gazebo interactions over different compute substrates (including FPGAs and GPUs). The main driver behind this WG is summarized at the ROS Enhancement Proposal (REP) *"ROS 2 Hardware Acceleration Architecture and Conventions"*. In here, Xilinx describes the architectural pillars and conventions required to introduce hardware acceleration in ROS 2 in a scalable and technology-agnostic manner.

As specified on its community announcement, the HAWG targets hardware acceleration in a) embedded (edge) devices, b) workstations, c) data centers and d) cloud. A roadmap with objectives is available in the community repository of the Working Group, which Xilinx is driving and executing disclosing results as they become available.

Through the HAWG, Xilinx is demonstrating hardware acceleration capabilities in ROS 2 on edge devices with simple examples at the application layer that roboticists could use as blueprint for their designs. In the future, the group will also target ROS 2 underlayers to optimize interactions between nodes within the ROS 2 computational graph.

To learn more about Xilinx contributions to the ROS 2 community, check out the following resources:

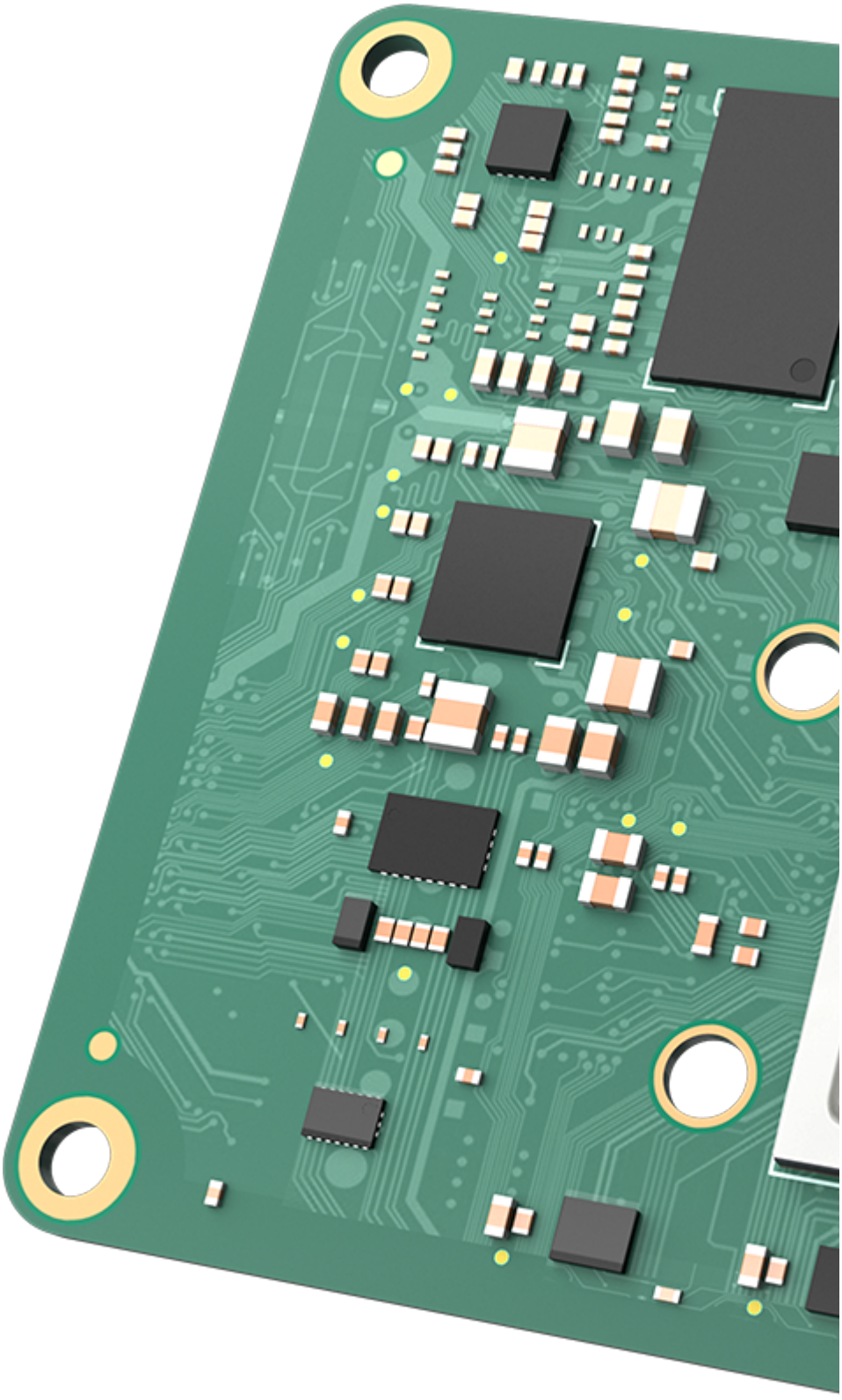
Resource	Link
`ROS 2 HAWG GitHub organization`	ros-acceleration

Resource	Link
`ROS 2 HAWG meeting minutes`	HAWG minutes
`ROS 2 HAWG meeting invite group`	ROS 2 Hardware Acceleration WG Google Group
`ROS 2 HAWG instant messaging`	Matrix community (Matrix is an open network for secure, decentralized communication).
`ROS 2 HAWG backlog management`	GitHub project
`ROS 2 HAWG discourse tag`	wg-acceleration

Hardware supported

KRS focuses on the Kria SOM portfolio. KRS full support is provided for the following boards:

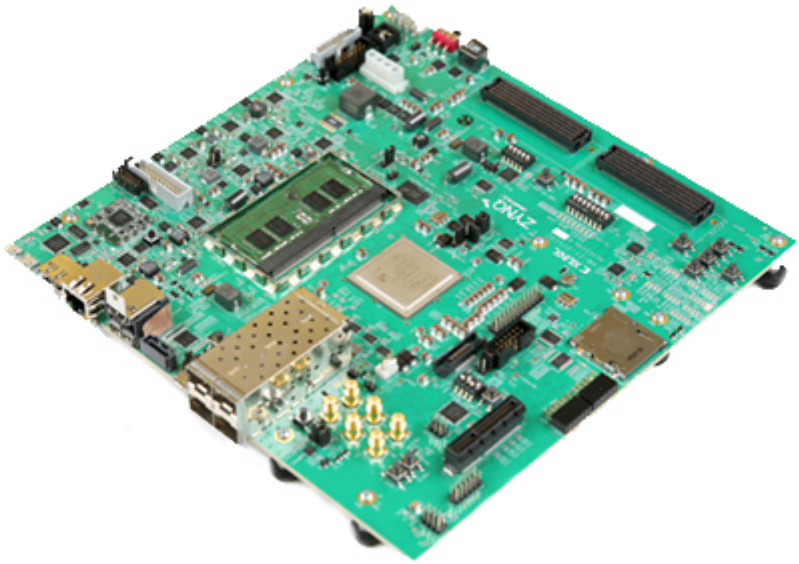
Kria K26
Adaptive
System-
on-Module



Board	Picture
Kria `KV260` Vision AI Starter Kit	

Unofficial support

Unofficial support is also available for the following:

Board	Picture
Xilinx Zynq UltraScale+ MPSoC `ZCU102` Evaluation Kit	

Board	Picture
Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit `ZCU104`	

Install KRS

Dependencies

KRS is currently only available for Linux and **has only been tested in `Ubuntu 20.04`**. KRS assumes the following is installed in your workstation:

- `Ubuntu 20.04` Focal Fossa operating system (download).
- the Vitis `2020.2.2` suite (Vitis, Vivado, Vitis HLS) (install instructions)
- the ROS 2 Foxy distribution (install instructions)

KRS is served as a group of ROS 2 packages that you can install in any arbitrary ROS 2 workspace, enhancing it with hardware acceleration capabilities. The following demonstrates how to create a new ROS 2 overlay workspace and fetch the KRS packages, including some acceleration examples:

Warning

The repos in here are only for internal evaluation. The final `krs.repos` file will contain only GitHub public references by the time it's released.

```
#####################################################
# 1. install some dependencies you might be missing
#####################################################
sudo apt-get -y install curl build-essential libssl-dev git wget \
                        ocl-icd-* opencl-headers python3-vcstool \
                        python3-colcon-common-extensions python3-colcon-
mixin \
                        kpartx u-boot-tools

#####################################################
# 2. create a new ROS 2 workspace
#####################################################
mkdir -p ~/krs_ws/src; cd ~/krs_ws

#####################################################
# 3. Create file with KRS alpha release
# TODO: this file should be fetched with wget once
#       KRS repository is publicly available
#####################################################
cat << 'EOF' > krs.repos
repositories:
  acceleration/acceleration_firmware:
    type: git
    url: https://github.com/ros-acceleration/acceleration_firmware
```

```
    version: main
  acceleration/acceleration_firmware_kv260:
    type: zip
    url: https://github.com/ros-acceleration/acceleration_firmware_kv260/
releases/download/v0.6.0/acceleration_firmware_kv260.zip
  acceleration/colcon-acceleration:
    type: git
    url: https://github.com/ros-acceleration/colcon-acceleration
    version: main
  acceleration/ros2acceleration:
    type: git
    url: https://github.com/ros-acceleration/ros2acceleration
    version: main
  acceleration/ament_vitis:
    type: git
    url: https://github.com/ros-acceleration/ament_vitis
    version: main
EOF

#####################################################
# 4. import repos of KRS alpha release
#####################################################
vcs import src --recursive < krs.repos  # about 5 mins

#####################################################
# 5. build the workspace and deploy firmware for hardware acceleration
#####################################################
source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+
colcon build --merge-install  # about 2 mins

#####################################################
# 6. source the overlay to enable all features
#####################################################
source install/setup.bash
```

That's pretty much it, you've got now KRS installed in the `krs_ws` ROS overlay workspace. You could also reproduce the same steps over an existing ROS 2 workspace if you'd like to avoid creating a new, or simply reusing the source code elsewhere.

Now's time to build and run some examples.

0. ROS 2 publisher

	Source code
`vadd_publisher`	
publisher	`vadd_publisher.cpp`
`doublevadd_publisher`	
publisher	`doublevadd_publisher.cpp`

This first example presents a trivial vector-add ROS 2 publisher, which adds two vector inputs in a loop, and tries to publish the result at 10 Hz. The ROS 2 package runs in the in the scalar processors (the CPUs). Step-by-step, the process is documented, walking through the different actions required to run the ROS 2 package in hardware, leveraging KRS capabilities. Afterwards, a slighly modified version of the publisher is presented which has additional computation demands. With these modifications, it becomes clear how the publisher isn't able to meet the publication goal anymore, which motivates the use of hardware acceleration.

The ultimate objective of this example is to generate a simple ROS 2-centric example that creates a CPU baseline to understand the value of hardware acceleration and how KRS facilitates it. Next examples will build upon this one.

Warning

The examples assume you've already installed KRS. If not, refer to install.

Tip

Learn ROS 2 before trying this out first.

vadd_publisher

Prepare the environment and fetch the example

We start by preparing the environment and fetching the source code of the example into our KRS workspace:

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash
```

Inspecting the ROS 2 publisher

The publisher is a CPU-based average one. The source code of the publisher has been split between the `vadd` function (vadd.cpp) and the rest (vadd_publisher.cpp) for simplicity. The `vadd` (vector-add) function is as follows:

```
/*
        ____  ____
       /   /\/   /
      /___/  \  /   Copyright (c) 2021, Xilinx®.
      \   \   \/    Author: Víctor Mayoral Vilches <victorma@xilinx.com>
       \   \
       /   /
      /___/   /\
      \   \  /  \
       \___\/\___\

Inspired by the Vector-Add example.
See https://github.com/Xilinx/Vitis-Tutorials/blob/master/
Getting_Started/Vitis

*/

#define DATA_SIZE 4096
// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

extern "C" {
    void vadd(
            const unsigned int *in1,  // Read-Only Vector 1
            const unsigned int *in2,  // Read-Only Vector 2
            unsigned int *out,        // Output Result
            int size                  // Size in integer
            )
    {
        for (int i = 0; i < size; ++i) {
        #pragma HLS loop_tripcount min = c_size max = c_size
            out[i] = in1[i] + in2[i];
        }
    }
}
```

The `loop_tripcount` is for analysis only and the pragma doesn't impact the function logic in any way. Instead, it allows HLS to identify how many iterations are expected in the loop to make time estimations. This will come handy if we want to run synthesis tests to estimate the timing it'll take for this function to run on a dedicated circuit in the FPGA.

Building, creating the raw image and running in hardware

Let's build the example, create a raw SD card image and run it in the `KV260` reference development platform. First, let's select the firmware for the target hardware, KV260:

```
$ colcon acceleration select kv260
```

To verify that we indeed that the right firmware selected, look for the one marked with a "*" at the end of its name:

```
$ colcon acceleration list
kv260*
```

Let's now build the package targeting the KV260:

```
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select ament_vitis vadd_publisher
```

Let's now create a raw disk image for the SD card with PetaLinux's rootfs, a vanilla Linux 5.4.0 kernel and the ROS 2 overlay workspace we've just created for the KV260:

```
$ colcon acceleration linux vanilla --install-dir install-kv260
```

We're now ready to run it on hardware. For that, we need to flash the `~/krs_ws/acceleration/firmware/select/sd_card.img` file into the SD card. One quick way to do it is as follows:

```
# first, find out where your SD card has been mapped, in my case, /dev/rdisk2
$ sudo diskutil umount /dev/rdisk2s1  # umount mounted partition
$ pv <your-path-to>/krs_ws/acceleration/firmware/select/sd_card.img | sudo
dd of=/dev/rdisk2 bs=4m  # dd the image
```

There are other methods. If you need help doing so, check out these instructions for different OSs. **Make sure to flash `~/krs_ws/acceleration/firmware/select/sd_card.img` we just generated, and not some other image**.

Once flashed, connect the board to the computer via its USB/UART/JTAG FTDI adapter and power it on. Then, launch your favority serial console (e.g. `sudo tio /dev/ttyUSB1`). You should get to the following prompt where you can use the `petalinux` username and define your own password:

```
PetaLinux 2020.2.2 xilinx-k26-starterkit-2020_2.2 ttyPS0

xilinx-k26-starterkit-2020_2 login:
```

Then, we can launch the example (feel free to split the actions below in various terminals, for convenience `byobu` allows to run it all in one):

```
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation

$ . /ros2_ws/local_setup.bash     # source the ROS 2 overlay workspace we
just
                                  # created. Note it has been copied to the
SD
                                  # card image while being created.

$ ros2 topic hz /vector --window 10 &
$ ros2 run vadd_publisher vadd_publisher
...
```

```
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00007s window: 10
[INFO] [1629649453.734865115] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 37'
[INFO] [1629649453.834855650] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 38'
[INFO] [1629649453.934853073] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 39'
[INFO] [1629649454.034854160] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 40'
[INFO] [1629649454.134853854] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 41'
[INFO] [1629649454.234855027] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 42'
[INFO] [1629649454.334856580] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 43'
[INFO] [1629649454.434858073] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 44'
[INFO] [1629649454.534854066] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 45'
[INFO] [1629649454.634857869] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 46'
[INFO] [1629649454.734853412] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 47'
average rate: 10.001
    min: 0.100s max: 0.100s std dev: 0.00006s window: 10
[INFO] [1629649454.834858295] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 48'
[INFO] [1629649454.934855449] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 49'
[INFO] [1629649455.034850672] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 50'
[INFO] [1629649455.134852075] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 51'
[INFO] [1629649455.234859537] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 52'
[INFO] [1629649455.334861150] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 53'
[INFO] [1629649455.434863893] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 54'
[INFO] [1629649455.534855537] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 55'
[INFO] [1629649455.634844142] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 56'
[INFO] [1629649455.734844195] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 57'
[INFO] [1629649455.834858327] [vadd_publisher]: Publishing: 'vadd finished,
iteration: 58'
...
```

You should see that the ROS 2 publisher successfully publishes at approximately 10 Hz.

Bonus: Getting a time intuition of a `vadd` acceleration kernel

What if instead of running the `vadd` in the scalar CPUs, we create a specific hardware circuit with the FPGA that runs this function? What'd would be the associated timing? In other words, a *robot chip* for the `vadd` computation.

Though going all the way down to implementing the harware cirtuit for `vadd` is beyond the scope of this example, we can get a quick intuition with HLS capabilities integrated in KRS. Let's get such intuition:

```
$ colcon acceleration hls --run --synthesis-report vadd_publisher
Found Tcl script "project_vadd_publisher.tcl" for package: vadd_publisher
Executing /home/xilinx/krs_ws/build-kv260/vadd_publisher/
project_vadd_publisher.tcl
 Project:  project_vadd_publisher
 Path:  /home/xilinx/krs_ws/build-kv260/vadd_publisher/project_vadd_publisher
    - Solution:  solution_4ns
        - C Simulation:              Pass
        - C Synthesis:               Run
        - C/RTL Co-simulation:       Not Run
        - Export:
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
        - Synthesis report: /home/xilinx/krs_ws/build-kv260/vadd_publisher/
project_vadd_publisher/solution_4ns/syn/report/vadd_csynth.rpt


            ================================================================
            == Vitis HLS Report for 'vadd'
            ================================================================
            * Date:           Sun Aug 22 18:24:43 2021

            * Version:        2020.2.2 (Build 3118627 on Tue Feb  9 05:13:49
MST 2021)
            * Project:        project_vadd_publisher
            * Solution:       solution_4ns (Vitis Kernel Flow Target)
            * Product family: zynquplus
            * Target device:  xck26-sfvc784-2LV-c


            ================================================================
            == Performance Estimates
            ================================================================
            + Timing:
                * Summary:
                +--------+---------+----------+------------+
                |  Clock |  Target | Estimated| Uncertainty|
                +--------+---------+----------+------------+
                |ap_clk  |  4.00 ns|  2.920 ns|     1.08 ns|
                +--------+---------+----------+------------+

            + Latency:
                * Summary:
```

```
                +---------+---------+----------+----------+-----+------
+---------+
                |  Latency (cycles) |  Latency (absolute)  |  Interval  |
Pipeline|
                |   min   |   max   |    min   |    max   | min |  max |
Type  |
                +---------+---------+----------+----------+-----+------
+---------+
                |        2|     8334|  8.000 ns|  33.336 us|    3|  8335|
none|
                +---------+---------+----------+----------+-----+------
+---------+

                + Detail:
                    * Instance:
                    N/A

                    * Loop:
                    +------------------+---------+---------+----------
+-----------+-----------+------+----------+
                    |                  |  Latency (cycles) | Iteration|
Initiation Interval  | Trip |          |
                    |     Loop Name    |   min   |   max   |  Latency |
achieved |   target  | Count| Pipelined|
                    +------------------+---------+---------+----------
+-----------+-----------+------+----------+
                    |- VITIS_LOOP_30_1  |     8264|     8264|        75|
2|          1|  4096|       yes|
                    +------------------+---------+---------+----------
+-----------+-----------+------+----------+



            ================================================================
            == Utilization Estimates
            ================================================================
            * Summary:
            +-----------------+---------+------+--------+--------+-----+
            |       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
            +-----------------+---------+------+--------+--------+-----+
            |DSP              |        -|     -|       -|       -|    -|
            |Expression       |        -|     -|       0|     280|    -|
            |FIFO             |        -|     -|       -|       -|    -|
            |Instance         |        2|     -|     796|    1068|    -|
            |Memory           |        -|     -|       -|       -|    -|
            |Multiplexer      |        -|     -|       -|     472|    -|
            |Register         |        -|     -|     623|      32|    -|
            +-----------------+---------+------+--------+--------+-----+
            |Total            |        2|     0|    1419|    1852|    0|
            +-----------------+---------+------+--------+--------+-----+
            |Available        |      288|  1248|  234240|  117120|   64|
            +-----------------+---------+------+--------+--------+-----+
            |Utilization (%)  |       ~0|     0|      ~0|       1|    0|
            +-----------------+---------+------+--------+--------+-----+

...
```

The output will show you that the `vadd` function with all its complexity can run with a `4 ns` clock (we could ask for other clocks) in a total of `33.336 us`. This result is **completely deterministic** from the vadd viewpoint, after all, the FPGA would create a specialized hardware cirtuit for the `vadd` computations. For what concerns vadd, there's no more deterministic execution than this. This is just a brief introduction, *if you wish to learn more about HLS and KRS, head to the second example: 1. Hello Xilinx*.

doublevadd_publisher

We've seen that as a simple ROS 2 package, `vadd_publisher` runs perfectly fine and meets the 10 Hz publishing objective. But what if the `vadd` has bigger vectors, or has operations that involve many more iterations? This second section explores this with a more computationally expensive publisher, the `doublevadd_publisher`. The source code of the new `vadd` function is presented below:

```
/*
        ____  ____
       /   /\/   /
      /___/  \  /   Copyright (c) 2021, Xilinx®.
      \   \   \/    Author: Víctor Mayoral Vilches <victorma@xilinx.com>
       \   \
       /   /
      /___/   /\
      \   \  /  \
       \___\/\___\

Inspired by the Vector-Add example.
See https://github.com/Xilinx/Vitis-Tutorials/blob/master/
Getting_Started/Vitis

*/

#define DATA_SIZE 4096
// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

extern "C" {
    void vadd(
            const unsigned int *in1,  // Read-Only Vector 1
            const unsigned int *in2,  // Read-Only Vector 2
            unsigned int *out,        // Output Result
            int size                  // Size in integer
            )
    {
        for (int j = 0; j < size; ++j) {  // stupidly iterate over
                                          // it to generate load
        #pragma HLS loop_tripcount min = c_size max = c_size
            for (int i = 0; i < size; ++i) {
            #pragma HLS loop_tripcount min = c_size max = c_size
              out[i] = in1[i] + in2[i];
            }
        }
    }
}
```

Note how instead of one `for` loop, we now have two, simulating a more complex computation. Let's try this out in hardware. Provided that the image was previously created, we just need to replace the ROS 2 workspace, the rest should be identical. You can do this in various ways (physically mounting the raw image in the SD card, `scp`-ing the ROS 2 workspace, etc.).

Briefly, in the workstation:

```
# generate the workspace with doublevadd_publisher (if exists already, add
to it)
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select ament_vitis doublevadd_publisher

# copy to rootfs in SD card, e.g.
$ scp -r install-kv260/* petalinux@192.168.1.86:/ros2_ws/
```

```
# Launch doublevadd_publisher
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation

$ . /ros2_ws/local_setup.bash     # source the ROS 2 overlay workspace we
just
                                  # created. Note it has been copied to the SD
                                  # card image while being created.

