D3.2 OPENCL PROTOTYPE TO SUPPORT DISTRIBUTED EXECUTION OF KERNELS AND DATA TRANSFERS IN CPSS (DEMONSTRATOR REPORT)

Authors	Pekka Jääskeläinen (TAU), Michal Babej (TAU), Topi Leppänen (TAU), I2CAT, 8Bells, UoP, Robotec

Work Package	WP3 Model based CP(H)S Layer Design and Development supporting Distributed Assisted, Augmented and Autonomous Intelligence

Abstract

This deliverable is a *demonstrator report* that describes the output of *Task 3.2: "Design and Develop CPS Layer CPSoSaware Deployment/Commissioning and Execution Mechanism."* In addition to describing the demonstrator that showcases the OpenCL-based deployment/commissioning/execution stack using a hardware-in-the-loop example, it gives technical details on the other relevant aspects of the mechanism such as the OpenCL-based distributed execution environment itself, accelerator utilization layer using the built-in kernel feature of the specifications, distributed use of FPGA, deployment of software stack (containerization), as well as reconfiguration of the FPGA fabric and the network.

Funded by the Horizon 2020 Framework Programme
of the European Union

Deliverable Information

Work Package	WP3 Model based CP(H)S Layer Design and Development supporting Distributed Assisted, Augmented and Autonomous Intelligence
Task	T3.2 [M6-M28] Design and Develop CPS Layer CPSoSaware Deployment/Commissioning and Execution Mechanism
Deliverable title	D3.2 OPENCL PROTOTYPE TO SUPPORT DISTRIBUTED EXECUTION OF KERNELS AND DATA TRANSFERS IN CPSs
Dissemination Level	PU
Status	Final
Version Number	1.1
Due date	30/04/2022

Project Information

Project start and duration	01/01/2020 – 31/12/2022, 36 months
Project Coordinator	Industrial Systems Institute, ATHENA Research and Innovation Center 26504, Rio-Patras, Greece
Partners	1. ATHINA-EREVNITIKO KENTRO KAINOTOMIAS STIS TECHNOLOGIES TIS PLIROFORIAS, TON EPIKOINONION KAI TIS GNOSIS (ISI) the Coordinator 2. FUNDACIO PRIVADA I2CAT, INTERNET I INNOVACIO DIGITAL A CATALUNYA (I2CAT) 3. IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD (IBM ISRAEL) 4. ATOS SPAIN SA (ATOS) 5. PANASONIC AUTOMOTIVE SYSTEMS EUROPE GMBH (PASEU) 6. EIGHT BELLS LTD (8BELLS) 7. UNIVERSITA DELLA SVIZZERA ITALIANA (USI), 8. TAMPEREEN KORKEAKOULUSAATIO SR (TAU) 9. UNIVERSITY OF PELOPONNESE (UoP) 10. CATALINK LIMITED (CATALINK) 11. ROBOTEC.AI SPOLKA Z OGRANICZONA ODPOWIEDZIALNOSCIA (RTC) 12. CENTRO RICERCHE FIAT SCPA (CRF) 13. PANEPISTIMIO PATRON (UPAT)
Website	www.CPSoSaware.eu

Control Sheet

VERSION	DATE	SUMMARY OF CHANGES	AUTHOR
1.1	26/April/2022	Final version for EU portal.	Pekka Jääskeläinen (TAU)

	NAME
Prepared by	TAU, 8BELLS, I2CAT, UoP
Reviewed by	
Authorised by	

DATE	RECIPIENT
01/04/2022	Project Consortium
30/04/2022	European Commission

Table of contents

List of tables

List of figures

Executive summary

This deliverable is a *demonstrator report* that describes the output of *Task 3.2: "Design and Develop CPS Layer CPSoSaware Deployment/Commissioning and Execution Mechanism."* In addition to describing the demonstrator that showcases the OpenCL-based deployment/commissioning/execution stack using a hardware-in-the-loop example, it gives technical details on other key relevant aspects of the mechanism such as the OpenCL-based distributed execution environment itself, accelerator utilization layer using the built-in kernel feature of the specifications, distributed use of FPGA, deployment of software stack (containerization), as well as reconfiguration of the FPGA fabric and the network.

1 Introduction

This report contains the output of Task 3.2: *"Design and Develop CPS Layer CPSoSaware Deployment/Commissioning and Execution Mechanism."*. The deliverable describes the environment developed for deploying OpenCL SW kernels and HW FPGA bitstreams and the associated data transfers in the CPSoSaware platform in an adaptive and dynamic way.

This deliverable is a *demonstrator report* that describes a demonstator made out of the technologies for Task 3.2. The demonstrator is recorded as a video available here.

In addition to describing the demonstrator that showcases the OpenCL-based deployment/commissioning/execution stack using a hardware-in-the-loop example, it gives technical details on other the key relevant aspects of the mechanism such as the OpenCL-based distributed execution environment itself, accelerator utilization layer using the built-in kernel feature of the specifications, distributed use of FPGA, deployment of software stack (containerization), as well as reconfiguration of the FPGA fabric and the network.

1.1 Document structure

This document is structured into the following sections:

- Section 2 describes the distributed OpenCL execution and deployment environment.
- Section 3 is an overview of the hardware-in-the-loop demonstrator associated with this deliverable.
- Section 4 is about the way to utilize fixed function hardware accelerators from the execution environment.
- Section 5 focuses on the FPGA utilization aspects, especially how it was connected to the HitL demonstrator.
- Section 6 reports the deployment approach of the OpenCL based stack by utilizing LXC containers.
- Section 7 details the FPGA reconfiguration support.
- Section 8 highlights the networking reconfiguration support.
- Section 9 concludes the report.

1.2 Acronyms and descriptions

Below are listed the most relevant acronyms used in the document and recurring definitions:

Acronym / Term	Description
AlmaIF	Alma Interface
API	Application Programming Interface
ASIP	Application-Specific Instruction-set Processor
AXI	Advanced eXtensible Interface
CNN	Convolutional Neural Networks
CPS	Cyber-Physical System
CPSoS	Cyber-Physical System of Systems
CPU	Central Processing Unit

CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DEST	Deformable Shape Tracking
DNN	Deep Neural Network
DSM	Driver Status Monitoring
DSP	Digital Signal Processor
ERT	Ensemble of Regression Trees
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
HitL	Hardware-in-the-Loop (simulation or development)
HW	Hardware
ISP	Image Signal Processor
JSON	Javascript Object Notation
LXC	Linux Containers
MLIR	Multi-Level Intermediate Representation
NFV	Network functions virtualization
ONNX	Open Neural Network Exchange
OpenCL	Open Computing Language
OpenCV	Open Computer Vision
PCM	Pulse-Code Modulation
PDM	Pulse-Density Modulation
PoCL	Portable Computing Language
PoCL-R	Portable Computing Language Remote
SDN	Software-Defined Networking
ST	Similarity Transform
SoC	System-on-a-Chip
SPIR-V	Standard Portable Intermediate Representation V
SW	Software
TVM	Apache TVM project
VM	Virtual Machine
XRT	Xilinx Real Time library

2 OpenCL Execution Environment for Distributed CPS

Cyber-physical systems represent a wide range of applications with various needs on the input/output (sensor/actuator) throughput/latency and computation performance requirements. However, there are key aspects in CPSs that can be identified as common challenges that pose requirements to an execution environment and the software layer to support it:

1. Distributed execution. CPSs are typically composed of multiple sensors, actuators and processors which are connected toa wired or wireless network. Network in the loop imposes extra challenges to latency critical applications.
2. Computation device diversity. The devices in the CPSs must be considered *heterogeneous*. The degree of processor specialization depends on how stringent the non-functional requirements are. Especially in cases with high computational complexity is combined with energy efficiency demands, high degree of specialization in form of use of GPUs, HW accelerators, FPGAs, DSPs, ISPs and other devices tailored for a task or a set of tasks at hand is desired.
3. Hardware vendor independence. Vendor lock-in must be avoided in order to maintain competition pressure on the hardware provider side and also to ensure *supply security* in case a processor vendor is not able to supply processors. The supply side problems have been highlighted over the pandemic period with car manufacturers being bottlenecked by a lack of electronic components such as processors.