$ ros2 topic hz /vector --window 10 &
$ ros2 run doublevadd_publisher doublevadd_publisher

...
[INFO] [1629656740.225647854] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 11'
[INFO] [1629656740.675023646] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 12'
[INFO] [1629656741.124260679] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 13'
average rate: 2.226
    min: 0.449s max: 0.449s std dev: 0.00011s window: 10
[INFO] [1629656741.573462323] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 14'
[INFO] [1629656742.022713366] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 15'
[INFO] [1629656742.471917079] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 16'
average rate: 2.226
    min: 0.449s max: 0.449s std dev: 0.00011s window: 10
[INFO] [1629656742.921175382] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 17'
[INFO] [1629656743.370423695] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 18'
[INFO] [1629656743.819651828] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 19'
average rate: 2.226
    min: 0.449s max: 0.449s std dev: 0.00010s window: 10
[INFO] [1629656744.268874211] [doublevadd_publisher]: Publishing: 'vadd
finished, iteration: 20'
...
```

This new publisher achieves only `2.2 Hz`, quite far from the `10 Hz` targeted. Using hardware acceleration, future examples will demonstrate how to build a custom compute pipeline that offloads computations to a kernel. *If you wish to jump directly into hardware acceleration with `doublevadd_publisher`, head to: 3. Offloading ROS 2 publisher*.

1. Hello Xilinx

	Source code
`publisher_xilinx`	
publisher	`member_function_publisher.cpp`

This example lets you experience KRS further, walking you through the process of building and launching a ROS 2 package across different targets: in the workstation, in the real hardware and in an emulation.

Warning

The examples assume you've already installed KRS. If not, refer to install.

Tip

Learn ROS 2 before trying this out first.

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash
```

Source code of the publisher

```cpp
/*
     ____  ____
    /   /\/   /
    /___/  \  /   Copyright (c) 2021, Xilinx®.
    \   \   \/    Author: Víctor Mayoral Vilches
<victorma@xilinx.com>
    \   \
    /   /
    /___/   /\
    \   \  /  \
    \___\/\___\

*/

#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to
register a
* member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
public:
MinimalPublisher()
: Node("minimal_publisher"), count_(0)
{
    publisher_ = this-
>create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
    500ms, std::bind(&MinimalPublisher::timer_callback, this));
}

private:
void timer_callback()
{
    auto message = std_msgs::msg::String();
    message.data = "Hello, Xilinx! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'",
message.data.c_str());
    publisher_->publish(message);
}
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
size_t count_;
};

int main(int argc, char * argv[])
{
rclcpp::init(argc, argv);
rclcpp::spin(std::make_shared<MinimalPublisher>());
rclcpp::shutdown();
return 0;
}
```

Launch `hello_xilinx` example in the workstation

```
$ source install/setup.bash  # source the overlay workspace
$ ros2 run publisher_xilinx member_function_publisher
[INFO] [1618407842.800443167] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 0'
[INFO] [1618407843.300407127] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 1'
[INFO] [1618407843.800389187] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 2'
...
```

Launch `hello_xilinx` example in the KV260 hardware

First, let's select the firmware for the target hardware, `KV260`:

```
$ colcon acceleration select kv260
```

To verify that we indeed that the right firmware selected, look for the one marked with a "*" at the end of its name:

```
$ colcon acceleration list
kv260*
```

Let's now build the package targeting the KV260:

```
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select publisher_xilinx
```

Let's now create a raw disk image for the SD card with PetaLinux's rootfs, a vanilla Linux 5.4.0 kernel and the ROS 2 overlay workspace we've just created for the KV260:

```
$ colcon acceleration linux vanilla --install-dir install-kv260
```

We're now ready to run it on hardware. For that, we need to flash the `~/krs_ws/acceleration/firmware/select/sd_card.img` file into the SD card. If you need help doing so, check out these instructions for different OSs. **Make sure to flash `~/krs_ws/acceleration/firmware/select/sd_card.img` we just generated, and not some other image**.

Once flashed, connect the board to the computer via its USB/UART/JTAG FTDI adapter and power it on. Then, launch your favority serial console (e.g. `sudo tio /dev/ttyUSB1`). You should get to the following prompt where you can use the `petalinux` username and define your own password:

```
PetaLinux 2020.2.2 xilinx-k26-starterkit-2020_2.2 ttyPS0

xilinx-k26-starterkit-2020_2 login:
```

To launch the `hello_xilinx` package:

```
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation
$ . /ros2_ws/local_setup.bash   # source the ROS 2 overlay workspace
$ ros2 run publisher_xilinx member_function_publisher  # launch the
hello_xilinx example

[INFO] [1618478074.116887661] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 0'
[INFO] [1618478074.611878275] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 1'
[INFO] [1618478075.112175121] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 2'
...
```

Launch `hello_xilinx` example in an emulated KV260

First, let's select the firmware for the target hardware, `KV260`:

```
$ colcon acceleration select kv260
```

Build the package for the KV260:

```
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select publisher_xilinx
```

Create a raw disk image for the SD card with PetaLinux's rootfs and a Linux 5.4.0 vanilla kernel:

```
$ colcon acceleration linux vanilla
```

Then use the `acceleration` extensions to the ROS 2 `colcon` meta-build system to launch an emulation:

```
$ colcon acceleration emulation sw_emu install-kv260
```

This will trigger the emulation with a prompt like what follows:

Warning

To be fixed in future releases.

Due to some issues with KV260, the current emulation capabilities only brings you to ramfs. To finalize booting the board it's necessary to manually introduce the following in ramfs:

```
mkdir /configfs
mount -t configfs configfs /configfs
cd /configfs/device-tree/overlays/
mkdir full
mkdir /lib/firmware/
cp /boot/devicetree/zynqmp-sck-kv-g-qemu.dtbo /lib/firmware/.
echo -n "zynqmp-sck-kv-g-qemu.dtbo" > full/path
exec /init
```

```
$ colcon acceleration emulation sw_emu install-kv260
SECURITY WARNING: This class invokes explicitly a shell via the shell=True
argument of the Python subprocess library, and uses admin privileges to
manage raw disk images. It is the user's responsibility to ensure that all
whitespace and metacharacters passed are quoted appropriately to avoid shell
injection vulnerabilities.
- Verified that install/ is available in the current ROS 2 workspace
- Confirmed availability of raw image file at: /home/xilinx/krs_ws/
acceleration/firmware/select/sd_card.img
- Finished inspecting raw image, obtained UNITS and STARTSECTOR P1/P2
- Image mounted successfully at: /tmp/sdcard_img_p2
- Successfully cleaned up prior overlay ROS 2 workspace at: /tmp/
sdcard_img_p2/ros2_ws
- Copied 'install-kv260' directory as a ROS 2 overlay workspace in the raw
image.
- Umounted the raw image.
- Generated PMU and QEMU files.
- Launching emulation...
cd /home/xilinx/krs_ws/acceleration/firmware/select/emulation && /tools/
Xilinx/Vitis/2020.2/bin/launch_emulator -device-family ultrascale -target
sw_emu -qemu-args-file /home/xilinx/krs_ws/acceleration/firmware/select/
emulation/qemu_args.txt -pmc-args-file /home/xilinx/krs_ws/acceleration/
firmware/select/emulation/pmu_args.txt -sd-card-image /home/xilinx/krs_ws/
acceleration/firmware/select/sd_card.img -enable-prep-target $*
Running SW Emulation
INFO :  [LAUNCH_EMULATOR] pl_sim_dir option is not provided.
Starting QEMU
 - Press <Ctrl-a h> for help
Waiting for QEMU to start. qemu_port 9720
Qemu_pids 601788 601789
qemu-system-aarch64: -chardev socket,path=./qemu-rport-_pmu@0,server,id=pmu-
apu-rp: info: QEMU waiting for connection on: disconnected:unix:./qemu-rport-
_pmu@0,server
QEMU started. qemu_pid=601788.
Waiting for PMU to start. Qemu_pids 601792 601793
qemu-system-aarch64: -chardev socket,id=pl-
rp,host=127.0.0.1,port=8500,server: info: QEMU waiting for connection on:
```

```
disconnected:tcp:127.0.0.1:8500,server
PMC started. pmc_pid=601792
Starting PL simulation.Generating PLLauncher commandline
running PLL Launcher
PMU Firmware 2020.2 Jun 10 2021   22:45:10
PMU_ROM Version: xpbr-v8.1.0-0
NOTICE:  ATF running on XCZUUNKN/QEMU v4/RTL0.0 at 0xfffea000
NOTICE:  BL31: v2.2(release):xilinx-v2020.2.2-k26
NOTICE:  BL31: Built : 22:39:14, Jun 10 2021


U-Boot 2020.01 (Jun 11 2021 - 08:39:52 +0000)

Model: ZynqMP SMK-K26 Rev1/B/A
Board: Xilinx ZynqMP
DRAM:  4 GiB
PMUFW:  v1.1
EL Level:   EL2
Chip ID:    unknown
NAND:  0 MiB
MMC:   mmc@ff170000: 1
In:    serial@ff010000
Out:   serial@ff010000
Err:   serial@ff010000
Bootmode: JTAG_MODE
Reset reason:
Net:   No ethernet found.
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc1 is current device
Scanning mmc 1:1...
Found U-Boot script /boot.scr
2145 bytes read in 17 ms (123 KiB/s)
## Executing script at 20000000
17932800 bytes read in 3251 ms (5.3 MiB/s)
38045 bytes read in 26 ms (1.4 MiB/s)
25150797 bytes read in 4541 ms (5.3 MiB/s)
## Loading init Ramdisk from Legacy Image at 04000000 ...
   Image Name:   petalinux-initramfs-image-zynqmp
   Image Type:   AArch64 Linux RAMDisk Image (uncompressed)
   Data Size:    25150733 Bytes = 24 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 00100000
   Booting using the fdt blob at 0x100000
   Loading Ramdisk to 77803000, end 78fff50d ... OK
   Loading Device Tree to 000000000fff3000, end 000000000ffff49c ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 5.4.0-xilinx-v2020.2 (oe-user@oe-host) (gcc
version 9.2.0 (GCC)) #1 SMP Thu Jun 10 22:03:38 UTC 2021
[    0.000000] Machine model: ZynqMP SMK-K26 Rev1/B/A
[    0.000000] earlycon: cdns0 at MMIO 0x00000000ff010000 (options
'115200n8')
```

```
[    0.000000] printk: bootconsole [cdns0] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 1000 MiB at 0x0000000039000000
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.1
[    0.000000] percpu: Embedded 22 pages/cpu s49880 r8192 d32040 u90112
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] CPU features: detected: ARM erratum 845719
[    0.000000] CPU features: detected: ARM erratum 843419

...

ccpath is incorrect: /sys/bus/i2c/devices/*51/eeprom
[   11.106002] random: python3: uninitialized urandom read (24 bytes read)

ccpath is incorrect: /sys/bus/i2c/devices/*51/eeprom
SOM: CARRIER_CARD: REVISION:
NO CARRIER DTBO FOUND, PLEASE CHECK /boot/devicetree/
Waiting for /dev/mmcblk0p2 to pop up (attempt 1)
Waiting for /dev/mmcblk0p2 to pop up (attempt 2)
Waiting for /dev/mmcblk0p2 to pop up (attempt 3)
Waiting for /dev/mmcblk0p2 to pop up (attempt 4)
Waiting for /dev/mmcblk0p2 to pop up (attempt 5)
Waiting for /dev/mmcblk0p2 to pop up (attempt 6)
Waiting for /dev/mmcblk0p2 to pop up (attempt 7)
Waiting for /dev/mmcblk0p2 to pop up (attempt 8)
Waiting for /dev/mmcblk0p2 to pop up (attempt 9)
Waiting for /dev/mmcblk0p2 to pop up (attempt 10)
Device /dev/mmcblk0p2 not found
ERROR: There's no '/dev' on rootfs.

sh: can't access tty; job control turned off
/ #
```

To finalize the boot on rootfs, introduce the following:

```
mkdir /configfs
mount -t configfs configfs /configfs
cd /configfs/device-tree/overlays/
mkdir full
mkdir /lib/firmware/
cp /boot/devicetree/zynqmp-sck-kv-g-qemu.dtbo /lib/firmware/.
echo -n "zynqmp-sck-kv-g-qemu.dtbo" > full/path
exec /init
```

This should bring you all the way down to the prompt:

```
PetaLinux 2020.2.2 xilinx-k26-starterkit-2020_2.2 ttyPS0

xilinx-k26-starterkit-2020_2 login:
```

Use the `petalinux` username and pick you own password. Then, inside of the emulation, launch `hello_xilinx`:

```
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation
$ . /ros2_ws/local_setup.bash   # source the ROS 2 overlay workspace
$ ros2 run publisher_xilinx member_function_publisher  # launch the
hello_xilinx example
[INFO] [1618478074.116887661] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 0'
[INFO] [1618478074.611878275] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 1'
[INFO] [1618478075.112175121] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 2'
...
```

To exit the emulation, type `Ctrl-A x`.

Bonus: Launch `hello_xilinx` example in emulation with another firmware

As introduced in Features, KRS is designed to be easily portable across boards. To demonstrate it, we'll now fetch the firmware for another board capable of hardware acceleration and launch `hello_xilinx` in this board using emulation. Let's start fetching the firmware:

```
$ git clone --recursive https://gitlab.com/xilinxrobotics/ros2/
acceleration_firmware_zcu102 src/acceleration/acceleration_firmware_zcu102
```

We then build the workspace to deploy the firmware:

```
# build the workspace
$ colcon build --merge-install  # about 2 mins
```

Then, we select the firmware and build the workspace for the ZCU102:

```
$ colcon acceleration select zcu102
$ colcon build --build-base=build-zcu102 --install-base=install-zcu102 --
merge-install --mixin zcu102 --packages-select publisher_xilinx
```

We create new raw disk image for the SD card (using ZCU102's firmware) with PetaLinux's rootfs and a Linux 5.4.0 vanilla kernel:

```
$ colcon acceleration linux vanilla --install-dir install-zcu102
```

Then use the `acceleration` extensions to the ROS 2 `colcon` meta-build system to launch an emulation:

```
$ colcon acceleration emulation sw_emu --no-install
```

The ZCU102 rootfs should login automatically with the `root` username. Inside of the emulation, launch `hello_xilinx`:

```
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation
$ . /ros2_ws/local_setup.bash   # source the ROS 2 overlay workspace
$ ros2 run publisher_xilinx member_function_publisher  # launch the
hello_xilinx example
[INFO] [1618478074.116887661] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 0'
[INFO] [1618478074.611878275] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 1'
[INFO] [1618478075.112175121] [minimal_publisher]: Publishing: 'Hello,
Xilinx! 2'
...
```

To exit the emulation, type `Ctrl-A x`.

Tip

You can also try the `hw_emu` target in the ZCU102.

2. HLS in ROS 2

	Source code
`simple_adder`	
adder1	
kernel 1	`adder1.cpp`
testbench 1	`testbench1.cpp`
adder2	
kernel 2	`adder2.cpp`
testbench 2	`testbench2.cpp`

HLS is at the core of C++ hardware acceleration. KRS allows to maintain a ROS 2-centric view while leveraging HLS, optimizing the flow and empowering roboticists toperform everything from the CLI, attaching nicely to the ROS 2 meta build system (`ament`) and the ROS 2 meta build tools (`colcon`).

This example demonstrates how KRS helps to transition from the Xilinx's Vitis-centric flow to the ROS 2 flow. The source code performs a trivial add operation. HLS is used directly from the ROS 2 CLI build tools. We explore the possiblities of offloading this operation to the Programmable Logic (PL). The latencies derived from the offloading operation are then analyzed both with the ROS 2 CLI tooling, and later, with the Vitis HLS GUI.

Though we'll be targeting KV260 hardware platform, no physical hardware is needed for this example.

Warning

The examples assume you've already installed KRS. If not, refer to install.

Tip

Learn ROS 2 before trying this out first.

Prepare the environment and fetch the example

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash
```

A bit of background on HLS, RTL and FPGAs

About C simulation

HLS C Simulation performs a pre-synthesis validation and checks that that the C program correctly implements the required functionality. This allows you to compile, simulate, and debug the C/C++ algorithm. C simulation allows for the function to be synthesized (e.g. `simple_adder` above) to be validated with a test bench using C simulation. A C test bench includes a main() top-level function, that calls the function to be synthesized by the Vitis HLS project. The test bench can also include other functions.

Refer to Verifying Code with C Simulation for more.

About Synthesis

Synthesis transforms the C or C++ function into RTL code for acceleration in programmable logic. HLS allows you to control synthesis process using optimization pragmas to create high-performance implementations.

Refer to Synthesizing the Code for more.

A further understanding of the *synthesis summary view*

The *synthesis summary view* provides lots of information with different fields which may not always be self-explanatory. The list below tries to clarify some of the most relevant ones:

- **Slack**: displays any timing issues in the implementation.
- **Latency(cycles)/Latency(ns)**: displays the number of cycles it takes to produce the output. The same information is also displayed in ns.
- **Iteration Latency**: latency of a single iteration for a loop.
- **Interval**: Interval or Initiation Interval (II) is the number of clock cycles before new inputs can be applied.
- **Trip Count**: displays the number of iterations of a specific loop in the implemented hardware. This reflects any unrolling of the loop in hardware.

A quick review of the resources in an FPGA

Xilinx FPGA designs contain the following core components: Configurable Logic Blocks (CLBs), Programmable Routing Blocks (PRBs), I/O Blocks (IOBs) and Digital Signal Processors (DSPs). Within CLBs we have 4 sub-components: Flip-flops (FFs), Loop-up-tables (LUTs), multiplexers (MUXs) and SRAM.

When accounting for resources consumed, we focus particularly on the following which are further defined below:

- **DSP**: DSPs have been optimized to implement various common digital signal processing functions with maximum performance and minimum logic resource utilization. They have functions to provide multipliers, adders, substractors, accumulators, coefficient register storage or a summation unit, amongst others.
- **FF**: they are the smallest storage resource on the FPGA and each FF within a CLB is a binary register used to save logic states between clock cycles on an FPGA circuit.
- **LUT**: stores a predefined list of outputs for every combination of inputs. LUTs provide a fast way to retrieve the output of a logic operation because possible results are stored and then referenced, rather than calculated.
- **URAM**: UltraRAM is a large, lightweight memory block intended to allow the replacement of off-board memories enabling better overall performance.

`simple_adder1`: a quick look into the flow

`simple_adder` is pretty straightforward:

```
int simple_adder(int a, int
b) {
    int c;
    c = a + b;
    return c;
}
```

We can build this example packaged as a ROS package (though it doesn't interact with the computational graph) as follows:

```
$ colcon acceleration select kv260  # select the KV260 firmware
$ colcon build  --merge-install --build-base=build-kv260 --install-
base=install-kv260 --mixin kv260 --packages-select simple_adder
```

The `simple_adder` package uses the `vitis_hls_generate_tcl` macro on its CMakeLists.txt. This macro commands the build system (`ament`) that when the package is built it should generate a Tcl script for `simple_adder`. This Tcl script will allow architects with HLS experience to use the traditional HLS interface to further optimize and investigate the kernel through C simulation, synthesis, RTL simulation, etc.

Here's a peek to the macro syntax used:

```
vitis_hls_generate_tc
  PROJECT

project_simpleadder1
  SRC
    src/adder1.cpp
  HEADERS
    include
  TESTBENCH
    src/
testbench.cpp
  TOPFUNCTION
    simple_adder
  CLOCK
    4
  SYNTHESIS
)
```

The Xilinx Tcl traditional path

When using the `vitis_hls_generate_tcl` macro above, a script is automatically generated under `~/krs_ws/build-kv60/simple_adder/project_simpleadder1.tcl`:

```
open_project -reset project_simpleadder1
add_files  /home/xilinx/ros2_ws/src/xilinx/xilinx_examples/simple_adder/
src/adder1.cpp
add_files -tb /home/xilinx/ros2_ws/src/xilinx/xilinx_examples/
simple_adder/src/testbench.cpp -cflags "-isystem  /home/xilinx/ros2_ws/
src/xilinx/xilinx_examples/simple_adder/include"
set_top simple_adder
# solution_4ns
open_solution solution_4ns
set_part {xczu9eg-ffvb1156-2-e}
create_clock -period 4
csim_design -ldflags "-lOpenCL" -profile
csynth_design

# solution_10ns
open_solution solution_10ns
set_part {xczu9eg-ffvb1156-2-e}
create_clock -period 10
csim_design -ldflags "-lOpenCL" -profile
csynth_design

exit
```

The resulting Tcl script can directly be launched with `vitis_hls -f <file-name>` allowing to perform C simulation and synthesis (in this particular case).