When designing the CPSoSAware execution and deployment environment, there was an early decision to focus on open multi-vendor adopted standards in order to address the challenges 2 and 3. OpenCL [OpenCL] was judged to provide the necessary support for diversity, since it's a standard carefully drafted by multiple important processor vendors in a common understanding. It is also not tied to a single device type as it can control devices ranging from multicore CPUs down to hardware accelerators and FPGA fabrics. Two other options were on table for the standard itself: Vulkan [Vulkan], which is even more popular than OpenCL, but unfortunately is very graphics and GPU leaning. Another is SYCL [SYCL], which is becoming increasingly popular, and stepping out of its original role of improved OpenCL C++ bindings to more towards a portability layer. However, we believe a low-level software portability layer API should be based on C instead of the more complex C++ language, therefore the choice of using OpenCL for this layer was retained.

However, the aspect that was lacking from the OpenCL standard and its implementations is distributed execution. OpenCL as a standard considers only single node computing, which posed a problem which needed to be tackled. This problem was solved by developing an OpenCL driver implementation that can mirror remote OpenCL devices efficiently and let them under the control of the local application. The initial work for this implementation was started in an ECSEL JU project FitOptiVis and it was adopted to the CPSoSAWare's specific needs which are described later in this section after providing the necessary information of the PoCL-R implementation itself.

2.1 Distributed OpenCL Implementation for Cyber Physical Systems

The aspects of PoCL-R most relevant to CPSoSAware's challenges are discussed in this subsection. For further details, the best source currently is a conference publication of PoCL-R published in 2021 [PoCL-R].

Unlike the previous distributed OpenCL attempts in the past, the PoCL-R focuses on latency and the edge cluster side scalability at the same time. Furthermore, it also provides support for low latency distributed streaming applications where data is read from a remote input device to the end user (client) device, which then needs to be further processed to produce the output. With PoCL-R, the input data can be streamed directly to the remote compute node, reducing the client's bandwidth use and overall latency.

Overall, a key benefit of PoCL-R is that the whole edge cluster workload distribution can be orchestrated from the client application logic side without application-specific server-side software, thanks to the generality and power of the heterogeneous OpenCL API. The focus is on minimizing the end-to-end latency to enable reactivity in challenging real-time control loops as well as enable scalable use of diverse compute resources in the cluster.

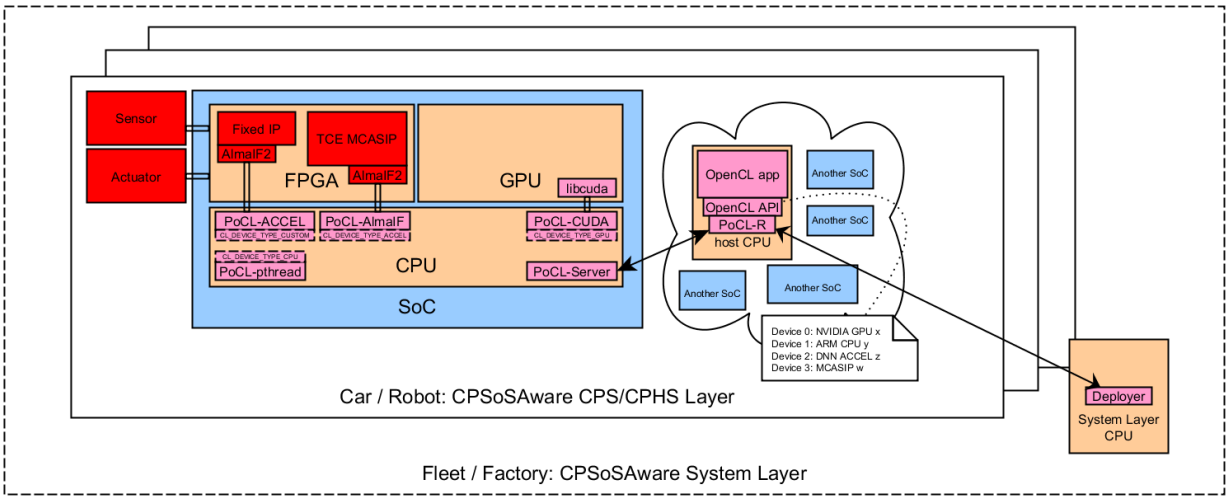

Figure 1: CPSoSAware distributed OpenCL-centric CPS software layer. A central host CPU controls the rest of the system from a single main application which uses the OpenCL API. PoCL-R, PoCL-accel and AlmaIF v2 are the central components of the project used to handle distributed computing, hardware interfacing and FPGA component wrapping, respectively.

The whole application logic is defined in a single host application, as specified by the OpenCL standard. The application includes both the main program running in the local device as well as the kernel programs that are executed on local, or in the case of PoCL-R, also the remote OpenCL devices. The OpenCL standard allows the kernel programs to be defined in a portable source code or an intermediate language, and alternatively using target-specific binary formats which enables various options for system-wide application deployment. The ability to store and deploy device-specific binaries can be used to bypass long synthesis steps at application runtime when using FPGA-based accelerators. This aspect is discussed more thoroughly in Section 5.3.

The runtime is implemented as a standard client-server architecture. The client of the architecture is implemented as a special remote driver in Portable Computing Language (PoCL) [PoCL], an open source implementation of the OpenCL API with flexible support for custom device backends. The remote driver acts as a ``smart proxy'' that exposes compute devices on a remote server through the OpenCL platform API the same way as local devices, making the use of remote devices in OpenCL applications identical to using local devices at the program logic level. Features of the remote devices depend on what their native drivers support.

One of the nice aspects of PoCL-R is that a host application using the OpenCL API can use PoCL-R as a drop-in implementation without recompilation. When the host application is linked against PoCL-R, OpenCL calls are made to the PoCL-R client driver, which in turn connects to one or multiple remote servers, each providing one or more remote compute devices. The remote servers can form interconnected clusters visible and controlled by PoCL-R as peers to avoid round-trips back to the client whenever synchronization or data transfers are needed between the remote devices. The application can identify remote devices by the device name string that contains an additional "pocl-remote". This allows optimising choices of command queues and kernel implementations.

The server side is a daemon that runs on the remote servers and receives commands from the client driver, and dispatches them to the OpenCL driver of the server's devices accompanied with proper event dependencies. The OpenCL devices can be controlled via a device-specific proprietary OpenCL driver by the daemon, or through the open source drivers provided by PoCL. This feature helps to provide maximal diversity in the set of devices that can be supported: The devices which have vendor-provided OpenCL drivers can be controlled through the proxy approach with those that lack vendor support augmented with open source drivers as backends in the PoCL framework.

The daemon is structured around network sockets for the client and peer connections. Each socket has a reader thread and a writer thread. The readers do blocking reads on the socket until they manage to read a new command, which they then dispatch to the underlying OpenCL runtime, store its associated OpenCL event in a queue and signal the corresponding writer thread. The server writer thread iterates through commands in the queue and when it finds one that the underlying OpenCL runtime reports as complete, it writes its result to the socket representing the host connection. Peer writers have separate queues that work similar to the server writer. Figure 2 illustrates this architecture and the flow of commands and data through it.