```
$ vitis_hls -f project_simpleadder1.tcl

****** Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2020.2.2
(64-bit)
**** SW Build 3118627 on Tue Feb  9 05:13:49 MST 2021
**** IP Build 3115676 on Tue Feb  9 10:48:11 MST 2021
    ** Copyright 1986-2021 Xilinx, Inc. All Rights Reserved.

source /tools/Xilinx/Vitis_HLS/2020.2/scripts/vitis_hls/hls.tcl -notrace
INFO: [HLS 200-10] Running '/tools/Xilinx/Vitis_HLS/2020.2/bin/unwrapped/
lnx64.o/vitis_hls'
INFO: [HLS 200-10] For user 'xilinx' on host 'xilinx' (Linux_x86_64
version 5.10.37-rt39-tsn-measurements) on Sat Jul 10 07:39:11 CEST 2021
INFO: [HLS 200-10] On os Ubuntu 20.04.2 LTS
INFO: [HLS 200-10] In directory '/home/xilinx/ros2_ws/build-zcu102/
simple_adder'
Sourcing Tcl script 'project_simpleadder1.tcl'
INFO: [HLS 200-1510] Running: open_project -reset project_simpleadder1
INFO: [HLS 200-10] Opening and resetting project '/home/xilinx/ros2_ws/
build-zcu102/simple_adder/project_simpleadder1'.
WARNING: [HLS 200-40] No /home/xilinx/ros2_ws/build-zcu102/simple_adder/
project_simpleadder1/solution_4ns/solution_4ns.aps file found.
WARNING: [HLS 200-40] No /home/xilinx/ros2_ws/build-zcu102/simple_adder/
project_simpleadder1/solution_10ns/solution_10ns.aps file found.
INFO: [HLS 200-1510] Running: add_files /home/xilinx/ros2_ws/src/xilinx/
xilinx_examples/simple_adder/src/adder1.cpp
```

```
INFO: [HLS 200-10] Adding design file '/home/xilinx/ros2_ws/src/xilinx/
xilinx_examples/simple_adder/src/adder1.cpp' to the project
INFO: [HLS 200-1510] Running: add_files -tb /home/xilinx/ros2_ws/src/
xilinx/xilinx_examples/simple_adder/src/testbench.cpp -cflags -isystem  /
home/xilinx/ros2_ws/src/xilinx/xilinx_examples/simple_adder/include
INFO: [HLS 200-10] Adding test bench file '/home/xilinx/ros2_ws/src/
xilinx/xilinx_examples/simple_adder/src/testbench.cpp' to the project
INFO: [HLS 200-1510] Running: set_top simple_adder
INFO: [HLS 200-1510] Running: open_solution solution_4ns
INFO: [HLS 200-10] Creating and opening solution '/home/xilinx/ros2_ws/
build-zcu102/simple_adder/project_simpleadder1/solution_4ns'.
INFO: [HLS 200-1505] Using default flow_target 'vivado'
Resolution: For help on HLS 200-1505 see www.xilinx.com/cgi-bin/docs/rdoc?
v=2020.2;t=hls+guidance;d=200-1505.html
INFO: [HLS 200-1510] Running: set_part xczu9eg-ffvb1156-2-e
INFO: [HLS 200-10] Setting target device to 'xczu9eg-ffvb1156-2-e'
INFO: [HLS 200-1510] Running: create_clock -period 4
INFO: [SYN 201-201] Setting up clock 'default' with a period of 4ns.
INFO: [HLS 200-1510] Running: csim_design -ldflags -lOpenCL -profile
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch CLANG as the compiler.
Compiling ../../../../../../src/xilinx/xilinx_examples/simple_adder/src/
testbench.cpp in debug mode
Compiling ../../../../../../src/xilinx/xilinx_examples/simple_adder/src/
adder1.cpp in debug mode
Generating csim.exe
Expected result: 100, Got Result: 100
Expected result: 103, Got Result: 103
Expected result: 106, Got Result: 106
Expected result: 109, Got Result: 109
Expected result: 112, Got Result: 112
Expected result: 115, Got Result: 115
Expected result: 118, Got Result: 118
Expected result: 121, Got Result: 121
Expected result: 124, Got Result: 124
Expected result: 127, Got Result: 127
Generating dot files
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time: 0.65
seconds. CPU system time: 0.31 seconds. Elapsed time: 0.8 seconds;
current allocated memory: 195.056 MB.
INFO: [HLS 200-1510] Running: csynth_design
INFO: [HLS 200-111] Finished File checks and directory preparation: CPU
user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0 seconds;
current allocated memory: 195.260 MB.
INFO: [HLS 200-10] Analyzing design file '/home/xilinx/ros2_ws/src/xilinx/
xilinx_examples/simple_adder/src/adder1.cpp' ...
INFO: [HLS 200-111] Finished Source Code Analysis and Preprocessing: CPU
user time: 0.13 seconds. CPU system time: 0.09 seconds. Elapsed time: 0.
23 seconds; current allocated memory: 195.911 MB.
INFO: [HLS 200-777] Using interface defaults for 'Vivado' flow target.
INFO: [HLS 200-111] Finished Compiling Optimization and Transform: CPU
user time: 2.84 seconds. CPU system time: 0.26 seconds. Elapsed time: 3.
23 seconds; current allocated memory: 196.203 MB.
INFO: [HLS 200-111] Finished Checking Pragmas: CPU user time: 0 seconds.
CPU system time: 0 seconds. Elapsed time: 0 seconds; current allocated
memory: 196.220 MB.
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms: CPU user time: 0.01
seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds; current
```

```
allocated memory: 197.187 MB.
INFO: [HLS 200-10] Checking synthesizability ...
INFO: [HLS 200-111] Finished Checking Synthesizability: CPU user time: 0.
01 seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds;
current allocated memory: 196.621 MB.
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU
user time: 0.03 seconds. CPU system time: 0.01 seconds. Elapsed time: 0.
04 seconds; current allocated memory: 216.648 MB.
INFO: [HLS 200-111] Finished Architecture Synthesis: CPU user time: 0.02
seconds. CPU system time: 0 seconds. Elapsed time: 0.02 seconds; current
allocated memory: 208.313 MB.
INFO: [HLS 200-10] Starting hardware synthesis ...
INFO: [HLS 200-10] Synthesizing 'simple_adder' ...
INFO: [HLS 200-10]
------------------------------------------------------------
INFO: [HLS 200-42] -- Implementing module 'simple_adder'
INFO: [HLS 200-10]
------------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-11] Finished scheduling.
INFO: [HLS 200-111] Finished Scheduling: CPU user time: 0.01 seconds. CPU
system time: 0 seconds. Elapsed time: 0.02 seconds; current allocated
memory: 208.520 MB.
INFO: [BIND 205-100] Starting micro-architecture generation ...
INFO: [BIND 205-101] Performing variable lifetime analysis.
INFO: [BIND 205-101] Exploring resource sharing.
INFO: [BIND 205-101] Binding ...
INFO: [BIND 205-100] Finished micro-architecture generation.
INFO: [HLS 200-111] Finished Binding: CPU user time: 0.01 seconds. CPU
system time: 0 seconds. Elapsed time: 0.01 seconds; current allocated
memory: 208.601 MB.
INFO: [HLS 200-10]
------------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'simple_adder'
INFO: [HLS 200-10]
------------------------------------------------------------
INFO: [RTGEN 206-500] Setting interface mode on port 'simple_adder/a' to
'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on port 'simple_adder/b' to
'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'simple_adder'
to 'ap_ctrl_hs'.
INFO: [RTGEN 206-100] Finished creating RTL model for 'simple_adder'.
INFO: [HLS 200-111] Finished Creating RTL model: CPU user time: 0.01
seconds. CPU system time: 0.01 seconds. Elapsed time: 0.01 seconds;
current allocated memory: 208.723 MB.
INFO: [HLS 200-111] Finished Generating all RTL models: CPU user time: 0.
8 seconds. CPU system time: 0.01 seconds. Elapsed time: 0.82 seconds;
current allocated memory: 214.826 MB.
INFO: [VHDL 208-304] Generating VHDL RTL for simple_adder.
INFO: [VLOG 209-307] Generating Verilog RTL for simple_adder.
INFO: [HLS 200-789] **** Estimated Fmax: 984.25 MHz
INFO: [HLS 200-111] Finished Command csynth_design CPU user time: 3.9
seconds. CPU system time: 0.38 seconds. Elapsed time: 4.43 seconds;
current allocated memory: 215.042 MB.
INFO: [HLS 200-1510] Running: open_solution solution_10ns
INFO: [HLS 200-10] Creating and opening solution '/home/xilinx/ros2_ws/
build-zcu102/simple_adder/project_simpleadder1/solution_10ns'.
INFO: [HLS 200-1505] Using default flow_target 'vivado'
Resolution: For help on HLS 200-1505 see www.xilinx.com/cgi-bin/docs/rdoc?
v=2020.2;t=hls+guidance;d=200-1505.html
```

```
INFO: [HLS 200-1510] Running: set_part xczu9eg-ffvb1156-2-e
INFO: [HLS 200-1510] Running: create_clock -period 10
INFO: [SYN 201-201] Setting up clock 'default' with a period of 10ns.
INFO: [HLS 200-1510] Running: csim_design -ldflags -lOpenCL -profile
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch CLANG as the compiler.
Compiling ../../../../../../src/xilinx/xilinx_examples/simple_adder/src/
testbench.cpp in debug mode
Compiling ../../../../../../src/xilinx/xilinx_examples/simple_adder/src/
adder1.cpp in debug mode
Generating csim.exe
Expected result: 100, Got Result: 100
Expected result: 103, Got Result: 103
Expected result: 106, Got Result: 106
Expected result: 109, Got Result: 109
Expected result: 112, Got Result: 112
Expected result: 115, Got Result: 115
Expected result: 118, Got Result: 118
Expected result: 121, Got Result: 121
Expected result: 124, Got Result: 124
Expected result: 127, Got Result: 127
Generating dot files
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time: 0.45
seconds. CPU system time: 0.25 seconds. Elapsed time: 0.71 seconds;
current allocated memory: 204.178 MB.
INFO: [HLS 200-1510] Running: csynth_design
INFO: [HLS 200-111] Finished File checks and directory preparation: CPU
user time: 0.01 seconds. CPU system time: 0 seconds. Elapsed time: 0.01
seconds; current allocated memory: 204.280 MB.
INFO: [HLS 200-10] Analyzing design file '/home/xilinx/ros2_ws/src/xilinx/
xilinx_examples/simple_adder/src/adder1.cpp' ...
INFO: [HLS 200-111] Finished Source Code Analysis and Preprocessing: CPU
user time: 0.12 seconds. CPU system time: 0.1 seconds. Elapsed time: 0.23
seconds; current allocated memory: 204.290 MB.
INFO: [HLS 200-777] Using interface defaults for 'Vivado' flow target.
INFO: [HLS 200-111] Finished Compiling Optimization and Transform: CPU
user time: 2.96 seconds. CPU system time: 0.28 seconds. Elapsed time: 3.
41 seconds; current allocated memory: 204.363 MB.
INFO: [HLS 200-111] Finished Checking Pragmas: CPU user time: 0 seconds.
CPU system time: 0 seconds. Elapsed time: 0 seconds; current allocated
memory: 204.364 MB.
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms: CPU user time: 0
seconds. CPU system time: 0 seconds. Elapsed time: 0 seconds; current
allocated memory: 205.081 MB.
INFO: [HLS 200-10] Checking synthesizability ...
INFO: [HLS 200-111] Finished Checking Synthesizability: CPU user time: 0.
01 seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds;
current allocated memory: 204.483 MB.
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU
user time: 0.04 seconds. CPU system time: 0.01 seconds. Elapsed time: 0.
04 seconds; current allocated memory: 224.451 MB.
INFO: [HLS 200-111] Finished Architecture Synthesis: CPU user time: 0.01
seconds. CPU system time: 0 seconds. Elapsed time: 0.02 seconds; current
allocated memory: 216.047 MB.
INFO: [HLS 200-10] Starting hardware synthesis ...
INFO: [HLS 200-10] Synthesizing 'simple_adder' ...
INFO: [HLS 200-10]
----------------------------------------------------------------
```

```
INFO: [HLS 200-42] -- Implementing module 'simple_adder'
INFO: [HLS 200-10]
---------------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-11] Finished scheduling.
INFO: [HLS 200-111] Finished Scheduling: CPU user time: 0.01 seconds. CPU
system time: 0 seconds. Elapsed time: 0.02 seconds; current allocated
memory: 216.082 MB.
INFO: [BIND 205-100] Starting micro-architecture generation ...
INFO: [BIND 205-101] Performing variable lifetime analysis.
INFO: [BIND 205-101] Exploring resource sharing.
INFO: [BIND 205-101] Binding ...
INFO: [BIND 205-100] Finished micro-architecture generation.
INFO: [HLS 200-111] Finished Binding: CPU user time: 0.01 seconds. CPU
system time: 0 seconds. Elapsed time: 0.01 seconds; current allocated
memory: 216.140 MB.
INFO: [HLS 200-10]
---------------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'simple_adder'
INFO: [HLS 200-10]
---------------------------------------------------------------
INFO: [RTGEN 206-500] Setting interface mode on port 'simple_adder/a' to
'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on port 'simple_adder/b' to
'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'simple_adder'
to 'ap_ctrl_hs'.
INFO: [RTGEN 206-100] Finished creating RTL model for 'simple_add<er'.
INFO: [HLS 200-111] Finished Creating RTL model: CPU user time: 0.01
seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds; current
allocated memory: 216.195 MB.
INFO: [HLS 200-111] Finished Generating all RTL models: CPU user time: 0.
75 seconds. CPU system time: 0.02 seconds. Elapsed time: 0.77 seconds;
current allocated memory: 216.821 MB.
INFO: [VHDL 208-304] Generating VHDL RTL for simple_adder.
INFO: [VLOG 209-307] Generating Verilog RTL for simple_adder.
INFO: [HLS 200-789] **** Estimated Fmax: 984.25 MHz
INFO: [HLS 200-111] Finished Command csynth_design CPU user time: 3.96
seconds. CPU system time: 0.41 seconds. Elapsed time: 4.57 seconds;
current allocated memory: 216.911 MB.
INFO: [HLS 200-112] Total CPU user time: 11.55 seconds. Total CPU system
time: 1.84 seconds. Total elapsed time: 12.47 seconds; peak allocated
memory: 224.451 MB.
INFO: [Common 17-206] Exiting vitis_hls at Sat Jul 10 07:39:24 2021...
```

With KRS, we can avoid using Tcl and use instead the ROS 2 extensions to `colcon` to easily perform C simulation, synthesis, implementation and more. Before running things, ROS 2 tools will dump the following status:

```
$ colcon acceleration hls simple_adder
 Project:  project_simpleadder2
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder2
    - Solution:  solution_4ns
        - C Simulation:              Not Run
        - C Synthesis:               Not Run
        - C/RTL Co-simulation:       Not Run
        - Export:
```

```
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
   - Solution:  solution_10ns
       - C Simulation:               Not Run
       - C Synthesis:                Not Run
       - C/RTL Co-simulation:        Not Run
       - Export:
           - IP Catalog:         Not Run
           - System Generator:   Not Run
           - Export Evaluation:  Not Run
 Project:  project_simpleadder1
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder1
   - Solution:  solution_4ns
       - C Simulation:               Not Run
       - C Synthesis:                Not Run
       - C/RTL Co-simulation:        Not Run
       - Export:
           - IP Catalog:         Not Run
           - System Generator:   Not Run
           - Export Evaluation:  Not Run
```

Let's run it and get some results:

```
$ colcon acceleration hls simple_adder --run
Found Tcl script "project_simpleadder2.tcl" for package: simple_adder
Executing /home/xilinx/krs_ws/build-kv260/simple_adder/
project_simpleadder2.tcl
Found Tcl script "project_simpleadder1.tcl" for package: simple_adder
Executing /home/xilinx/krs_ws/build-kv260/simple_adder/
project_simpleadder1.tcl
 Project:  project_simpleadder2
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder2
   - Solution:  solution_4ns
       - C Simulation:               Pass
       - C Synthesis:                Run
       - C/RTL Co-simulation:        Not Run
       - Export:
           - IP Catalog:         Not Run
           - System Generator:   Not Run
           - Export Evaluation:  Not Run
   - Solution:  solution_10ns
       - C Simulation:               Pass
       - C Synthesis:                Run
       - C/RTL Co-simulation:        Not Run
       - Export:
           - IP Catalog:         Not Run
           - System Generator:   Not Run
           - Export Evaluation:  Not Run
 Project:  project_simpleadder1
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder1
   - Solution:  solution_4ns
       - C Simulation:               Pass
       - C Synthesis:                Run
       - C/RTL Co-simulation:        Not Run
       - Export:
```

```
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
```

If we want to inspect the results from the CLI, we can add the `--synthesis-report` flag:

```
$ colcon acceleration hls simple_adder --synthesis-report

...

  Project:  project_simpleadder1
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder1
    - Solution:  solution_4ns
        - C Simulation:               Pass
        - C Synthesis:                Run
        - C/RTL Co-simulation:        Not Run
        - Export:
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
        - Synthesis report: /home/xilinx/krs_ws/build-kv260/simple_adder/
project_simpleadder1/solution_4ns/syn/report/simple_adder_csynth.rpt


            ================================================================
            == Vitis HLS Report for 'simple_adder'
            ================================================================
            * Date:           Sun Aug 22 15:58:31 2021

            * Version:        2020.2.2 (Build 3118627 on Tue Feb  9 05:13:49
MST 2021)
            * Project:        project_simpleadder1
            * Solution:       solution_4ns (Vitis Kernel Flow Target)
            * Product family: zynquplus
            * Target device:  xck26-sfvc784-2LV-c


            ================================================================
            == Performance Estimates
            ================================================================
            + Timing:
                * Summary:
                +--------+---------+----------+------------+
                |  Clock |  Target | Estimated| Uncertainty|
                +--------+---------+----------+------------+
                |ap_clk  |  4.00 ns|  2.016 ns|     1.08 ns|
                +--------+---------+----------+------------+

            + Latency:
                * Summary:
                +---------+---------+----------+----------+-----+-----
+---------+
                |  Latency (cycles) |  Latency (absolute) |  Interval |
Pipeline|
                |   min   |   max   |    min   |    max   | min | max |
Type  |
```

```
                +---------+---------+----------+----------+-----+-----
+---------+
                |        0|        0|      0 ns|      0 ns|    1|    1|
none|
                +---------+---------+----------+----------+-----+-----
+---------+

                + Detail:
                    * Instance:
                    N/A

                    * Loop:
                    N/A



            ================================================================
            == Utilization Estimates
            ================================================================
            * Summary:
            +-----------------+---------+------+--------+--------+-----+
            |       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
            +-----------------+---------+------+--------+--------+-----+
            |DSP              |        -|     -|       -|       -|    -|
            |Expression       |        -|     -|       0|      41|    -|
            |FIFO             |        -|     -|       -|       -|    -|
            |Instance         |        0|     -|     144|     232|    -|
            |Memory           |        -|     -|       -|       -|    -|
            |Multiplexer      |        -|     -|       -|      18|    -|
            |Register         |        -|     -|      37|       -|    -|
            +-----------------+---------+------+--------+--------+-----+
            |Total            |        0|     0|     181|     291|    0|
            +-----------------+---------+------+--------+--------+-----+
            |Available        |      288|  1248|  234240|  117120|   64|
            +-----------------+---------+------+--------+--------+-----+
            |Utilization (%)  |        0|     0|      ~0|      ~0|    0|
            +-----------------+---------+------+--------+--------+-----+
...
```

Results show how the `simple_adder` function can be implemented with only a few LUTs and using only one single cycle (0 is counted as the first cycle). The FPGA can be programmed to run with a clock at `2.016 ns` (estimated). This means that **by using solely 291 LUTs and 181 FFs of the FPGA to build specialized circuitry, we can get deterministic responses when launching the `simple_adder` function with a maximum response period of (`2.016 ns` (estimated) + `1.08 ns` (uncertainty)). In other words, `3.09600` ns**.

Let's compare this with the results we may get if we were to run this (only) on the PS system witout relying on the FPGA. The Zynq UltraScale+ MPSoC targeted uses Quad-core Arm Cortex-A53 which has a CPU frequency of up to 1.5 GHz. Since the clock of the processor can theoretically run at 1.5 GHz, this means that one cycle has a period of approximately `0.666 ns`. If the CPU were to be able to fit the whole function into one cycle, this would is indeed better than the `3.09600 ns` we estimated before when using the FPGA, however, things aren't that easy.

Neither Von-Neumann-based CPU architectures can fit all operations into one cycle nor adaptive SoCs like the Zynq UltraScale+ MPSoC can leverage hardware acceleration without considering the interfacing with the FPGA. The following considerations should be taken into account:

1. CPUs control-driven machine model is based on a token of control, which indicates when a statement should be executed. This gives CPUs full control to implement easily complex data and control structures however, this also comes at the cost of being less efficient since every operation needs to push data in-and-out of ALUs (each operation needs to be managed in this control flow mechanism). Ultimately this leads to multiple cycles for even the simplest operations like what's illustrated in `simple_adder` above.
2. Besides the additional cycles required for the sole computation, the PS deals with tons of complex aspects that may very easily interrupt the computation, produce a switch of context and get back to the computation of `simple_adder` afterwards. Even in a somewhat ideal scenario, with a soft/firm real-time operating system running on the PS, the kernel can incur on delays of several tenths of microseconds.
3. FPGAs offer a deterministic response which can be specially exploited when relying onits resources, e.g. while driving I/O directly. In adaptive SoCs, interfacing the Processing System (PS, the CPU), with the Programmable Logic (PL, the FPGA) has a time/cycle cost which should be considered. If the output of the function/kernel is to return to the PS, one should account for the complexity of such function/kernel. In very simple examples like the `simple_adder` above, this cost of interfacing PS-PL is generally much bigger than the optimizations one could get and the determinism is essentially degraded due to the involvement of the CPU.

Using the Vitis HLS GUI

The project files generated by ROS 2 CLI tools can also be opened with `vitis_hls` GUI by pointing to the new project folder created. Within the Vitis HLS GUI, the various reports generated can be inspected graphically:

(*click on the images to see them big*)

Image	Comments
	The *synthesis summary view* shows that the target clock is `10ns`, as specified in the first solution in the Tcl script above. Note however that the synthesized clock ends up being much lower. The *Performance & Resource Estimates* section summarizes that overall timing characteristics. Note that the timing characteristics show a 0 latency

Image	Comments
	The *schedule viewer view* shows how the two read operation are executed in the same clock and then get fed into the add operation. Everything gets executed in the same cycle. The *pre-synthesis control flow view* shows that this function has a trivial control flow. The *synthesis details view* shows a summary of the latency and also the resources consumed to synthesis the function. In this case only 39 LUTs.

The data that we just inspected through Vitis HLS GUI is available in reports which can be parsed and exposed in a CLI interface. KRS does exactly this. KRS provides a series of CLI verbs and subverbs that allow to fetch this information directly from the CLI allowing ROS developers to create their own development flows.

Let's now complicate a bit more the `simple_adder` function and see how faster clocks aren't alwasy better. Specially, we show how FPGAs can be optimized to fit operations in less cycles delivering lower latencies.

`simple_adder2`: optimizing FPGA synthesis for lower latency responses

The source code of `simple_adder2` will now be the following:

simple_adder source code