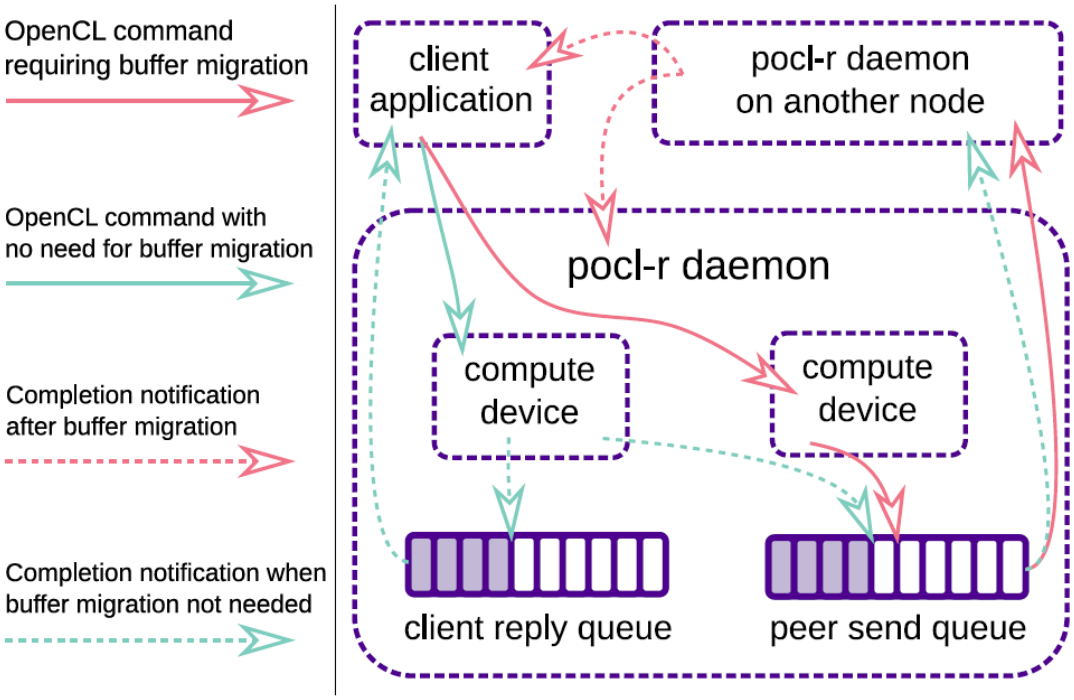

Figure 2: The information flow from an application to the PoCL-daemon and between remote servers. Two different commands are illustrated, one that transfers buffer contents from one remote node to another and one that doesn't.

2.1.1 Distributed Data Sourcing

The networking support of the software distribution layer's current implementation is based on the common TCP/IP software stack with the socket API used for software interfacing. However, since there is nothing tying the implementation to sockets, any networking layer that supports robust streaming for the control messages should be easily adoptable.

In CPSoSAware, the system can be a complex distributed platform of systems with heterogeneous interconnection with onchip, offchip and even network links. This requires careful consideration of how to source the data originating from a sensor and, on the other hand, how the actuators are controlled from their nearby devices. The trivial solution is to route all of the data through the central CPU, which naturally would quickly become a bottleneck for system scalability and latency. With the OpenCL 1.2's custom device feature it is possible to wrap arbitrary data sources to appear as devices in the OpenCL platform. Such devices can then utilize the P2P buffer migration functionality supported by PoCL-R to transfer input data directly to the compute device that needs it, rather than making an expensive round trip through the host application. This is utilized in CPSoSAware together with the capability of wrapping hardware IPs as custom devices which are controlled through the built-in kernels functionality, as described in Section 4.

2.1.2 Low-Overhead Communication

The base of the client-server communication is a pair of raw TCP sockets. One socket is dedicated to commands and the other to buffer data transfers, with their send and receive buffer sizes tuned for their respective purposes. To minimize latency on the network level, TCP fast retransmission is enabled for both sockets. While optimization of serialization protocols has been researched a lot and some extremely low-

overhead protocols such as FlatBuffers and MessagePack have emerged, using a separate wire format for communication still adds overhead both on the sending and receiving side. PoCL-R uses the in-memory representation of command structures as its wire format, thus minimizing the serialization step. The only added data is a fixed-size integer indicating the length of the next command structure. This is done based on the notion that there is only a small number of commands generated from OpenCL commands that require a massive command structure, while most commands can fit all information, they need in a few dozen bytes. Buffer contents are transmitted separately and the command structure only contains the transferable data size for each relevant buffer so even when commands transfer large amounts of buffer contents, the command structures themselves remain small.

The trade-off of this approach is that all remote servers as well as the client device running the host application need to have the same word byte order (endianness). In practice, we consider this not a noticeable limitation in today's hardware. If a need arises to handle mixed endianness platforms, it should be a straightforward modification with minimal performance impact; adding byte swaps for commands' data fields to the runtime is simple and, albeit cumbersome, likely has only negligible impact on overall performance. Additionally, the swap can be eliminated entirely at compile time if the sender and receiver machines' endianness is detected to be already the same as the host application's.