```
int simple_adder(int a, int
b) {
    int c;
    c = a*a*a + b*b;
    return c;
}
```

The CMakeLists.txt file uses now a different testbench and kernel source code:

```
vitis_hls_generate_tc
  PROJECT

project_simpleadder2
  SRC
    src/adder2.cpp
  HEADERS
    include
  TESTBENCH
    src/
testbench2.cpp
  TOPFUNCTION
    simple_adder
  CLOCK
    4 5 6 7 8 9 10
  SYNTHESIS
)
```

Note that the macro will generate one solution per each `CLOCK` (in ns) argument provided.

Let's compare the *Schedule Viewer* of the `4 ns` and `10 ns` clock solutions. *These were generated with a different firmware (`zcu102`). Similar ones can be also obtained with the KV260 board. See Hello Xilinx example if you need to recap on how to switch between multiple firmware options ;)*.

(*Click on each image to see it big*)

Image	Comments
	Targeting `4 ns` clock. Note the whole operation takes 4 cycles.
	Targeting `10 ns` clock. Note the whole operation takes 2 cycles.

Futher inspecting the solutions with `colcon` CLI extensions:

```
$ colcon krs hls simple_adder --synthesis-report
 Project:  project_simpleadder2
 Path:  /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder2
    - Solution:  solution_4ns
        - C Simulation:               Pass
        - C Synthesis:                Run
        - C/RTL Co-simulation:        Not Run
        - Export:
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
        - Synthesis report: /home/xilinx/krs_ws/build-kv260/simple_adder/
project_simpleadder2/solution_4ns/syn/report/simple_adder_csynth.rpt
```

```
            ================================================================
            == Vitis HLS Report for 'simple_adder'
            ================================================================
            * Date:           Sun Aug 22 15:58:17 2021

            * Version:        2020.2.2 (Build 3118627 on Tue Feb  9 05:13:49
MST 2021)
            * Project:        project_simpleadder2
            * Solution:       solution_4ns (Vitis Kernel Flow Target)
            * Product family: zynquplus
            * Target device:  xck26-sfvc784-2LV-c


            ================================================================
            == Performance Estimates
            ================================================================
            + Timing:
                * Summary:
                +--------+---------+----------+------------+
                |  Clock |  Target | Estimated| Uncertainty|
                +--------+---------+----------+------------+
                |ap_clk  |  4.00 ns|  2.365 ns|     1.08 ns|
                +--------+---------+----------+------------+

            + Latency:
                * Summary:
                +---------+---------+-----------+-----------+-----+-----
+---------+
                |  Latency (cycles) |   Latency (absolute)  |  Interval |
Pipeline|
                |   min   |   max   |    min    |    max    | min | max |
Type  |
                +---------+---------+-----------+-----------+-----+-----
+---------+
                |        5|        5|  20.000 ns|  20.000 ns|    6|    6|
none|
                +---------+---------+-----------+-----------+-----+-----
+---------+

                + Detail:
                    * Instance:
                    N/A

                    * Loop:
                    N/A



            ================================================================
            == Utilization Estimates
            ================================================================
            * Summary:
            +-----------------+---------+------+--------+--------+-----+
            |       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
            +-----------------+---------+------+--------+--------+-----+
            |DSP              |        -|     -|       -|       -|    -|
            |Expression       |        -|     -|       0|      41|    -|
```

```
            |FIFO             |        -|     -|       -|       -|    -|
            |Instance         |        0|     0|     639|     379|    -|
            |Memory           |        -|     -|       -|       -|    -|
            |Multiplexer      |        -|     -|       -|      46|    -|
            |Register         |        -|     -|     106|       -|    -|
            +-----------------+---------+------+--------+--------+-----+
            |Total            |        0|     0|     745|     466|    0|
            +-----------------+---------+------+--------+--------+-----+
            |Available        |      288|  1248|  234240|  117120|   64|
            +-----------------+---------+------+--------+--------+-----+
            |Utilization (%)  |        0|     0|      ~0|      ~0|    0|
            +-----------------+---------+------+--------+--------+-----+

...

    - Solution:  solution_10ns
        - C Simulation:               Pass
        - C Synthesis:                Run
        - C/RTL Co-simulation:        Not Run
        - Export:
            - IP Catalog:         Not Run
            - System Generator:   Not Run
            - Export Evaluation:  Not Run
        - Synthesis report: /home/xilinx/krs_ws/build-kv260/simple_adder/
project_simpleadder2/solution_10ns/syn/report/simple_adder_csynth.rpt


            ================================================================
            == Vitis HLS Report for 'simple_adder'
            ================================================================
            * Date:           Sun Aug 22 15:58:23 2021

            * Version:        2020.2.2 (Build 3118627 on Tue Feb  9 05:13:49
MST 2021)
            * Project:        project_simpleadder2
            * Solution:       solution_10ns (Vitis Kernel Flow Target)
            * Product family: zynquplus
            * Target device:  xck26-sfvc784-2LV-c


            ================================================================
            == Performance Estimates
            ================================================================
            + Timing:
                * Summary:
                +--------+----------+----------+------------+
                |  Clock |  Target  | Estimated| Uncertainty|
                +--------+----------+----------+------------+
                |ap_clk  |  10.00 ns|  5.436 ns|     2.70 ns|
                +--------+----------+----------+------------+

            + Latency:
                * Summary:
                +---------+---------+-----------+-----------+-----+-----
+---------+
                |  Latency (cycles) |   Latency (absolute)  |  Interval |
Pipeline|
```

```
                |   min   |   max   |    min     |    max     | min | max |
Type  |
                +---------+---------+------------+------------+-----+-----
+---------+
                |        1|        1|  10.000 ns|  10.000 ns|    2|    2|
none|
                +---------+---------+------------+------------+-----+-----
+---------+

                + Detail:
                    * Instance:
                    N/A

                    * Loop:
                    N/A



            ================================================================
            == Utilization Estimates
            ================================================================
            * Summary:
            +-----------------+---------+------+--------+--------+-----+
            |       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
            +-----------------+---------+------+--------+--------+-----+
            |DSP              |        -|     -|       -|       -|    -|
            |Expression       |        -|     -|       0|      41|    -|
            |FIFO             |        -|     -|       -|       -|    -|
            |Instance         |        0|     0|     144|     292|    -|
            |Memory           |        -|     -|       -|       -|    -|
            |Multiplexer      |        -|     -|       -|      32|    -|
            |Register         |        -|     -|      70|       -|    -|
            +-----------------+---------+------+--------+--------+-----+
            |Total            |        0|     0|     214|     365|    0|
            +-----------------+---------+------+--------+--------+-----+
            |Available        |      288|  1248|  234240|  117120|   64|
            +-----------------+---------+------+--------+--------+-----+
            |Utilization (%)  |        0|     0|      ~0|      ~0|    0|
            +-----------------+---------+------+--------+--------+-----+

...
```

We observe how using a target `10 ns` clock (which is slower than `4 ns`) leads to a) the use of less LUT and FF resources and b) a lower latency (due to a smaller number of cycles required). It's pretty interesting to note that getting a higher frequency in the clock does not necesarily mean we'll obtain a lower period for the function. This happens very clearly in this case.

`colcon` CLI tooling also allows to obtain a quick summary of all the solutions to evaluate time and use of resources:

```
$ colcon acceleration hls simple_adder --summary
# /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder2.tcl
Solution#   tar.clk est.clk     latency_max BRAM_18K    DSP FF      LUT
```

```
solution_10ns    10.00    5.436       10.000      0 (0%)      0 (0%)  214 (~0%)
365 (~0%)
solution_4ns     4.00     2.365       20.000      0 (0%)      0 (0%)  745 (~0%)
466 (~0%)
# /home/xilinx/krs_ws/build-kv260/simple_adder/project_simpleadder1.tcl
Solution#    tar.clk est.clk     latency_max BRAM_18K    DSP FF      LUT
solution_4ns     4.00     2.016       0       0 (0%)      0 (0%)  181 (~0%)
291 (~0%)
```

3. Offloading ROS 2 publisher

	Source code
`offloaded_doublevadd_publisher`	
kernel	`vadd.cpp`
publisher	`accelerated_doublevadd_publisher.cpp`

This example leverages KRS to offload the `vadd` function operations to the FPGA, showing how easy it is for ROS package maintainers to extend their packages to include hardware acceleration, and create deterministic kernels. The objective is to publish the resulting vector at 10 Hz. This example builds on top of a prior one:

- 0. ROS 2 publisher - `doublevadd_publisher`, which runs completely on the scalar quad-core Cortex-A53 Application Processing Units (APUs) of the KV260 and is only able to publish at `2.2 Hz`.

The source code is taken from `doublevadd_publisher` purposely *as is*, and HLS transforms the C++ code directly to RTL, creating a dedicated hardware circuit in the form of a kernel that offloads the CPU from the heavy vadd computations and provides deterministic responses.

Though deterministic, the resulting kernel computation is slower than its CPUs counterpart. The reason behind this is that the code is taken *as is*, and the kernel doesn't really exploit any parallelism, nor optimizes the computation flow. Given that the kernel clock (`4 ns`) is slower much than the one of the Arm CPUs, this leads altogether to an actual worse performance than the only-CPU case previously studied at 0. ROS 2 publisher.

The ultimate objective of this example is to illustrate roboticists how **more deterministic (connected to real-time) does not necessarily lead to faster (lower latency) or better performance, quite the opposite**. Future examples will demonstrate how to achieve both, determinism and low latency.

Warning

The examples assume you've already installed KRS. If not, refer to install.

> **Tip**
>
> Learn ROS 2 before trying this out first.

`offloaded_doublevadd_publisher`

The kernel is identitical to the one presented in 0. ROS 2 publisher.

```
/*
        ____  ____
       /   /\/   /
      /___/  \  /   Copyright (c) 2021, Xilinx®.
      \   \   \/    Author: Víctor Mayoral Vilches <victorma@xilinx.com>
       \   \
       /   /
      /___/   /\
      \   \  /  \
       \___\/\___\

Inspired by the Vector-Add example.
See https://github.com/Xilinx/Vitis-Tutorials/blob/master/
Getting_Started/Vitis

*/

#define DATA_SIZE 4096
// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

extern "C" {
    void vadd(
            const unsigned int *in1,  // Read-Only Vector 1
            const unsigned int *in2,  // Read-Only Vector 2
            unsigned int *out,        // Output Result
            int size                  // Size in integer
            )
    {
        for (int j = 0; j < size; ++j) {  // stupidly iterate over
                                          // it to generate load
        #pragma HLS loop_tripcount min = c_size max = c_size
            for (int i = 0; i < size; ++i) {
            #pragma HLS loop_tripcount min = c_size max = c_size
              out[i] = in1[i] + in2[i];
            }
        }
    }
}
```

The only difference in this package is that it declares a kernel on its CMakeLists.txt file using the `vitis_acceleration_kernel` CMake macro:

```
# vadd kernel
vitis_acceleration_kern
  NAME vadd
  FILE src/vadd.cpp
  CONFIG src/
kv260.cfg
  INCLUDE
    include
  TYPE
    hw
  PACKAGE
)
```

Let's build it:

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# if not done before, fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace to deploy KRS components
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash

# select kv260 firmware (in case you've been experimenting with something
else)
$ colcon acceleration select kv260

# build offloaded_doublevadd_publisher
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select ament_vitis ros2acceleration
offloaded_doublevadd_publisher

# copy to KV260 rootfs, e.g.
$ scp -r install-kv260/* petalinux@192.168.1.86:/ros2_ws/
```

Since this package contains a kernel and we're using the Vitis `hw` build target (*more on Vitis build targets in future tutorials*), it'll take a bit longer to build the package. In an *AMD Ryzen 5 PRO 4650G* andit took **14 minutes**.

Note also the process is *slightly different* this time since we have an acceleration kernel. Before launching the binary in the CPUs, we need to load the kernel in the FPGA. For that, we'll be using some of the extensions KRS provides to the ROS 2 CLI tooling, particularly the `ros2 acceleration` suite:

Warning

While you can re-arrange permissions and execute the following with the `petalinux` user, the simplest way forward is to execute as root.

```
$ sudo su
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation

$ . /ros2_ws/local_setup.bash     # source the ROS 2 overlay workspace we
just
                                  # created. Note it has been copied to the
SD
                                  # card image while being created.

# restart the daemon that manages the acceleration kernels
$ ros2 acceleration stop; ros2 acceleration start

# list the accelerators
$ ros2 acceleration list
                                      Accelerator           Type    Active
                                         kv260-dp       XRT_FLAT         1
                                             base       XRT_FLAT         0
                     offloaded_doublevadd_publisher       XRT_FLAT         0

# select the offloaded_doublevadd_publisher
$ ros2 acceleration select offloaded_doublevadd_publisher

# launch binary
$ cd /ros2_ws/lib/offloaded_doublevadd_publisher
$ ros2 topic hz /vector_acceleration --window 10 &
$ ros2 run offloaded_doublevadd_publisher offloaded_doublevadd_publisher

[INFO] [1629663768.633315230] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 6'
[INFO] [1629663769.150109773] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 7'
average rate: 1.935
    min: 0.517s max: 0.517s std dev: 0.00010s window: 7
[INFO] [1629663769.666922955] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 8'
[INFO] [1629663770.183640105] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 9'
average rate: 1.935
    min: 0.517s max: 0.517s std dev: 0.00010s window: 9
[INFO] [1629663770.700318913] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 10'
[INFO] [1629663771.217068001] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 11'
average rate: 1.935
    min: 0.517s max: 0.517s std dev: 0.00010s window: 10
[INFO] [1629663771.733872538] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 12'
[INFO] [1629663772.250599612] [accelerated_doublevadd_publisher]:
```

```
Publishing: 'vadd finished, iteration: 13'
...
```

The publishing rate is `1.935 Hz`, which is lower than the `2.2 Hz` obtained in 0. ROS 2 publisher. As introduced before and also in example 2. HLS in ROS 2, the rationale behind this is a combination of two aspects: - First, the CPU clock is generally faster than the FPGA one, which means that pure offloading of operations (unless dataflow is optimized) are deterministic, but most of the time subject to be coherent with the slower clock. - Second, the computation needs to be adapted to the dataflow and parallelism exploited (if available).

4. Accelerated ROS 2 publisher

	Source code
`accelerated_doublevadd_publisher`	
kernel	`vadd.cpp`
publisher	`accelerated_doublevadd_publisher.cpp`

This example leverages KRS to offload and accelerate the `vadd` function operations to the FPGA, showing how easy it is for ROS package maintainers to extend their packages, include hardware acceleration and create deterministic kernels. The objective is to publish the resulting vector at 10 Hz. This example builds on top of two prior ones:

- 3. Offloading ROS 2 publisher - `offloaded_doublevadd_publisher`, which offloads the `vadd` operation to the FPGA and leads to a deterministic vadd operation, yet insuficient overall publishing rate of `1.935 Hz`.
- 0. ROS 2 publisher - `doublevadd_publisher`, which runs completely on the scalar quad-core Cortex-A53 Application Processing Units (APUs) of the KV260 and is only able to publish at `2.2 Hz`.

This example upgrades the previous offloading operation at 3. Offloading ROS 2 publishe - `offloaded_doublevadd_publisher`, and includes an optimization for the dataflow. This allows the publisher to improve its publishing rate from 2 Hz, up to 6 Hz. For that, the HLS `INTERFACE` pragma is used. The HLS INTERFACE specifies how RTL ports are created from the function definition during interface synthesis. Sharing ports helps save FPGA resources by eliminating AXI interfaces, but it can limit the performance of the kernel because all the memory transfers have to go through a single port. The `m_axi` port has independent READ and WRITE channels, so with a single m_axi port, we can do reads and writes simultaneously but since the kernel (`vadd`) has two vectors from where its reading (simultaneously), we can optimize the dataflows by simply asking for an extra AXI interface.

After this dataflow optimization in the kernel, the `accelerated_doublevadd_publisher` ROS 2 package is able to publish at 6 Hz. *For a faster kernel that meets the 10 Hz goal, refer to 5. Faster ROS 2 publisher*.

Warning

The examples assume you've already installed KRS. If not, refer to install.

Tip

Learn ROS 2 before trying this out first.

accelerated_doublevadd_publisher

Let's take a look at the kernel source code this time:

```
/*
        ____  ____
       /   /\/   /
      /___/  \  /   Copyright (c) 2021, Xilinx®.
      \   \   \/    Author: Víctor Mayoral Vilches <victorma@xilinx.com>
       \   \
       /   /
      /___/   /\
      \   \  /  \
       \___\/\___\

Inspired by the Vector-Add example.
See https://github.com/Xilinx/Vitis-Tutorials/blob/master/
Getting_Started/Vitis

*/

#define DATA_SIZE 4096
// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

extern "C" {
    void vadd(
            const unsigned int *in1,  // Read-Only Vector 1
            const unsigned int *in2,  // Read-Only Vector 2
            unsigned int *out,        // Output Result
            int size                  // Size in integer
            )
    {
#pragma HLS INTERFACE m_axi port=in1 bundle=aximm1
#pragma HLS INTERFACE m_axi port=in2 bundle=aximm2
#pragma HLS INTERFACE m_axi port=out bundle=aximm1
        for (int j = 0; j < size; ++j) {  // stupidly iterate over
                                          // it to generate load
        #pragma HLS loop_tripcount min = c_size max = c_size
            for (int i = 0; i < size; ++i) {
            #pragma HLS loop_tripcount min = c_size max = c_size
              out[i] = in1[i] + in2[i];
            }
        }
    }
}
```

Note the aforementioned `INTERFACE` pragma as the only relevant change introduced, which specifies how RTL ports are created from the function definition during interface synthesis.

Short explanation of the `HLS INTERFACE` pragma

The parameters of the software functions defined in a HLS design are synthesized into ports in the RTL code [1] that group multiple signals to encapsulate the communication protocol between the HLS design and things external to the design. **The `HLS INTERFACE` specifies how RTL ports are created from the function definition during interface synthesis [2]. Sharing ports helps save FPGA resources by eliminating AXI interfaces, but it can limit the performance of the kernel because all the memory transfers have to go through a single port. The `m_axi` port has independent READ and WRITE channels, so with a single `m_axi` port, we can do reads and writes simultaneously but since we have two vectors from where we're reading (simultaneously), we can optimize the dataflows by simply asking for an extra AXI interface**.

Note the bandwidth and throughput of the kernel can be increased by creating multiple ports, using different bundle names, to connect multiple memory banks but this cames at the cost of resources in the PL fabric.

To understand this better, it's important to also understand that in RTL design, input and output operations must be performed through a port in the design interface and typically operate using a specific I/O (input-output) protocol. The implementation of a function-level protocol is indicated in `<mode>` after the `#pragma HLS INTERFACE <mode>`. In this case, the pragma is using the `m_axi` mode which corresponds with all ports as an AXI4 interface. The complete syntax of the pragma is as follows:

```
#pragma HLS interface <mode> port=<name> bundle=<string>
```

where[^8]:

- `<mode>` corresponds with the function-level protocol for the input/output operations through the RTL port
- `<port>` specifies the name of the function argument, return value or global variable the pragma applies to
- `<bundle>` groups function arguments into AXI interface ports. By default, HLS groups all function arguments specified as an AXI4 (`m_axi` mode) interface into a single AXI4 port. This option explicitly groups all interface ports with the same `bundle=<string>` into the same AXI interface port and names the RTL port the value specified by `<string>`.

In other words, the pragmas below define the `INTERFACE` standards for the RTL ports of the `vadd` function.