A bigger hurdle is the OpenCL C application code itself, as OpenCL has no knowledge about buffer contents' endianness and makes mixed endianness related swapping the application writer's responsibility: Applications meant to work on platforms with mixed endianness need their kernels to be adapted to account for the difference and swap the byte order of multi-byte values stored in OpenCL buffers when crossing devices with different byte orders.

2.2 The Work on Top of PoCL-R within CPSoSAware

The OpenCL-based distributed software stack was initially created in an ECSEL JU project FitOptiVis and brought to the CPSoSAware project as a basis for further extensions. The main extensions that have been implemented to PoCL-R in order to support the CPSoSAware's needs are following.

- Interfacing to FPGA components via the hardware component wrapper. The work conducted within Task 2.3 resulted in a new version of a hardware component interface called AlmaIF v2. This work was continued in this task by integrating it to the OpenCL-based software stack through a new PoCL-based driver. The driver called pocl-accel was initiated in WP2 and continued to the WP3's system scope. The driver received improvements in various aspects such as asynchronous command queue execution on the FPGA device side, interfacing to input devices with built-in kernels along with overall bug fixing and stabilization work.
- Remote FPGA execution. The hardware component wrapping work was combined with the remote OpenCL execution platform to provide an end-to-end solution where FPGA components can be invoked remotely through the PoCL-R-based software stack. This part required unexpected amount of effort due to new issues found when integrating the whole stack.
- System level demonstration. The system level demonstration of the OpenCL based software stack was conducted by showcasing it in the context of a hardware-in-the-loop simulation. This involved finding out the feasible integration points in the Carla simulator, which was used for the functional part of the simulation for the automotive pillar, as well as implementing an example proof of concept remote offloading case study. This resulted in the demonstrator video accompanied with this report and is described in more detail in the next section.

3 From Hardware-in-the-Loop Simulation to a CPS Deployable Program

This report accompanies a demonstrator video which shows how the OpenCL based distributed software stack can be used within a development cycle that involves hardware-in-the-loop (HitL) simulation. The demo video is downloadable here.

A problem with software development for a large CPS that is either not yet physically built or not available to available developers, is that timing accurate simulation of all parts of the system is unrealistically slow. It is typical that simulating even a simple processor can be three to four magnitudes slower than the real hardware, and a modern CPS can easily have tens of processors integrated in the system. Furthermore, the networking between the nodes of a distributed system adds to the simulation details, making the whole system simulation so slow that feasible software development or benchmark is simply not possible.

The idea of HiTL is that developers can gradually convert a functional simulation of the application to a deployable real-time implementation running in the target platform, testing the functionality of interest separately on the real hardware while providing inputs to it from a less timing accurate but functionally reasonably accurate simulation model. This enables getting feedback on hardware specific optimizations while continuously testing the integrated system within the functional simulation.

A key benefit of using the OpenCL-based stack for this purpose is that the developed system level application will be then deployable also on the real system, if it can run the OpenCL runtime as well as the hardware devices of interest are supported by OpenCL drivers.

3.1 CARLA and PoCL-R based HitL simulation example

For the demonstrator, we utilized the CARLA open-source simulator for autonomous driving research [CARLA] as a basis. The setup is illustrated in Figure 3. The simulation loop was extended with a spot where the developer can add OpenCL kernel launches that input sensory data and affect the control of the car. The calls are purely based on the OpenCL API, thus can be executed on any OpenCL supported driver. For the purposes of the HitL development, we utilized the PoCL-R driver to offload kernels from the CARLA to a development board accessible remotely across ethernet. The development board contained an FPGA SoC that represented the “real hardware” that could reside in a car for an algorithmic acceleration purpose. The CARLA simulator itself ran on a common desktop PC, and was considered as not interesting for accurate modeling since it models the environment and the rest of the car driving logic.

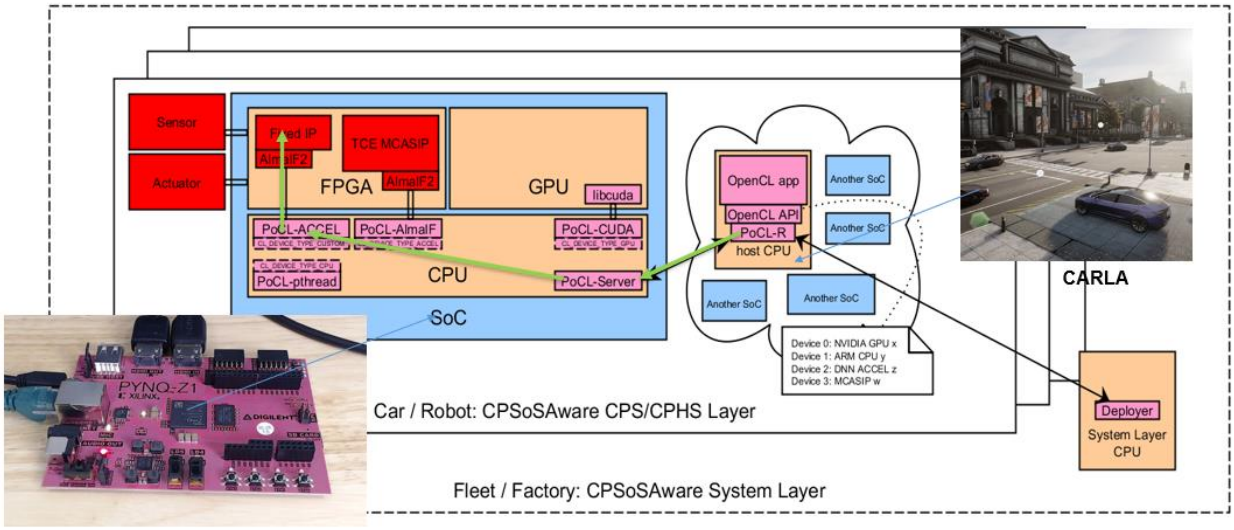

Figure 3: A CARLA and PoCL-R based hardware-in-the-loop setup for the automotive pillar development. CARLA represents the main program running the host CPU as well as models the car's physical environment. Calls to OpenCL API result in execution on the "real hardware", which is represented with an FPGA SoC development board in this example.