Let's build it:

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
```

```
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# if not done before, fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace to deploy KRS components
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash

# select kv260 firmware (in case you've been experimenting with something
else)
$ colcon acceleration select kv260

# build accelerated_doublevadd_publisher
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select ament_vitis ros2acceleration
accelerated_doublevadd_publisher

# copy to KV260 rootfs, e.g.
$ scp -r install-kv260/* petalinux@192.168.1.86:/ros2_ws/
```

and run it:

Troubleshooting loading an acceleration kernel with a long name

Due to a bug in the daemon client used underneath the ROS 2 CLI extensions, the following is likely to happen:

```
xilinx-k26-starterkit-2020_2:/lib/firmware/xilinx# ros2 acceleration
select accelerated_doublevadd_publisher
Removing accel /lib/firmware/xilinx/kv260-dp
*** buffer overflow detected ***: dfx-mgr-client terminated
```

This overflow is caused by a fixed buffer which is getting overflowed due to an accelerator name that's longer than expected. **This should be addressed in future firmware releases**.

For the time being, a quick patch involves changing the name of the accelerator manually and loading it with that different name. E.g.:

```
cd /lib/firmware/xilinx/
mv accelerated_doublevadd_publisher shorter
ros2 acceleration stop; ros2 acceleration start
ros2 acceleration select shorter
# and then launch the ROS 2 package as usual
```

```
$ sudo su
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation
```

```
$ . /ros2_ws/local_setup.bash     # source the ROS 2 overlay workspace we
just
                                  # created. Note it has been copied to the
SD
                                  # card image while being created.

# restart the daemon that manages the acceleration kernels
$ ros2 acceleration stop; ros2 acceleration start

# list the accelerators
$ ros2 acceleration list
                                        Accelerator           Type    Active
                  accelerated_doublevadd_publisher       XRT_FLAT         0
                                           kv260-dp       XRT_FLAT         1
                                               base       XRT_FLAT         0
                    offloaded_doublevadd_publisher       XRT_FLAT         0

# select the offloaded_doublevadd_publisher
# NOTE: see troubleshooting note above, implement
#       countermeasure if necessary
$ ros2 acceleration select accelerated_doublevadd_publisher

# launch binary
$ cd /ros2_ws/lib/accelerated_doublevadd_publisher
$ ros2 topic hz /vector_acceleration --window 10 &
$ ros2 run accelerated_doublevadd_publisher accelerated_doublevadd_publisher

ros2 run faster_doublevadd_publisher faster_doublevadd_publisher

...
[INFO] [1629667329.650547207] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 9'
[INFO] [1629667329.850627179] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 10'
[INFO] [1629667329.988200464] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 11'
[INFO] [1629667330.125859198] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 12'
[INFO] [1629667330.263443133] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 13'
[INFO] [1629667330.463512045] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 14'
[INFO] [1629667330.601060629] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 15'
average rate: 6.396
    min: 0.137s max: 0.200s std dev: 0.02870s window: 10
[INFO] [1629667330.738635863] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 16'
[INFO] [1629667330.876191868] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 17'
[INFO] [1629667331.076263260] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 18'
[INFO] [1629667331.213853284] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 19'
[INFO] [1629667331.351435218] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 20'
[INFO] [1629667331.488997143] [accelerated_doublevadd_publisher]:
```

```
Publishing: 'vadd finished, iteration: 21'
[INFO] [1629667331.689110265] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 22'
average rate: 6.397
    min: 0.137s max: 0.200s std dev: 0.02871s window: 10
[INFO] [1629667331.826752610] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 23'
[INFO] [1629667331.964359824] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 24'
[INFO] [1629667332.101934778] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 25'
[INFO] [1629667332.302034131] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 26'
[INFO] [1629667332.439673035] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 27'
[INFO] [1629667332.577249279] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 28'
[INFO] [1629667332.714848544] [accelerated_doublevadd_publisher]:
Publishing: 'vadd finished, iteration: 29'
average rate: 6.662
    min: 0.138s max: 0.200s std dev: 0.02507s window: 10
...
```

The optimizations in the dataflow introduced in the kernel via the use of the HLS `INTERFACE` pragma lead to a `6.3 Hz` publishing rate, which is about 3 times better than what was obtained before, but still insuficient to meet the `10 Hz` goal.

Next, in 5. Faster ROS 2 publisher, we'll review an even faster kernel that meets the 10 Hz publishing goal, by optimizing both the dataflow (as in this example) and exploiting `vadd` parallelism (via loop unrolling).

1. Managing Interface Synthesis. Retrieved from https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/managing_interface_synthesis.html#jro1585107736856
2. pragma HLS interface. Retrieved from https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/hls_pragmas.html#jit1504034365862.

5. Faster ROS 2 publisher

	Source code
`faster_doublevadd_publisher`	
kernel	`vadd.cpp`
publisher	`faster_doublevadd_publisher.cpp`

This example is the last one of the *ROS 2 publisher series*. It features a trivial vector-add ROS 2 publisher, which adds two vector inputs in a loop, and tries to publish the result at 10 Hz. This example will leverage KRS to produce an acceleration kernel that a) optimizes the dataflow and b) leverages parallelism via loop unrolling to meet the initial goal established by the `doublevadd_publisher`. Past examples of this series include:

- 4. Accelerated ROS 2 publisher - `offloaded_doublevadd_publisher`, which offloads and accelerates the `vadd` operation to the FPGA, optimizing the dataflow and leading to a deterministic vadd operation with an improved publishing rate of `6.3 Hz`.
- 3. Offloading ROS 2 publisher - `offloaded_doublevadd_publisher`, which offloads the `vadd` operation to the FPGA and leads to a deterministic vadd operation, yet insuficient overall publishing rate of `1.935 Hz`.
- 0. ROS 2 publisher - `doublevadd_publisher`, which runs completely on the scalar quad-core Cortex-A53 Application Processing Units (APUs) of the KV260 and is only able to publish at `2.2 Hz`.

Warning

The examples assume you've already installed KRS. If not, refer to install.

Tip

Learn ROS 2 before trying this out first.

accelerated_doublevadd_publisher

Let's take a look at the kernel source code first:

```
1   /*
2           ____  ____
3          /   /\/   /
4         /___/  \  /   Copyright (c) 2021, Xilinx®.
5         \   \   \/    Author: Víctor Mayoral Vilches <victorma@xilinx.com>
6          \   \
7          /   /
8         /___/   /\
9         \   \  /  \
10         \___\/\___\
11
12  Inspired by the Vector-Add example.
13  See https://github.com/Xilinx/Vitis-Tutorials/blob/master/
14  Getting_Started/Vitis
15
16  */
17
18  #define DATA_SIZE 4096
19  // TRIPCOUNT identifier
20  const int c_size = DATA_SIZE;
21
22  extern "C" {
23      void vadd(
24              const unsigned int *in1,  // Read-Only Vector 1
25              const unsigned int *in2,  // Read-Only Vector 2
26              unsigned int *out,        // Output Result
27              int size                  // Size in integer
28              )
29      {
30  #pragma HLS INTERFACE m_axi port=in1 bundle=aximm1
31  #pragma HLS INTERFACE m_axi port=in2 bundle=aximm2
32  #pragma HLS INTERFACE m_axi port=out bundle=aximm1
33
34          for (int j = 0; j <size; ++j) {  // stupidly iterate over
35                                           // it to generate load
36          #pragma HLS loop_tripcount min = c_size max = c_size
37              for (int i = 0; i<(size/16)*16; ++i) {
38              #pragma HLS UNROLL factor=16
39              #pragma HLS loop_tripcount min = c_size max = c_size
40                out[i] = in1[i] + in2[i];
41              }
42          }
43      }
    }
```

Besides the dataflow optimizations between input and output arguments in the PL-PS interaction, *line 37* utilizes a pragma to unroll the inner `for` loop by a factor of `16`, executing `16` sums *in parallel*, within the same clock cycle. The value of `16` is not arbitrary, but selected specifically to consume the whole bandwidth (`512-bits`) of the `m_axi` ports at each clock cycle

available after previous dataflow optimizations (see 4. Accelerated ROS 2 publisher to understand more about the dataflow optimizations). To fill in `512` bits, we pack together `16 unsigned int` inputs, each of `4 bytes` (\(\texttt{16 unsigned int} \cdot 4 \frac{bytes}{\texttt{unsigned int}} \cdot 8 \frac{bits}{byte} = 512 bits \)). Altogether, this leads to the most optimized form of the `vadd` kernel, *delivering both dataflow optimizations and code parallelism*, which is **able to successfully meet the publishing target of 10 Hz**. Overall, when compared to the initial example 0. ROS 2 publisher - `doublevadd_publisher`, the one presented in here obtains a **speedup of 5x** (*In fact, the speedup is higher than 5x however the ROS 2 `WallRate` instance is set to `100ms`, so the kernel is idle waiting for new data to arrive, discarding further acceleration opportunities.*).

Let's build it:

```
$ cd ~/krs_ws  # head to your KRS workspace

# prepare the environment
$ source /tools/Xilinx/Vitis/2020.2/settings64.sh  # source Xilinx tools
$ source /opt/ros/foxy/setup.bash  # Sources system ROS 2 installation
$ export PATH="/usr/bin":$PATH  # FIXME: adjust path for CMake 3.5+

# if not done before, fetch the source code of examples
$ git clone https://github.com/ros-acceleration/acceleration_examples src/
acceleration_examples

# build the workspace to deploy KRS components
$ colcon build --merge-install  # about 2 mins

# source the workspace as an overlay
$ source install/setup.bash

# select kv260 firmware (in case you've been experimenting with something
else)
$ colcon acceleration select kv260

# build accelerated_doublevadd_publisher
$ colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --packages-select ament_vitis ros2acceleration
faster_doublevadd_publisher

# copy to KV260 rootfs, e.g.
$ scp -r install-kv260/* petalinux@192.168.1.86:/ros2_ws/
```

and run it:

```
$ sudo su
$ source /usr/bin/ros_setup.bash  # source the ROS 2 installation

$ . /ros2_ws/local_setup.bash     # source the ROS 2 overlay workspace we
just
                                  # created. Note it has been copied to the
SD
                                  # card image while being created.
```

```
# restart the daemon that manages the acceleration kernels
$ ros2 acceleration stop; ros2 acceleration start

# list the accelerators
$ ros2 acceleration list
                                      Accelerator           Type    Active
                  accelerated_doublevadd_publisher       XRT_FLAT         0
                                           kv260-dp       XRT_FLAT         1
                                               base       XRT_FLAT         0
                    offloaded_doublevadd_publisher       XRT_FLAT         0
                       faster_doublevadd_publisher       XRT_FLAT         0

# select the faster_doublevadd_publisher
$ ros2 acceleration select faster_doublevadd_publisher

# launch binary
$ cd /ros2_ws/lib/faster_doublevadd_publisher
$ ros2 topic hz /vector_acceleration --window 10 &
$ ros2 run faster_doublevadd_publisher faster_doublevadd_publisher
INFO: Found Xilinx Platform
INFO: Loading 'vadd.xclbin'
[INFO] [1629669100.348149277] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 0'
[INFO] [1629669100.437331164] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 1'
[INFO] [1629669100.626349680] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 2'
[INFO] [1629669100.737320080] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 3'
[INFO] [1629669100.837296068] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 4'
[INFO] [1629669100.937279027] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 5'
[INFO] [1629669101.037308705] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 6'
[INFO] [1629669101.137309244] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 7'
[INFO] [1629669101.237301062] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 8'
[INFO] [1629669101.337311561] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 9'
[INFO] [1629669101.437294539] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 10'
[INFO] [1629669101.537323708] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 11'
average rate: 9.091
    min: 0.100s max: 0.189s std dev: 0.02659s window: 10
[INFO] [1629669101.637296386] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 12'
[INFO] [1629669101.737290495] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 13'
[INFO] [1629669101.837324683] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 14'
[INFO] [1629669101.937318152] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 15'
[INFO] [1629669102.037294040] [faster_doublevadd_publisher]: Publishing:
```

```
'vadd finished, iteration: 16'
[INFO] [1629669102.137285269] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 17'
[INFO] [1629669102.237289977] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 18'
[INFO] [1629669102.337286815] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 19'
[INFO] [1629669102.437316744] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 20'
[INFO] [1629669102.537339583] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 21'
[INFO] [1629669102.637276821] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 22'
average rate: 10.001
    min: 0.100s max: 0.100s std dev: 0.00007s window: 10
[INFO] [1629669102.737308289] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 23'
[INFO] [1629669102.837287528] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 24'
[INFO] [1629669102.937298656] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 25'
[INFO] [1629669103.037285145] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 26'
[INFO] [1629669103.137287133] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 27'
[INFO] [1629669103.237286742] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 28'
[INFO] [1629669103.337307070] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 29'
[INFO] [1629669103.437327789] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 30'
[INFO] [1629669103.537345337] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 31'
[INFO] [1629669103.637311146] [faster_doublevadd_publisher]: Publishing:
'vadd finished, iteration: 32'
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00004s window: 10
...
```

HOWTO

KV260

How do I update the KV260 firmware ?

According to https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1641152513/Kria+K26+SOM#Boot-FW-via-xmutil initial KV260 boards were shipped with a firmware that requires an update for `xmutil` tools to work appropriately. The corresponding firmware can be fetched from https://www.xilinx.com/member/forms/download/xef.html?filename=XilinxSom_QspiImage_v1.1_20210422.bin.

How do I emulate KV260 images?

In Vitis 2020.2 there's a bug which limits the emulation capabilities with KV260. The current images are able to boot only up until the ramfs and end up in something like:

```
...
SOM: CARRIER_CARD: REVISION:
NO CARRIER DTBO FOUND, PLEASE CHECK /boot/devicetree/
Waiting for /dev/mmcblk0p2 to pop up (attempt 1)
Waiting for /dev/mmcblk0p2 to pop up (attempt 2)
Waiting for /dev/mmcblk0p2 to pop up (attempt 3)
Waiting for /dev/mmcblk0p2 to pop up (attempt 4)
Waiting for /dev/mmcblk0p2 to pop up (attempt 5)
Waiting for /dev/mmcblk0p2 to pop up (attempt 6)
Waiting for /dev/mmcblk0p2 to pop up (attempt 7)
Waiting for /dev/mmcblk0p2 to pop up (attempt 8)
Waiting for /dev/mmcblk0p2 to pop up (attempt 9)
Waiting for /dev/mmcblk0p2 to pop up (attempt 10)
Device /dev/mmcblk0p2 not found
ERROR: There's no '/dev' on rootfs.
```

To finalize the boot, manually type the following:

```
mkdir /configfs
mount -t configfs configfs /configfs
cd /configfs/device-tree/overlays/
mkdir full
mkdir /lib/firmware/
cp /boot/devicetree/zynqmp-sck-kv-g-qemu.dtbo /lib/firmware/.
echo -n "zynqmp-sck-kv-g-qemu.dtbo" > full/path
exec /init
```

this should lead you all the way down to the prompt:

```
...
Starting system log daemon...0
```

```
Starting kernel log daemon...0
Starting crond: OK
Starting tcf-agent: OK
Starting TCG TSS2 Access Broker and Resource Management daemon: device
driver not loaded, skipping.

PetaLinux 2020.2.2 xilinx-k26-starterkit-2020_2.2 ttyPS0

xilinx-k26-starterkit-2020_2 login:
```

How do I configure the KV260 to JTAG boot mode?

The easiest way to do so is through a Tcl script and the Xilinx Software Commandline Tool (`xsct`). Connect the board to the computer via its USB/UART/JTAG FTDI adapter and power it on:

```
source /tools/Xilinx/Vitis/2020.2/settings64.sh  # path might be different
in your machine
cat << 'EOF' > som_bootmode.tcl

proc boot_jtag { } {
############################
# Switch to JTAG boot mode #
############################
targets -set -nocase -filter {name =~ "PSU"}
stop
# update multiboot to ZERO
mwr 0xffca0010 0x0
# change boot mode to JTAG
mwr 0xff5e0200 0x0100
# reset
rst -system

}
EOF

xsct
...

xsct%
xsct%
xsct%
xsct% source som_bootmode.tcl
xsct% connect
attempting to launch hw_server

****** Xilinx hw_server v2020.2.2
  **** Build date : Feb  9 2021 at 05:51:02
    ** Copyright 1986-2021 Xilinx, Inc. All Rights Reserved.

INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application

INFO: To connect to this hw_server instance use url: TCP:127.0.0.1:3121
```

```
tcfchan#0
xsct%
xsct% ta
  2  PS TAP
     3  PMU
     4  PL
  6  PSU
     7  RPU
        8  Cortex-R5 #0 (Halted)
        9  Cortex-R5 #1 (Lock Step Mode)
    10  APU
       11  Cortex-A53 #0 (Running)
       12  Cortex-A53 #1 (Power On Reset)
       13  Cortex-A53 #2 (Power On Reset)
       14  Cortex-A53 #3 (Power On Reset)
xsct%
xsct% boot_jtag
xsct% exit
```

That's it, now JTAG boot mode is set :).

I can't set JTAG boot mode because xsct is not behaving as expected?

If you're getting the following behavior:

```
xsct  # enter the Xilinx Software Command-Line Tool
xsct% source ./som_bootmode.tcl
xsct%
xsct%
xsct% connect
attempting to launch hw_server

****** Xilinx hw_server v2020.2.2
  **** Build date : Feb  9 2021 at 05:51:02
    ** Copyright 1986-2021 Xilinx, Inc. All Rights Reserved.

INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application

INFO: To connect to this hw_server instance use url: TCP:127.0.0.1:3121

tcfchan#0
xsct% ta
xsct% boot_jtag
no targets found with "name =~ "PSU"". available targets: none
xsct%
```

Chances are you've got the wrong drivers. Install the ones that come with Vivado as documented at https://forums.xilinx.com/t5/Xilinx-Evaluation-Boards/Unable-to-connect-to-ZCU104-with-Ubuntu-16-04LTS/td-p/889856:

```
cd /tools/Xilinx/Vivado/2020.2/data/xicom/cable_drivers/lin64/install_script/
install_drivers
sudo ./install_drivers
```

```
[sudo] password for xilinx:
INFO: Installing cable drivers.
INFO: Script name = ./install_drivers
INFO: HostName = xilinx
INFO: Current working dir = /tools/Xilinx/Vivado/2020.2/data/xicom/
cable_drivers/lin64/install_script/install_drivers
INFO: Kernel version = 5.10.37-rt39-tsn-measurements.
INFO: Arch = x86_64.
Successfully installed Digilent Cable Drivers
--File /etc/udev/rules.d/52-xilinx-ftdi-usb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-ftdi-usb.rules = 0000.
--Updating rules file.
--File /etc/udev/rules.d/52-xilinx-pcusb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-pcusb.rules = 0000.
--Updating rules file.

INFO: Digilent Return code = 0
INFO: Xilinx Return code = 0
INFO: Xilinx FTDI Return code = 0
INFO: Return code = 0
INFO: Driver installation successful.
CRITICAL WARNING: Cable(s) on the system must be unplugged then plugged back
in order for the driver scripts to update the cables.
```

How can I deal with "[XRT] ERROR: No devices found" issues when launching an accelerated app?

> **Note**
>
> This is caused by issues within the `2020.2.2` PetaLinux BSP release. `2021.1` should address these issues.

If when launching accelerated applications you observe the following

```
XRT build version: 2.8.0
Build hash: b94857f15ba8c8251df446e8c51af7e0a7c9e061
Build date: 2021-06-11 07:18:33
Git branch: 2020.2
PID: 1694
UID: 0
[Sat Jul 31 16:33:09 2021 GMT]
HOST: xilinx-k26-starterkit-2020_2.2
EXE: /ros2_ws/lib/faster_vadd_publisher/faster_vadd
[XRT] ERROR: No devices found
[XRT] ERROR: No devices found
[XRT] ERROR: No devices found
ERROR: Failed to find Xilinx platform
```

And `xbutil` shows an error like the following one:

```
xilinx-k26-starterkit-2020_2:/ros2_ws/lib/faster_vadd_publisher# xbutil list
INFO: Found total 1 card(s), 1 are usable
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
System Configuration
OS name:    Linux
Release:    5.4.0-xilinx-v2020.2
Version:    #1 SMP Thu Jun 10 22:03:38 UTC 2021
Machine:    aarch64
Glibc:      2.30
Distribution:   N/A
Now:        Sat Jul 31 16:40:06 2021 GMT
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XRT Information
Version:    2.8.0
Git Hash:   b94857f15ba8c8251df446e8c51af7e0a7c9e061
Git Branch: 2020.2
Build Date: 2021-06-11 07:18:33
ZOCL:       2.8.0,b94857f15ba8c8251df446e8c51af7e0a7c9e061
Failed to open device[0]
ERROR: Card index 0 is out of range
```

then, chances are your device tree does not include the corresponding `zocl` entry. A quick fix to to manually add it and rebuild the blob. To do so, get the sources from the device tree blob:

```
dtc -I dtb -O dts -o system.dts system.dtb
```

Edit `system.dts` and add the following:

```
                zyxclmm_drm {
                        compatible = "xlnx,zocl";
                        status = "okay";
                };
```

Build again the device tree into its blob:

```
dtc -I dts -O dtb -o system.dtb system.dts
```

After this, a good looking tree should have the following response:

```
xbutil list
INFO: Found total 1 card(s), 1 are usable
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
System Configuration
OS name:    Linux
Release:    5.4.0-xilinx-v2020.2
Version:    #1 SMP Thu Jun 10 22:03:38 UTC 2021
Machine:    aarch64
Glibc:      2.30
Distribution:   N/A
Now:        Wed Aug  4 14:49:47 2021 GMT
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XRT Information
```

```
Version:    2.8.0
Git Hash:   b94857f15ba8c8251df446e8c51af7e0a7c9e061
Git Branch: 2020.2
Build Date: 2021-06-11 07:18:33
ZOCL:       2.8.0,b94857f15ba8c8251df446e8c51af7e0a7c9e061
 [0]:edge
```

Basic embedded

How do I install a package built with PetaLinux in my embedded target?

You can build individual packages with PetaLinux using the following syntax:

```
petalinux-build -c <package>  # e.g. petalinux-build -c dfx-mgr
```

the resulting package will be built and archived, resulting in an `.rpm` file which you can often find at `build/tmp/deploy/rpm/aarch64/dfx-mgr*.rpm`.

We can install this in the embedded target as follows:

```
# host
scp build/tmp/deploy/rpm/aarch64/dfx-mgr*.rpm root@192.168.1.86:~/  # copy
them

# embedded target
rpm -i --force dfx-mgr-1.0-r0.aarch64.rpm  # force install to overwrite same
version
```

How do I get the sources (.dts) from a device tree blob (.dtb)?

```
dtc -I dtb -O dts -o system.dts system.dtb
```

How do I build a device tree blob (.dtb) from the sources (.dts)?

```
dtc -I dts -O dtb -o system.dtb system.dts
```

How do I get a better gdb debugging environment?

```
wget -P ~ https://git.io/.gdbinit
```

Get do I get a serial to ZCU10X boards?

Use a simple TTY terminal application. There're various you could use: `minicom`, `picocom`, etc.

I like tio:

```
sudo apt-get install tio
sudo tio /dev/ttyUSB0
```

Get HDMI in my ZCU10X boards?

Through the DisplayPort port. We support selected connectors. The following have been tested:

Product	Works
JSAUX JSESNZ4KDP2HDF	Yes
J5create JDA158	Yes
IVANKY-DP11	No
ICZI IZEC-A10-IT	No
Snowkids cable	No

How do I calculate the offsets for the raw `.img` images, so that I can mount them?

Offsets can be inspected in the following manner:

```
fdisk -l <path-to-img>/sd_card.img
Disk /home/xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img: 2.47
GiB, 2635071488 bytes, 5146624 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe29caf3d

Device                                          Boot   Start     End
Sectors  Size Id Type
/home/xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img1 *       2048
2000895 1998848  976M 83 Linux
/home/xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img2      2000896
5146623 3145728  1.5G 83 Linux
```

To calculate them, multiple the `End` by the `Units`. For example, for the second partition:

```
echo $(($(fdisk -l /home/xilinx/ros2_ws/acceleration/firmware/xilinx/
sd_card.img | grep 'img2' | awk '{print $2}') * $(fdisk -l /home/xilinx/
ros2_ws/acceleration/firmware/xilinx/sd_card.img | grep 'Units' | awk
'{print $8}')))
1024458752
```

How do I mount a `.img` file for inspection?

> **Note**
>
> The offsets need to be calculated first.

Raw image files contain several partitions. The first one is often the boot partition and the second one typically contains the file system. The following two commands will help mount the first two:

```
# mount boot partition
mkdir -p /tmp/sdcard_img_p1 && sudo mount -o loop,offset=1048576 /home/
xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img /tmp/sdcard_img_p1

# mount rootfs partition
mkdir -p /tmp/sdcard_img_p2 && sudo mount -o loop,offset=588251136 /home/
xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img /tmp/sdcard_img_p2
```

To umount:

```
sudo umount /tmp/sdcard_img_p1  # or *_p2, as appropriate
```

How do I create an initramfs from an rootfs partition?

See [32] for more details:

```
colcon krs mkinitramfs out.cpio.gz
```

Alternatively:

```
# mount sd card rootfs partition
sudo mount -o loop,offset=1024458752 /home/xilinx/ros2_ws/acceleration/
firmware/xilinx/sd_card.img /tmp/sdcard_img_p2
# create script, no variable expansion
cat << 'EOF' > mkinitramfs.sh
#!/bin/sh

# Copyright 2006 Rob Landley <rob@landley.net> and TimeSys Corporation.
# Licensed under GPL version 2

if [ $# -ne 2 ]
then
    echo "usage: mkinitramfs directory imagename.cpio.gz"
    exit 1
fi

if [ -d "$1" ]
then
    echo "creating $2 from $1"
```

```
    (cd "$1"; find . | cpio -o -H newc | gzip) > "$2"
else
    echo "First argument must be a directory"
    exit 1
fi
EOF
chmod +x mkinitramfs.sh
# create initramfs
sudo ./mkinitramfs.sh /tmp/sdcard_img_p2 test.cpio.gz
```

How do I handle corrupted sd_card.img files?

It's been observed that `sd_card.img` files get sometimes corrupted but neither `ddrescue` nor `testdisk` helped recovering them. The typicaly behavior observed in a corrupted image can be reproduced by trying to mount its second partition:

```
mkdir -p /tmp/sdcard_img_p2 && sudo mount -o loop,offset=1024458752 /home/
xilinx/ros2_ws/acceleration/firmware/xilinx/sd_card.img /tmp/sdcard_img_p2
[sudo] password for xilinx:
NTFS signature is missing.
Failed to mount '/dev/loop1': Invalid argument
The device '/dev/loop1' doesn't seem to have a valid NTFS.
Maybe the wrong device is used? Or the whole disk instead of a
partition (e.g. /dev/sda, not /dev/sda1)? Or the other way around?
```

Reproduce this behavior with krs hypervisor functionality

```
cd ~/ros2_ws/xilinx/firmware
tar -xzf sd_card.img.tar.gz  # decompress the image
cd ~/ros2_ws/
colcon krs hypervisor --dom0 vanilla --domU vanilla --ramdisk
initrd.cpio  # this works just fine, can mount afterwards
colcon krs hypervisor --dom0 vanilla --domU vanilla --ramdisk
initrd.cpio.gz  # this works just fine, can mount afterwards
colcon krs hypervisor --dom0 vanilla --domU vanilla --ramdisk
rootfs.cpio.gz  # this works just fine, can mount afterwards
colcon krs hypervisor --dom0 vanilla --domU vanilla --ramdisk test.cpio.gz
```

The current solution for this issue is to discard the sd_card.img file and create a new one from the `sd_card.img.tar.gz` file.

Note

This might be related to the fact that p1 in the sd_card is filling and overloading p2.

Extracting bitstream from xclbin file:

```
xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input vadd.xclbin
```

How do I figure out version of a kernel binary?

```
strings Image | grep "Linux version"
Linux version 5.4.0-xilinx-v2020.2 (oe-user@oe-host) (gcc version 9.2.0
(GCC)) #1 SMP Thu Jun 10 22:03:38 UTC 2021
Linux version %s (%s)
```

or more elaborated:

```
strings Image | grep "5\.[0123456789]\.[0123456789]" | grep "Linux version"
Linux version 5.4.0-xilinx-v2020.2 (oe-user@oe-host) (gcc version 9.2.0
(GCC)) #1 SMP Thu Jun 10 22:03:38 UTC 2021
```

How do I copy an image to the SD card in OS X from CLI?

```
sudo diskutil unmount /dev/rdisk2s1
pv sd_card.img | sudo dd of=/dev/rdisk2 bs=4m
```

How do I increase swap in Linux for Vitis/Vivado builds?

For a 30G swap:

```
sudo swapoff -a
sudo dd if=/dev/zero of=/swapfile bs=1G count=30
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

To fix things at boot, edit `/etc/fstab` and add:

```
/swapfile none swap sw 0 0
```

Network boot

TFTP server

```
cat << EOF > /etc/xinetd.d/tftp
service tftp
{
    protocol        = udp
    port            = 69
    socket_type     = dgram
    wait            = yes
    user            = xilinx
    server          = /usr/sbin/in.tftpd
    server_args     = /srv/tftp
```

```
    disable         = no
}
EOF
sudo systemctl status xinetd.service
setenv ipaddr 192.168.1.86
setenv serverip 192.168.1.33
tftpb 0xC00000 xen_boot_tftp.scr  # fetch one file in address 0xC00000
source 0xC00000
```

iPXE

iPXE is the "swiss army knife" of network booting. It supports both HTTPS and iSCSI. In addition, it has a script engine for fine grained control of the boot process and can provide a command shell. iPXE can be built as an EFI application (named snp.efi) which can be loaded and run by U-Boot:

```
# compile iPXE
git clone http://git.ipxe.org/ipxe.git
cd ipxe/src/
cat << EOF > myscript.ipxe
#!ipxe

dhcp
:loop
echo Hello world
goto loop
EOF
make CROSS_COMPILE=aarch64-linux-gnu- ARCH=arm64 bin-arm64-efi/snp.efi -j6
EMBED=myscript.ipxe

# copy to tftp folder
cp bin-arm64-efi/snp.efi /srv/tftp

# in the embedded board, fetch result
setenv ipaddr 192.168.1.86
setenv serverip 192.168.1.33
tftpb 0xC00000 snp.efi
bootefi 0xc00000
```

that'll get us something like:

```
ZynqMP> tftpb 0xC00000 snp.efi
Using ethernet@ff0e0000 device
TFTP from server 192.168.1.33; our IP address is 192.168.1.86
Filename 'snp.efi'.
Load address: 0xc00000
Loading: #########################################
     3.7 MiB/s
done
Bytes transferred = 197632 (30400 hex)
ZynqMP> bootefi 0xc00000
Scanning disk mmc@ff170000.blk...
Found 3 disks
efi_load_pe: Invalid DOS Signature
```

```
iPXE initialising devices...ok



iPXE 1.21.1+ (g3ae83) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE EFI Menu
Configuring (net0 00:0a:35:00:22:01)...... ok
Hello world
Hello world
Hello world
Hello world
Hello world
...
```

Alternatively, one could deposit the iPXE binary in the boot partition and create a boot script to automate it. The following example creates the boot script and fetches it over TFTP (it could also be pushed into the boot partition in which case will initiate automatically):

```
# create source file and compile it as a script for u-boot
cat << EOF > boot.source
load mmc 0:1 0xE00000 snp.efi
bootefi 0xE00000
EOF
mkimage -A arm -O linux -T script -C none -n "u-boot commands" -d
boot.source boot.scr
# copy to tftp directory and test it manually
cp boot.scr /srv/tftp

# within u-boot
tftpb 0xC00000 boot.scr
source 0xC00000
## Executing script at 00c00000
197632 bytes read in 39 ms (4.8 MiB/s)
Scanning disk mmc@ff170000.blk...
** Unrecognized filesystem type **
Found 3 disks
efi_load_pe: Invalid DOS Signature
iPXE initialising devices...ok



iPXE 1.21.1+ (g3ae83) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE EFI Menu
Configuring (net0 00:0a:35:00:22:01)...... ok
Hello world
Hello world
Hello world
Hello world
Hello world
...
```

How do I create boot scripts from their sources?

```
mkimage -c none -A arm -T script -d boot.source boot.scr
```

I can't boot my image because I get a kernel panic like the following one

```
[    4.549695] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[    4.550434] clk: Not disabling unused clocks
[    4.579404] ALSA device list:
[    4.579697]   No soundcards found.
[    4.589301] platform regulatory.0: Direct firmware load for regulatory.db
failed with error -2
[    4.590014] cfg80211: failed to load regulatory.db
[    4.609105] List of all partitions:
[    4.610016] 0100          65536 ram0
[    4.610034]  (driver?)
[    4.610678] 0101          65536 ram1
[    4.610683]  (driver?)
[    4.611911] 0102          65536 ram2
[    4.611915]  (driver?)
[    4.612126] 0103          65536 ram3
[    4.612129]  (driver?)
[    4.612401] 0104          65536 ram4
[    4.612405]  (driver?)
[    4.612617] 0105          65536 ram5
[    4.612620]  (driver?)
[    4.613813] 0106          65536 ram6
[    4.613817]  (driver?)
[    4.614004] 0107          65536 ram7
[    4.614055]  (driver?)
[    4.614430] 0108          65536 ram8
[    4.614435]  (driver?)
[    4.614645] 0109          65536 ram9
[    4.614649]  (driver?)
[    4.614953] 010a          65536 ram10
[    4.614958]  (driver?)
[    4.615299] 010b          65536 ram11
[    4.615305]  (driver?)
[    4.615627] 010c          65536 ram12
[    4.615631]  (driver?)
[    4.615957] 010d          65536 ram13
[    4.615961]  (driver?)
[    4.616295] 010e          65536 ram14
[    4.616298]  (driver?)
[    4.616614] 010f          65536 ram15
[    4.616618]  (driver?)
[    4.617030] 1f00          30720 mtdblock0
[    4.617054]  (driver?)
[    4.617373] 1f01            256 mtdblock1
[    4.617379]  (driver?)
[    4.617653] 1f02          36864 mtdblock2
[    4.617666]  (driver?)
[    4.618745] b300         2573312 mmcblk0
[    4.618774]  driver: mmcblk
[    4.619190]   b301          999424 mmcblk0p1 e29caf3d-01
[    4.619204]
[    4.619532]   b302         1572864 mmcblk0p2 e29caf3d-02
[    4.619536]
[    4.619883] No filesystem could mount root, tried:
[    4.619909]  ext3
```

```
[    4.620114]  ext2
[    4.620170]  ext4
[    4.620222]  cramfs
[    4.620274]  vfat
[    4.620326]  msdos
[    4.620375]  iso9660
[    4.620427]  btrfs
[    4.620486]
[    4.620735] Kernel panic - not syncing: VFS: Unable to mount root fs on
unknown-block(1,0)
[    4.620979] CPU: 0 PID: 1 Comm: swapper/0 Tainted: G        W         5.
4.0-xilinx-v2020.2 #1
[    4.621158] Hardware name: ZynqMP ZCU102 Rev1.0 (DT)
[    4.621283] Call trace:
[    4.621374]  dump_backtrace+0x0/0x140
[    4.621462]  show_stack+0x14/0x20
[    4.621553]  dump_stack+0xac/0xd0
[    4.621713]  panic+0x140/0x30c
[    4.621884]  mount_block_root+0x254/0x284
[    4.622123]  mount_root+0x124/0x158
[    4.622338]  prepare_namespace+0x15c/0x1a4
[    4.622561]  kernel_init_freeable+0x234/0x258
[    4.622815]  kernel_init+0x10/0xfc
[    4.623047]  ret_from_fork+0x10/0x18
[    4.623774] Kernel Offset: disabled
[    4.624068] CPU features: 0x0002,24002004
[    4.624218] Memory Limit: none
[    4.624790] ---[ end Kernel panic - not syncing: VFS: Unable to mount
root fs on unknown-block(1,0) ]---
```

This is often indicator that the kernel's not finding the root file system. This might be caused by tools to generate boot scripts (`boot.scr`) automatically. Make sure the boot line of your boot scripts points to your rootfs partition (e.g. `/dev/mmcblk0p2`).

PetaLinux

How do I setup the ROS 2 BSP?

```
git clone https://gitlab.com/xilinxrobotics/zcu102/bsp
source /tools/Xilinx/PetaLinux/2020.2/bin/settings.sh
petalinux-create -t project -s bsp/xilinx-zcu102-2020.2-ros2-foxy.bsp
cd xilinx-zcu102-2020.2
mkdir -p build/conf
cp project-spec/configs/plnxtool.conf build/conf/
cp project-spec/configs/bblayers.conf build/conf/

# IMPORTANT: fix bug manually
# Edit build/conf/bblayers.conf and remove the last workspace line
#

petalinux-config --silentconfig  # generate additional files
echo 'ROS_DISTRO = "foxy"' >> build/conf/bblayers.conf
```

```
# NOTE: you might need to do what's described in:
#  How do I deal with the following error: Could not inherit file classes/
ros_distro_${ROS_DISTRO}.bbclass

petalinux-build  # takes about 30 mins
```

How do I deal with the following error: Could not inherit file classes/ros_distro_${ROS_DISTRO}.bbclass?

When building the Yocto project with ROS, you may face:

```
petalinux-build  # takes about 30 mins
INFO: Sourcing build tools
[INFO] Building project
[INFO] Sourcing build environment
[INFO] Generating workspace directory
INFO: bitbake petalinux-image-minimal
WARNING: Layer example should set LAYERSERIES_COMPAT_example in its conf/
layer.conf file to list the core layer names it is compatible with.
WARNING: Host distribution "ubuntu-20.04" has not been validated with this
version of the build system; you may possibly experience unexpected
failures. It is recommended that you use a tested distribution.
ERROR: ParseError at /home/xilinx/xilinx-zcu102-2020.2/project-spec/meta-
user/recipes-images/images/petalinux-image-ros2-basic.bb:6: Could not
inherit file classes/ros_distro_${ROS_DISTRO}.bbclass40

Summary: There were 2 WARNING messages shown.
Summary: There was 1 ERROR message shown, returning a non-zero exit code.
ERROR: Failed to build project
```

Add the following to the `BBLAYERS` variable in the `build/conf/bblayers.conf` file:

```
  ${SDKBASEMETAPATH}/../../project-spec/meta-ros/meta-ros2-foxy \
  ${SDKBASEMETAPATH}/../../project-spec/meta-ros/meta-ros2 \
  ${SDKBASEMETAPATH}/../../project-spec/meta-ros/meta-ros-common \
  ${SDKBASEMETAPATH}/../../project-spec/meta-ros/meta-ros-backports-dunfell \
  ${SDKBASEMETAPATH}/../../project-spec/meta-user \
```

How to update the pre-built directory in a PetaLinux/Yocto project

```
petalinux-package --prebuilt
```

How to create a project out of the BSP

E.g. from ZCU102 Robotics BSP:

```
petalinux-create -t project -s <location-to>/xilinx-zcu102-2020.2-ros2-
foxy.bsp
```

How to generate a BSP (out of a PetaLinux/Yocto project):

```
petalinux-package --bsp --o /tmp/xilinx-zcu102-2020.2-ros2-foxy.bsp -p $(pwd)
```

How to launch a quick emulation from the BSP pre-built packages:

```
petalinux-boot --qemu --prebuilt 3
```

How to launch a quick emulation from images/linux folder (just built artifacts):

```
petalinux-boot --qemu --kernel
```

How to launch a quick emulation from a built PetaLinux/Yocto project using a TFTP directory (fetching things from TFTP):

```
petalinux-boot --qemu --prebuilt 2 --qemu-args "-net nic -net nic -net nic -
net nic -net user,tftp=images/linux/tftpboot,hostfwd=tcp:
127.0.0.1:2222-10.0.2.15:22"
```

How to add the rootfs resulting from a new build to a pre-built `.img`?

Partially inspired by [26]:

```
# mount p2, which contains the rootfs
sudo mount -o loop,offset=1024458752 /home/xilinx/ros2_ws/acceleration/
firmware/xilinx/sd_card.img /tmp/sdcard_img_p2
# extract new rootfs on .cpio format
cd /tmp/sdcard_img_p2
sudo cpio -idv < /media/xilinx/hd/xilinx-zcu102-2020.2/images/linux/
rootfs.cpio
```

or if the file available is gzipped:

```
cat /home/xilinx/ros2_ws/acceleration/firmware/xilinx/rootfs.cpio.gz |
gunzip | sudo cpio -idv
```

How to find Linux Kernel Latencies

Inspired by [26]:

```
# run cyclictests for 10 seconds
# NOTE: you should run benchmarks for a much longer time
cyclictest --smp -p95 -m -D 10
```

How to generate the `BOOT.BIN` file?

```
petalinux-package --boot \
   --fsbl images/linux/zynqmp_fsbl.elf \
   --u-boot images/linux/u-boot.elf \
```

```
    --pmufw images/linux/pmufw.elf \
    --atf images/linux/bl31.elf
```

How do I compile a PREEMPT_RT patched environment?

```
petalinux-build -c linux-xlnx -x distclean  # clean up kernel build files
petalinux-config -c kernel  # get the kernel under components for
configuration, do not modify anything
cd components/yocto/workspace/sources/linux-xlnx/  # head to the kernel
source tree
wget http://cdn.kernel.org/pub/linux/kernel/projects/rt/5.4/older/
patch-5.4.10-rt4.patch.gz  # fetch patches
zcat patch-5.4.10-rt4.patch.gz | patch -p1  # apply patches

# manual fixes, unfortunately patches are a hack and not meant for this
version. Need to:
#   Change instances of spin_lock for raw_spin_lock in function
rescuer_thread(). See
#   https://gitlab.com/xilinxrobotics/docs/-/issues/11#note_523328000 for
the actual fixes.

cd ../../../../.. # then back to the root of the project and:
petalinux-config -c kernel  # configure PREEMPT_RT options, setting to =y
the following
# CONFIG_PREEMPT_RT
# CONFIG_HIGH_RES_TIMERS
# CONFIG_NO_HZ_FULL
# CONFIG_HZ_1000
# CPU_FREQ_DEFAULT_GOV_PERFORMANCE
petalinux-build
```

Done with 2020.2

How do I set up TSN in a Yocto/PetaLinux project?

1. `git clone https://gitlab.com/xilinxrobotics/meta-tsn` under project*
2. `build/conf/local.conf` as in https://gist.github.com/vmayoral/f93d0070b93d9a7f43756ae9ae9c0bc3
3. Edit `build/conf/bblayers.conf` and add

```
  ${SDKBASEMETAPATH}/../../project-spec/meta-tsn/meta-xilinx-tsn \
```

1. Also, change the machine name in `petalinux-config` to match the one matching the config of the TSN IP core. Use `zcu102-zynqmp` instead (Yocto Seetings --> `YOCTO_MACHINE_NAME`)
2. Pick up the right hardware description file so that the right `dtb` is generated. E.g.:

```
petalinux-config --get-hw-description project-spec/meta-tsn/tsn-packages/
design/zcu102-zynqmp/
```

6. `petalinux-build`

How do I set up Xen in PetaLinux?

To set up Xen hypervisor in PetaLinux, follow the Confluence instructions.

How do I set up XRT in PetaLinux?

To set up XRT in PetaLinux, follow these instructions.

How do I clean up a PetaLinux component?

There're different ways to do. See PetaLinux Command Line Reference UG1157 for more details:

```
petalinux-buid -c <component> -x <verb>
```

wherein `<component>` corresponds with one of the recipes, e.g. `u-boot`.

verb	Description
`clean`	Cleans build data for the target component.
`cleansstate`	This removes the sstate cache of the corresponding component.
`distclean`	This removes the sstate cache of the corresponding component.
`cleanall`	This removes the downloads, sstate cache and cleans the work directory of a component.
`mrproper`	Cleans the build area. This removes the `<plnx-proj-root>/build/` and `<plnx-proj-root>/images/directories`

How do I build an SDK out of a PetaLinux project?

The following command builds SDK and copies it at `<proj_root>/images/linux/sdk.sh`:

```
petalinux-build --sdk
```

The following is the equivalent bitbake command bitbake

```
petalinux-user-image -c do_populate_sdk
```

Xen and mixed-criticality

How do I switch in between consoles/serial inputs (Dom0, Dom1, Xen)?

Press Ctrl-aaa **twice**.

How do I connect to a VM from Dom0?

```
xl console domU0  # or whatever the domain name is
```

Why there's no prompt in Dom0 after boot?

The default FS built with PetaLinux may contain a wrong configuration. Make sure that the following is set appropriately in `/etc/inittab`

```
#PS0:12345:respawn:/bin/start_getty 115200 ttyPS0 vt102
X0:12345:respawn:/sbin/getty 115200 hvc0
```

How do I create a Dom0less setup?

By default, VMs created with the Imagebuilder tool (e.g. `uboot-script-gen`) through configuration scripts are dom0less at the time of writing. For example, in the following piece, Dom1 is Dom0less:

```
MEMORY_START="0x0"
MEMORY_END="0x80000000"

DEVICE_TREE="system.dtb"
XEN="xen"
DOM0_KERNEL="Image"
DOM0_RAMDISK="initrd.cpio"

NUM_DOMUS=1
DOMU_KERNEL[0]="Image"
DOMU_RAMDISK[0]="initrd.cpio"

UBOOT_SOURCE="boot.source"
UBOOT_SCRIPT="boot.scr"
```

How do I create DomU machines then?

Through the `xl` tooling. For example:

```
# create config file and launch VM
cd /etc/xen
cat << EOF > example-minimalistic.cfg
name = "guest0"
kernel = "/media/sd-mmcblk0p1/Image"
ramdisk = "/media/sd-mmcblk0p1/initrd.cpio"
memory = 256
```

```
EOF
xl create -c example-minimalistic.cfg
```

You can exit Xen pressing `Ctrl-]` and then check that the VM is running:

```
root@xilinx-zcu102-2020_2-ros2-foxy:/etc/xen# xl list
Name                                        ID   Mem VCPUs  State   Time(s)
Domain-0                                     0  1024     1     r-----
206.3
guest0                                       2   255     1     r-----
9.0
```

What are VMs with no name?

When VMs are dom0less, they will get presented as follows:

```
root@xilinx-zcu102-2020_2-ros2-foxy:/# xl list
Name                                        ID   Mem VCPUs  State   Time(s)
(null)                                       0  1024     1     r-----
722.6
(null)                                       1   512     1     r-----
722.6
```

One way to attempt to access these VMs is by switching between serials/consoles as described above.

How do I exit from a DomU without stopping it and get back to Dom0?

Press `Ctrl-]`. That'll exit from the DomU console.

DomU VMs are not initiating with an error like the following

It's been observe that PetaLinux creates file systems that miss some Xen-related folders. This leads to issues initiating DomUs:

```
xl create -c example-minimalistic.c
Parsing config from example-minimalistic.cfg
libxl: error: libxl_domain.c:1405:libxl__get_domid: failed to get own domid
(domid)
libxl: error: libxl_dm.c:3415:libxl__need_xenpv_qemu: unable to get domain id
libxl: error: libxl_domain.c:1405:libxl__get_domid: failed to get own domid
(domid)
libxl: error: libxl_internal.c:421:libxl__lock_domain_userdata: Domain
2:cannot open lockfile /var/lib/xen/userdata-l.2.f10ed589-4870-40d0-
a4cc-758e10e8c9e3.domain-userdata-lock, errno=2: No such file or directory
libxl: error: libxl_domain.c:1131:domain_destroy_callback: Domain 2:Unable
to destroy guest
libxl: error: libxl_create.c:1826:domcreate_destruction_cb: Domain 2:unable
to destroy domain following failed creation
libxl: error: libxl_xshelp.c:201:libxl__xs_read_mandatory: xenstore read
failed: `/libxl/2/type': No such file or directory
libxl: warning: libxl_dom.c:52:libxl__domain_type: unable to get domain type
```

```
for domid=2, assuming HVM
libxl: error: libxl_internal.c:421:libxl__lock_domain_userdata: Domain
2:cannot open lockfile /var/lib/xen/userdata-l.2.f10ed589-4870-40d0-
a4cc-758e10e8c9e3.domain-userdata-lock, errno=2: No such file or directory
libxl: error: libxl_domain.c:1131:domain_destroy_callback: Domain 2:Unable
to destroy guest
libxl: error: libxl_domain.c:1058:domain_destroy_cb: Domain 2:Destruction of
domain failed
```

The reason why this is happening is because Xen is missing some folders which block from it to behave normally. This is illustrated also by the following error present in the boot log:

```
libxl: error: libxl_internal.c:421:libxl__lock_domain_userdata: Domain
0:cannot open lockfile /var/lib/xen/userdata-l.
0.00000000-0000-0000-0000-000000000000.domain-userdata-lock, errno=2: No
such file or directory
```

The solution is to manually create the missing folder and reboot:

```
mkdir /var/lib/xen
reboot
```

How do I deal with kernel errors while building Xen with PetaLinux?

When facing the following error:

```
NOTE: Setscene tasks completed
ERROR: petalinux-image-minimal-1.0-r0 do_rootfs: Could not invoke dnf.
Command '/home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/zcu102_zynqmp-
xilinx-linux/petalinux-image-minimal/1.0-r0/recipe-sysroot-native/usr/bin/
dnf -v --rpmverbosity=info -y -c /home/xilinx/xilinx-zcu102-2020.2/build/tmp/
work/zcu102_zynqmp-xilinx-linux/petalinux-image-minimal/1.0-r0/rootfs/etc/
dnf/dnf.conf --setopt=reposdir=/home/xilinx/xilinx-zcu102-2020.2/build/tmp/
work/zcu102_zynqmp-xilinx-linux/petalinux-image-minimal/1.0-r0/rootfs/etc/
yum.repos.d --installroot=/home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/
zcu102_zynqmp-xilinx-linux/petalinux-image-minimal/1.0-r0/rootfs --
setopt=logdir=/home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/zcu102_zynqmp-
xilinx-linux/petalinux-image-minimal/1.0-r0/temp --repofrompath=oe-repo,/
home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/zcu102_zynqmp-xilinx-linux/
petalinux-image-minimal/1.0-r0/oe-rootfs-repo --nogpgcheck install base-
passwd bridge-utils byobu can-utils cyclonedds demo-nodes-cpp demo-nodes-cpp-
rosnative demo-nodes-py ethtool examples-rclcpp-minimal-action-client
examples-rclcpp-minimal-action-server examples-rclcpp-minimal-client
examples-rclcpp-minimal-composition examples-rclcpp-minimal-publisher
examples-rclcpp-minimal-service examples-rclcpp-minimal-subscriber examples-
rclcpp-minimal-timer examples-rclcpp-multithreaded-executor examples-rclpy-
executors examples-rclpy-minimal-action-client examples-rclpy-minimal-action-
server examples-rclpy-minimal-client examples-rclpy-minimal-publisher
examples-rclpy-minimal-service examples-rclpy-minimal-subscriber fpga-
manager-script gdb libc6-utils haveged hellopm kernel-modules localedef mtd-
utils opencl-clhpp opencl-clhpp-dev opencl-headers opencl-headers-dev
openssh openssh-scp openssh-sftp-server openssh-ssh openssh-sshd
packagegroup-core-boot packagegroup-core-ssh-openssh packagegroup-petalinux-
```

```
xen packagegroup-petalinux-xrt pciutils python3-argcomplete rmw-cyclonedds-
cpp ros-base rpm rt-tests run-postinsts shadow stress tcf-agent tmux udev-
extraconf watchdog-init xrt xrt-dev zocl' returned 1:
DNF version: 4.2.2
cachedir: /home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/zcu102_zynqmp-
xilinx-linux/petalinux-image-minimal/1.0-r0/rootfs/var/cache/dnf
Added oe-repo repo from /home/xilinx/xilinx-zcu102-2020.2/build/tmp/work/
zcu102_zynqmp-xilinx-linux/petalinux-image-minimal/1.0-r0/oe-rootfs-repo
repo: using cache for: oe-repo
not found other for:
not found modules for:
not found deltainfo for:
not found updateinfo for:
oe-repo: using metadata from Wed 26 May 2021 03:15:15 PM UTC.
Last metadata expiration check: 0:00:01 ago on Wed 26 May 2021 03:15:15 PM
UTC.
No module defaults found
--> Starting dependency resolution
--> Finished dependency resolution
Error:
 Problem: conflicting requests
  - nothing provides kernel-module-xen-blkback needed by packagegroup-
petalinux-xen-1.0-r0.noarch
  - nothing provides kernel-module-xen-gntalloc needed by packagegroup-
petalinux-xen-1.0-r0.noarch
  - nothing provides kernel-module-xen-gntdev needed by packagegroup-
petalinux-xen-1.0-r0.noarch
  - nothing provides kernel-module-xen-netback needed by packagegroup-
petalinux-xen-1.0-r0.noarch
  - nothing provides kernel-module-xen-wdt needed by packagegroup-petalinux-
xen-1.0-r0.noarch
(try to add '--skip-broken' to skip uninstallable packages or '--nobest' to
use not only best candidate packages)

ERROR: Logfile of failure stored in: /home/xilinx/xilinx-zcu102-2020.2/build/
tmp/work/zcu102_zynqmp-xilinx-linux/petalinux-image-minimal/1.0-r0/temp/
log.do_rootfs.1714208
ERROR: Task (/home/xilinx/xilinx-zcu102-2020.2/components/yocto/layers/meta-
petalinux/recipes-core/images/petalinux-image-minimal.bb:do_rootfs) failed
with exit code '1'
NOTE: Tasks Summary: Attempted 9800 tasks of which 9498 didn't need to be
rerun and 1 failed.
```

DomUs stored in SD partitions create a kernel panic when Launching

If you're running Vitis `2020.2` or earlier and observe something like when creating DomUs:

```
...
[    3.257786] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[    3.258491] ALSA device list:
[    3.258603]   No soundcards found.
[    3.264081] platform regulatory.0: Direct firmware load for regulatory.db
failed with error -2
[    3.264785] cfg80211: failed to load regulatory.db
[    3.318562] random: fast init done
[    3.389439] List of all partitions:
```

```
[    3.389763] 0100           65536 ram0
[    3.389800]  (driver?)
[    3.389941] 0101           65536 ram1
[    3.389949]  (driver?)
[    3.390181] 0102           65536 ram2
[    3.390185]  (driver?)
[    3.390318] 0103           65536 ram3
[    3.390324]  (driver?)
[    3.390420] 0104           65536 ram4
[    3.390424]  (driver?)
[    3.390510] 0105           65536 ram5
[    3.390514]  (driver?)
[    3.390926] 0106           65536 ram6
[    3.390931]  (driver?)
[    3.391029] 0107           65536 ram7
[    3.391032]  (driver?)
[    3.391112] 0108           65536 ram8
[    3.391115]  (driver?)
[    3.391301] 0109           65536 ram9
[    3.391305]  (driver?)
[    3.391395] 010a           65536 ram10
[    3.391398]  (driver?)
[    3.391484] 010b           65536 ram11
[    3.391487]  (driver?)
[    3.391573] 010c           65536 ram12
[    3.391578]  (driver?)
[    3.391680] 010d           65536 ram13
[    3.391696]  (driver?)
[    3.391924] 010e           65536 ram14
[    3.391929]  (driver?)
[    3.392739] 010f           65536 ram15
[    3.392744]  (driver?)
[    3.392916] ca00         1656832 xvda
[    3.392945]  driver: vbd
[    3.393071] No filesystem could mount root, tried:
[    3.393173]  ext3
[    3.393266]  ext2
[    3.393337]  ext4
[    3.393378]  cramfs
[    3.393412]  vfat
[    3.393447]  msdos
[    3.393489]  iso9660
[    3.393628]  btrfs
[    3.393681]
[    3.394073] Kernel panic - not syncing: VFS: Unable to mount root fs on
unknown-block(202,0)
[    3.395086] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 5.4.0-xilinx-
v2020.2 #1
[    3.395268] Hardware name: XENVM-4.13 (DT)
[    3.395587] Call trace:
[    3.395667]  dump_backtrace+0x0/0x140
[    3.395859]  show_stack+0x14/0x20
[    3.396728]  dump_stack+0xac/0xd0
[    3.396825]  panic+0x140/0x30c
[    3.396912]  mount_block_root+0x254/0x284
[    3.396985]  mount_root+0x124/0x158
[    3.397061]  prepare_namespace+0x15c/0x1a4
```

```
[    3.397222]  kernel_init_freeable+0x234/0x258
[    3.397396]  kernel_init+0x10/0xfc
[    3.397532]  ret_from_fork+0x10/0x18
[    3.398088] Kernel Offset: disabled
[    3.398414] CPU features: 0x0002,24002004
[    3.398541] Memory Limit: none
[    3.399028] ---[ end Kernel panic - not syncing: VFS: Unable to mount
root fs on unknown-block(202,0) ]---
```

Chances are that your Xen tree doesn't include https://github.com/Xilinx/xen/commit/ab9054b8e902b0fa80af02ad1e0a8c4d383e90e1 which fixes a bug with PV drivers that causes that error. Make sure you rebuild Xen including that commit.

KRS

How do I ssh into an emulation through KRS tools (e.g. `colcon krs emulation`)?

`colcon krs` extensions enable emulations which forward ports between the guest/host machines. Particularly, the following flags are passed to QEMU: `hostfwd=tcp:127.0.0.1:2222-10.0.2.15:22` (among others). To ssh into the emulation:

```
ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -p 2222
root@localhost
```

How do I scp files from/into an emulation launched through KRS tools (e.g. `colcon krs emulation`)?

`colcon krs` extensions enable emulations which forward ports between the guest/host machines. Particularly, the following flags are passed to QEMU: `hostfwd=tcp:127.0.0.1:2222-10.0.2.15:22` (among others). E.g., to copy files from the emulation into the host machine:

```
scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -P 2222
root@localhost:/ros2_ws/lib/accelerated_vadd_publisher_once/*.csv .
```

How do I rebuild ROS 2 packages without the acceleration kernels?

Just add `--cmake-args -DNOKERNELS=true` to the build command. E.g.:

```
colcon build --merge-install --mixin zcu102 --install-base install-zcu102 --
build-base build-zcu102 --cmake-args -DNOKERNELS=true
```

> **Note**
>
> This flag persists so if you wish to change it afterwards and re-build kernels, you'll need to set it to false `--cmake-args -DNOKERNELS=false`.

How do I port KRS to a new board?

1. Build firmware artifacts in `acceleration_firmware_*`
2. If your board is based on a Xilinx adaptive SoC (e.g. Versal, Zynq Ultrascale+, Zynq 7000), we recommend you to clone acceleration_firmware_xilinx and create a new branch for your board replacing all artifacts as defined in the ARTIFACTS.md file. Afterwards, submit a pull request.
3. A minimal rootfs should be created including:
    - ROS 2 Foxy (at least bare bones)
    - Xen hypervisor capabilities
    - Xilinx Runtime (XRT)
    - `vanilla` and `PREEMPT_RT` patched kernels including:
    - LTTng kernel support
    - U-boot with PXE support
4. Create a sysroot (or an SDK that produces such sysroot) for the target rootfs. NOTE that this sysroot might differ across different file systems (e.g. Ubuntu-based vs Yocto-based).
5. Add custom mixins in `acceleration_firmware_*` for the new board. See `xilinx.mixin.template` for an example.
6.

How do I build several accelerators when invoking `colcon build`?

The current implementation of KRS leverages Vitis Unified platform and compiler which expects to build one accelerator at a time. If two are being built simultaneously, undefined behaviour is triggered and typically, only one of the accelerators will finalize while the other one won't (and thereby no `xclbin` file will be generated as a result).

The solution to this is to serialize the generation of acceleration kernels using colcon arguments as follows:

```
colcon build --executor sequential --merge-install --mixin kv260 --install-
base install-kv260 --build-base build-kv260
```

For more, review this ticket.

How do I handle device-mapper issues?

KRS relies on imagebuilder which employs `/dev/mapper`. This Linux interface is a bit sensitive and if not disabled properly in past runs will lead into:

```
colcon krs kernel --install-dir install-kv260
SECURITY WARNING: This class invokes explicitly a shell via the shell=True
argument of the Python subprocess library, and uses admin privileges to
manage raw disk images. It is the user's responsibility to ensure that all
whitespace and metacharacters passed are quoted appropriately to avoid shell
injection vulnerabilities.
- Creating a new base image using /home/xilinx/ros2_ws/acceleration/firmware/
xilinx/rootfs.cpio.gz ...
- Detected previous sd_card.img raw image, moving to sd_card.img.old.
device-mapper: create ioctl on diskimage  failed: Device or resource busy
Command failed.
...
```

To fix this:

```
sudo kpartx -d /dev/mapper/diskimage
sudo  dmsetup remove diskimage
sudo losetup -d $(($(sudo losetup -f | sed -s 's*/dev/loop**') - 1))
```

A simpler approach was implemented in KRS with `colcon krs umount --fix` which does exactly the same.

How do I handle "board_part definition was not found" type of issues?

Consider the following error while building an acceleration kernel:

> **Note**
>
> This issue was first documented at https://forums.xilinx.com/t5/Kria-SOMs/Vitis-IDE-ERROR-v-60-602-Source-file-does-not-exist/m-p/1264669/highlight/true

```
INFO: [VPL 60-839] Read in kernel information from file '/home/xilinx/
ros2_ws/build-kv260/accelerated_vadd_publisher/_x/link/int/kernel_info.dat'.
INFO: [VPL 60-423]   Target device: kv260_ispMipiRx_vcu_DP
INFO: [VPL 60-1032] Extracting hardware platform to /home/xilinx/ros2_ws/
build-kv260/accelerated_vadd_publisher/_x/link/vivado/vpl/.local/hw_platform
WARNING: /tools/Xilinx/Vitis/2020.2/tps/lnx64/jre9.0.4 does not exist.
[19:52:08] Run vpl: Step create_project: Started
Creating Vivado project.
[19:52:10] Run vpl: Step create_project: Failed
[19:52:11] Run vpl: FINISHED. Run Status: create_project ERROR
ERROR: [VPL 60-773] In '/home/xilinx/ros2_ws/build-kv260/
accelerated_vadd_publisher/_x/link/vivado/vpl/runme.log', caught Tcl error:
ERROR: [Board 49-71] The board_part definition was not found for
xilinx.com:kv260:part0:1.1. The project's board_part property was not set,
```

```
but the project's part property was set to xck26-sfvc784-2LV-c. Valid
board_part values can be retrieved with the 'get_board_parts' Tcl command.
Check if board.repoPaths parameter is set and the board_part is installed
from the tcl app store.
ERROR: [VPL 60-704] Integration error, Failed to rebuild a project required
for hardware synthesis. The project is 'prj'. The rebuild script is '.local/
hw_platform/prj/rebuild.tcl'. The rebuild script was delivered as part of
the hardware platform. Consult with the hardware platform provider to
investigate the rebuild script contents. An error stack with function names
and arguments may be available in the 'vivado.log'.
ERROR: [VPL 60-1328] Vpl run 'vpl' failed
WARNING: [VPL 60-1142] Unable to read data from '/home/xilinx/ros2_ws/build-
kv260/accelerated_vadd_publisher/_x/link/vivado/vpl/output/
generated_reports.log', generated reports will not be copied.
ERROR: [VPL 60-806] Failed to finish platform linker
INFO: [v++ 60-1442] [19:52:11] Run run_link: Step vpl: Failed
Time (s): cpu = 00:00:05 ; elapsed = 00:00:15 . Memory (MB): peak = 1574.
906 ; gain = 0.000 ; free physical = 1594 ; free virtual = 20748
ERROR: [v++ 60-661] v++ link run 'run_link' failed
ERROR: [v++ 60-626] Kernel link failed to complete
ERROR: [v++ 60-703] Failed to finish linking
INFO: [v++ 60-1653] Closing dispatch client.
```

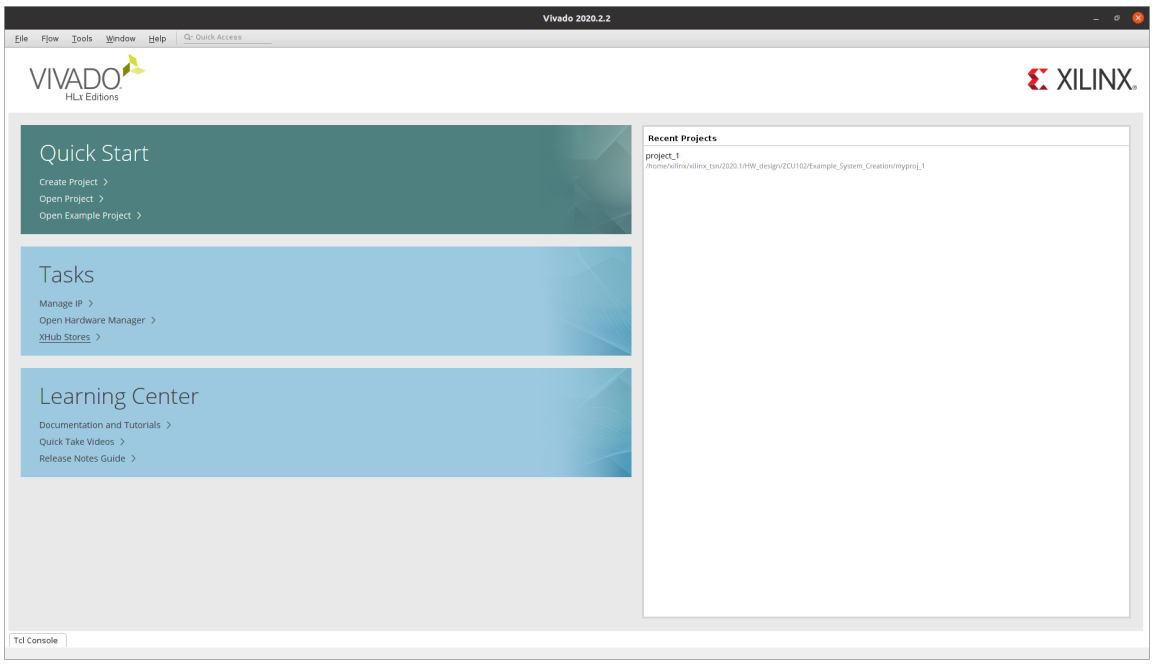
In this case, Vivado's not able to find the `xilinx.com:kv260:part0:1.1` board part. We can verify this manually through Vivado CLI:

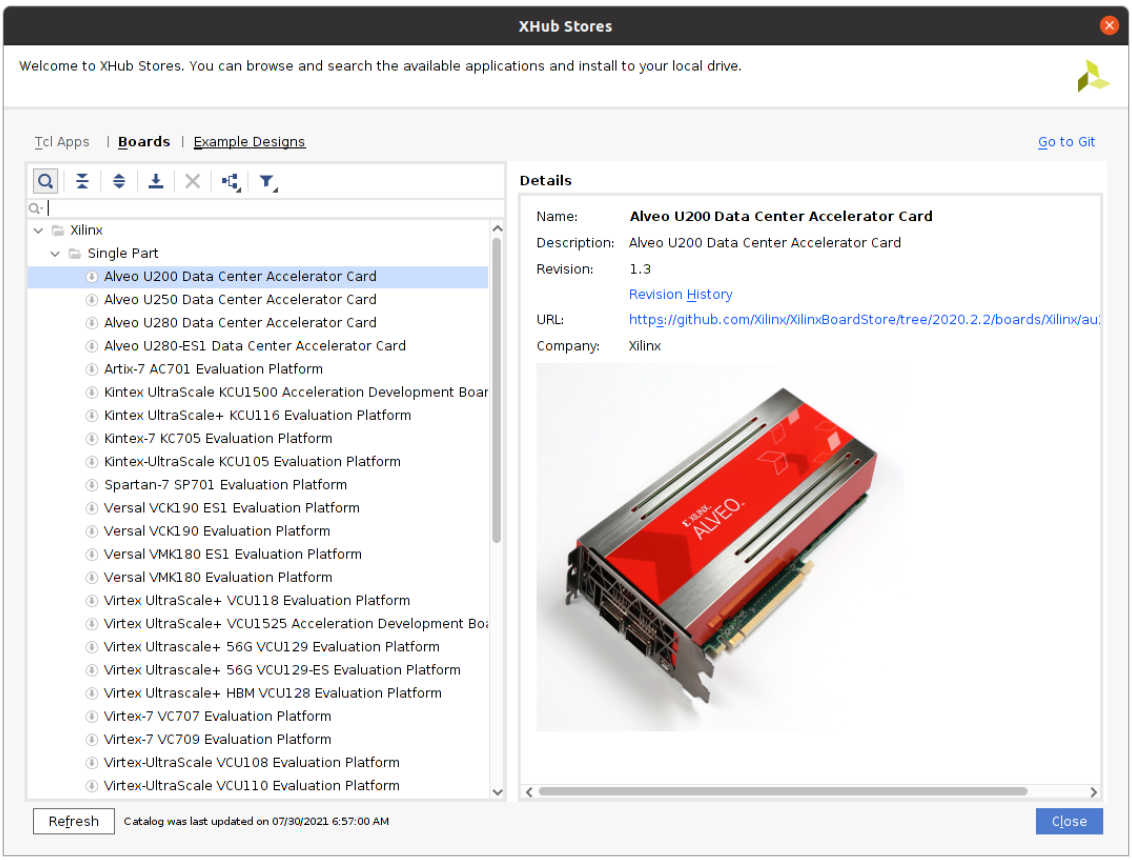
```
vivado -mode tcl
Vivado% get_board_parts xilinx.com:kv*
WARNING: [Vivado 12-4842] No board parts matched 'get_board_parts
xilinx.com:kv*'.
```

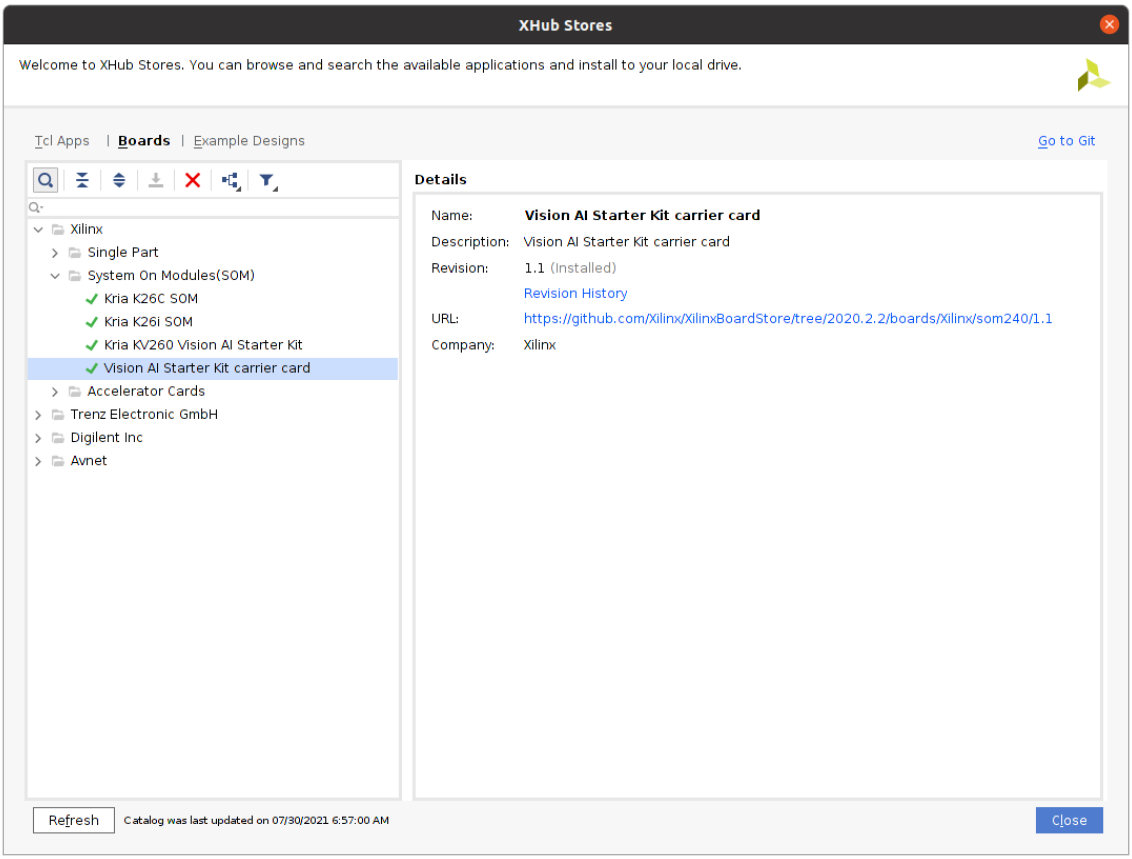
Figure out if this can be automated via scripting

This could likely be automated through Tcl or similar. Find out if time allows for it in the future.

To fetch these, one needs to go ahead and install the corresponding files. This can be done through Vivado's XHub Stores:


Vivado 2020.2.2
File Flow Tools Window Help
VIVADO
HLx Editions
XILINX
Quick Start
Create Project
Open Project
Open Example Project
Tasks
Manage IP
Open Hardware Manager
XHub Stores
Learning Center
Documentation and Tutorials
Quick Take Videos
Release Notes Guide
Recent Projects
project_1
Tcl Console


XHub Stores
Welcome to XHub Stores. You can browse and search the available applications and install to your local drive.
Tcl Apps | Boards | Example Designs
Go to Git
Xilinx
Single Part
Alveo U200 Data Center Accelerator Card
Alveo U250 Data Center Accelerator Card
Alveo U280 Data Center Accelerator Card
Alveo U280-ES1 Data Center Accelerator Card
Artix-7 AC701 Evaluation Platform
Kintex UltraScale KCU1500 Acceleration Development Boar
Kintex UltraScale+ KCU116 Evaluation Platform
Kintex-7 KC705 Evaluation Platform
Kintex-UltraScale KCU105 Evaluation Platform
Spartan-7 SP701 Evaluation Platform
Versal VCK190 ES1 Evaluation Platform
Versal VCK190 Evaluation Platform
Versal VMK180 ES1 Evaluation Platform
Versal VMK180 Evaluation Platform
Virtex UltraScale+ VCU118 Evaluation Platform
Virtex UltraScale+ VCU1525 Acceleration Development Boa
Virtex Ultrascale+ 56G VCU129 Evaluation Platform
Virtex Ultrascale+ 56G VCU129-ES Evaluation Platform
Virtex Ultrascale+ HBM VCU128 Evaluation Platform
Virtex-7 VC707 Evaluation Platform
Virtex-7 VC709 Evaluation Platform
Virtex-UltraScale VCU108 Evaluation Platform
Virtex-UltraScale VCU110 Evaluation Platform
Details
Name: Alveo U200 Data Center Accelerator Card
Description: Alveo U200 Data Center Accelerator Card
Revision: 1.3
Revision History
URL: https://github.com/Xilinx/XilinxBoardStore/tree/2020.2.2/boards/Xilinx/au
Company: Xilinx
Refresh
Catalog was last updated on 07/30/2021 6:57:00 AM
Close



And then add a script that gets them automatically:

```
# create a file that loads 2020.2.2 board files at vivado init
cat << 'EOF' > ~/.Xilinx/Vivado/Vivado_init.tcl
set_param board.repoPaths {\
    /home/xilinx/.Xilinx/Vivado/2020.2.2/xhub/board_store/xilinx_board_store
\
}
EOF

# then search again
vivado -mode tcl
...
Vivado% get_board_parts xilinx.com:kv*
xilinx.com:kv260:part0:1.1
```

ROS

How do I change the install path whereto search for libraries?

Using the `--cmake-args -DCMAKE_FIND_ROOT_PATH=$(pwd)/install-kv260` argument. E.g.

```
colcon build --build-base=build-kv260 --install-base=install-kv260 --merge-
install --mixin kv260 --cmake-args -DNOKERNELS=true -DCMAKE_FIND_ROOT_PATH=$
(pwd)/install-kv260
```

How do I add additional verbosity to the build process with `colcon`?

```
colcon --log-level 10 build --merge-install
```

How do I add debugging symbols to ROS 2 builds?

Add `--cmake-args -DCMAKE_BUILD_TYPE=Debug` :

```
colcon build --merge-install --cmake-args -DCMAKE_BUILD_TYPE=Debug
```

or alternative, if all mixins are installed, you can:

```
colcon build --merge-install --mixin debug
```

Vitis

What's the Boot Image Format (`.bif` files) and where do I learn more about it?

From [33]

> The Xilinx® boot image layout has multiple files, file types, and supporting headers to parse those files by boot loaders. Bootgen defines multiple attributes for generating the boot images and interprets and generates the boot images, based on what is passed in the files. Because there are multiple commands and attributes available, Bootgen defines a boot image format (BIF) to contain those inputs. A BIF comprises of the following:
>
> ```
> - Configuration attributes to create secure/non-secure boot images
> - Bootloader
>     - First stage boot loader (FSBL) for Zynq® devices and Zynq®
> UltraScale+™ MPSoCs
>     - Platform loader and manager (PLM) for Versal™ ACAP
> - One or more Partition Images
> ```

To learn more, refer to: - UG1283 - Bootgen tool documentation

How do I generate a boot.bin from a `.bif` file and its related artifacts?

Let's assume:

```
xilinx@xilinx:/tmp/latebinding$ ls
arm-trusted-firmware.elf  fsbl-zcu102-zynqmp.elf          tsn.bit
bootgen.bif               pmu-firmware-zcu102-zynqmp.elf  u-boot-zcu102-
zynqmp.elf
```

where `bootgen.bif` contains:

```
cat bootgen.bif
the_ROM_image:
{
     [bootloader,  destination_cpu=a53-0] fsbl-zcu102-zynqmp.elf
     [destination_device=pl] tsn.bit
     [destination_cpu=pmu] pmu-firmware-zcu102-zynqmp.elf
     [destination_cpu=a53-0, exception_level=el-3, trustzone] arm-trusted-
firmware.elf
     [destination_cpu=a53-0, exception_level=el-2] u-boot-zcu102-zynqmp.elf
}
```

Building `boot.bin` can be accomplished with:

```
bootgen -arch zynqmp -image bootgen.bif -o boot.bin


****** Xilinx Bootgen v2020.2
  **** Build date : Nov 18 2020-09:50:31
    ** Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.


[INFO]   : Bootimage generated successfully
```

How do I introspect a BOOT.BIN file?

The `bootgen_utility` is a tool used to dump the contents of a Boot Image generated by Bootgen, into a human-readable log file. This is useful in debugging and understanding the contents of the different header tables of a boot image. More at the documentation.

```
bootgen_utility -arch zynqmp -bin boot.bin -out boot.bin.info
```

How do I execute a Tcl script from CLI?

1. Source Vitis platform environment
2. `vivado -mode batch -source <script.tcl>`

Vivado

How do I map an SoC featured in datasheets to the part for HLS?

> **Note**
>
> The part listed in datasheets is sometimes different from the one that can be used in `Tcl` scripts. E.g.:
>
> - ZCU102:
> - datasheet: XCZU9EG-2FFVB1156
> - part: xczu9eg-ffvb1156-2-e

To figure things out, use vivado "get_parts" command and match the closes. E.g.:

```
get_parts xczu9eg*

xczu9eg-ffvb1156-1-e xczu9eg-ffvb1156-1-i xczu9eg-ffvb1156-1L-i xczu9eg-
ffvb1156-1LV-i xczu9eg-ffvb1156-2-e xczu9eg-ffvb1156-2-i xczu9eg-ffvb1156-2L-
e xczu9eg-ffvb1156-2LV-e xczu9eg-ffvb1156-3-e xczu9eg-ffvc900-1-e xczu9eg-
ffvc900-1-i xczu9eg-ffvc900-1L-i xczu9eg-ffvc900-1LV-i xczu9eg-ffvc900-2-e
xczu9eg-ffvc900-2-i xczu9eg-ffvc900-2L-e xczu9eg-ffvc900-2LV-e xczu9eg-
ffvc900-3-e
```


1. Rifenbark, S. (2015). Yocto Project Mega-Manual. Retrieved from: https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html.
2. Xilinx (2016). *Zynq UltraScale+ MPSoC product brief*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/product-briefs/zynq-ultrascale-plus-product-brief.pdf.
3. Xilinx (2020). *Zynq UltraScale+ Device Technical Reference Manual*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.
4. Xilinx (2020). *UltraScale Architecture and Product Data Sheet: Overview*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.
5. Xilinx (2019). *Zynq UltraScale+ MPSoC Data Sheet:Overview*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.
6. Xilinx (2020). *Zynq UltraScale+ MPSoC Data Sheet:DC and AC Switching Characteristics*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf.
7. Xilinx (2016). *Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp470-ultrascale-plus-power-flexibility.pdf.

8. Xilinx (2016). *Managing Power and Performance with the Zynq UltraScale+ MPSoC*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp482-zu-pwr-perf.pdf.
9. Xilinx (2021). *Zynq UltraScale+ MPSoC Software Developer Guide*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1137-zynq-ultrascale-mpsoc-swdev.pdf.
10. Xilinx (2020). *Xilinx Quick Emulator User Guide*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1169-xilinx-qemu.pdf.
11. Xilinx (2020). *UltraScale Architecture Memory Resources*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.
12. Xilinx (2020). *UltraScale Architecture System Monitor*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug580-ultrascale-sysmon.pdf.
13. Xilinx (2020). *Zynq UltraScale+ Device Packaging and Pinouts*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug1075-zynq-ultrascale-pkg-pinout.pdf.
14. Xilinx (2016). *Software Migration to 64-bit ARM Heterogeneous Platforms*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp473-migration-to-64-bit-arm.pdf.
15. Xilinx (2016). *Enabling Virtualization with Xen Hypervisor on Zynq UltraScale+ MPSoCs*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp474-xen-hypervisor.pdf.
16. Xilinx (2016). *Toward 5G Xilinx Solutions and Enablers for Next-Generation Wireless Systems*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp476-toward-5g.pdf.
17. Khona, C. (2017). *Key Attributes of an Intelligent IIoT Edge Platform*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp493-iiot-edge-platforms.pdf.
18. Xilinx (2020). *Isolate Security-Critical Applications on Zynq UltraScale+ Devices*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp516-security-apps.pdf.
19. Xilinx (2020). *Key Revocation Lab*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/application_notes/xapp1344-key-revocation-lab.pdf.
20. Xilinx (2020). *UltraFast Design Methodology Guide for the Vivado Design Suite*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug949-vivado-design-methodology.pdf.
21. Xilinx (2018). *UltraFast Embedded Design Methodology Guide*. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/ug1046-ultrafast-design-methodology-guide.pdf.
22. Xilinx (2017). *Zynq UltraScale+ MPSoC Embedded Design Methodology Guide*. Xilinx. Retrieved from https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/ug1228-ultrafast-embedded-design-methodology-guide.pdf.
23. Xilinx (2018). *Zynq UltraScale+ MPSoC: Embedded Design Tutorial*. Xilinx. Retrieved from https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2018_2/ug1209-embedded-design-tutorial.pdf.

24. Xilinx (2020). *PetaLinux Tools Documentation. Reference Guide*. Xilinx. UG1144 (v2020.2) November 24, 2020. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1144-petalinux-tools-reference-guide.pdf.
25. Apertis. *Sysroots and Devroots*. Retrieved from https://www.apertis.org/architecture/sysroots-and-devroots/.
26. Arief Wicaksana. *Booting Xen Hypervisor with Petalinux on Zynq Ultrascale+* .Retrieved from https://witjaksana.github.io/2018/02/12/booting-xen-on-zcu102/.
27. Retrieved from http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt.
28. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. Retrieved from https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/xtp426-zcu102-quickstart.pdf
29. ZCU102 Evaluation Board. Retrieved from https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf
30. Xilinx ZCU102. Erika Enterprise. Retrieved from http://www.erika-enterprise.com/wiki/index.php?title=Xilinx_ZCU102
31. Zynq UltraScale + MPSoC Ubuntu part 2 - Building and Running the Ubuntu Desktop From Sources. Xilinx Confluence. Retrieved from https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841937/Zynq+UltraScale+MPSoC+Ubuntu+part+2+-+Building+and+Running+the+Ubuntu+Desktop+From+Sources
32. ramfs, rootfs and initramfs. Rob Landley. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt.
33. Creating Boot Images. Xilinx. Retrieved from https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/creatingbootimages.html#rsz1534706729087.

Definitions

- **Block RAM (BRAM)**: hardened RAM resource. More efficient memories than using LUTs for more than a few elements.
- **Compute Unit (CU)**: An OpenCL device has one or more compute units. A work-group executes on a single compute unit. A compute unit is composed of one or more processing elements and local memory. A compute unit may also include dedicated texture filter units that can be accessed by its processing elements[3].
- **Digital Signal Processor (DSP)**: performs multiplication and other arithmetic in the FPGA
- **Field-Programmable Gate Array (FPGA)**: semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing.
- **Flip Flops (FF)**: control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **Hardware Description Language (HDL)**: a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits[2].
- **High Level Synthesis (HLS)**: compiler for C, C++, SystemC into FPGA IP cores
- **Kernel**: An OpenCL kernel is a function declared in a program and executed on an OpenCL device. A kernel is identified by the kernel or kernel qualifier applied to any function defined in a program[3].
- **Look Up Table (LUT)**: generic functions on small bitwidth inputs (the logic itself). Combine many to build the algorithm.
- **Register Transfer Level (RTL)**: a design abstraction (typically graphical) which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. The very low level description of the function and connection of logic gates: *Register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is typical practice in modern digital design*.
- **Robot Operating System (ROS)**: a framework for robot application development. Includes tools, libraries, conventions and a community. The de-facto standard in robotics.
- **VHSIC Hardware Description Language (VHDL)**: a hardware description language (HDL) that can model the behavior and structure of digital systems.
- **Verilog**: standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems.

1. Hardware description language. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Hardware_description_language
2. Register-transfer level. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Register-transfer_level
3. The OpenCL™ Specification Version V2.2-11. Retrieved from https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html.