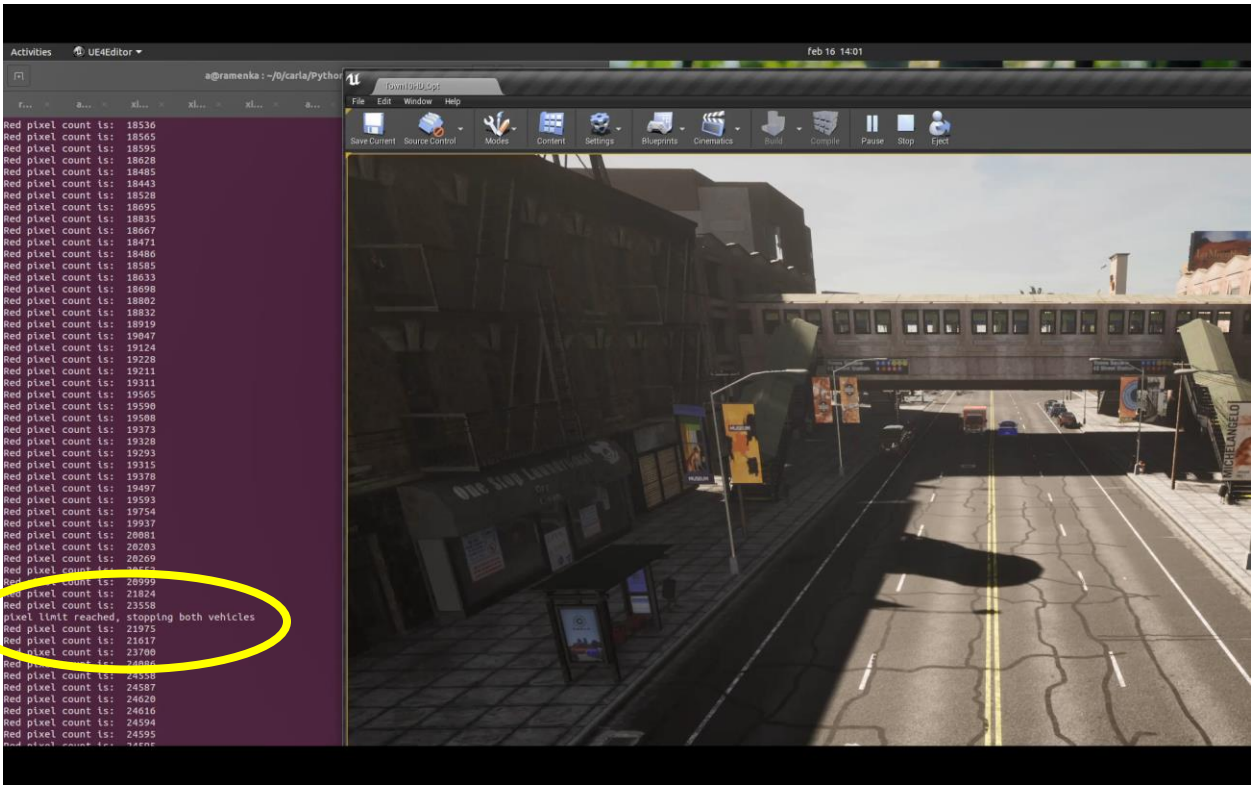

Figure 4: The CARLA/PoCL-R based HitL simulation in action. On the left, there are printouts to console of the dummy red color detector kernel's output. As it encounters the red ambulance, it reaches a threshold and stops the car. Circled in yellow is the notification of the detection happening.

This demonstrator served two main purposes 1) showcase the concept of HitL for CPSs when the compute devices can be accessed portably using a portability layer based on OpenCL API and 2) exemplify CPS software development and deployment using a centralized application binary running in a host processor that commands the other co-processors. The idea here is that the OpenCL-based acceleration layer can be easily converted to the final version due to the open standard API used.

3.2 Capturing the execution time of the kernel in the hardware of interest

In order to address a question in the M18 review of the project about a relatively high offloading latency reported in an intermediate presentation made of the software stack, it should be emphasized what is the interesting output from this type of HitL execution: The idea is to get accurate results of a (kernel) function's execution on a hardware device that is going to be in the final system. However, this HiTL-based development environment does not include network latency modeling so far, thus the time that it takes to interface from the functional simulation running in the PC through the network connection to the development board representing the "real hardware" should be excluded from the timings since the networking in the final CPS (e.g. a car internal network) will be different, and could be even an on-chip network in case the device is integrated in the same SoC as the host processor.

Figure 5-8 demonstrate how in the demo we get execution time feedback from the target hardware that is modeled with the Xilinx development board.

```
1644941822805314764 | EV ID 80 | DEV 2 | CQ 7 | read_buffer | running | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
1644941822805620197 | EV ID 80 | DEV 2 | CQ 7 | read_buffer | complete | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
DEP | EV ID 78 -> EV ID 80
1644941822832197484 | EV ID 81 | DEV 1 | CQ 6 | write_buffer | queued | MEM ID 12 | size=1920000 | host_ptr=0x7ffc8c28f590
1644941822832230357 | EV ID 81 | DEV 1 | CQ 6 | write_buffer | submitted | MEM ID 12 | size=1920000 | host_ptr=0x7ffc8c28f590
1644941822832532975 | EV ID 81 | DEV 1 | CQ 6 | write_buffer | running | MEM ID 12 | size=1920000 | host_ptr=0x7ffc8c28f590
1644941822833561582 | EV ID 81 | DEV 1 | CQ 6 | write_buffer | complete | MEM ID 12 | size=1920000 | host_ptr=0x7ffc8c28f590
1644941822833757909 | EV ID 82 | DEV 1 | CQ 6 | ndrange_kernel | queued | KERNEL ID 9 | name=downsample_image
1644941822833807852 | EV ID 82 | DEV 1 | CQ 6 | ndrange_kernel | submitted | KERNEL ID 9 | name=downsample_image
1644941822833825763 | EV ID 82 | DEV 1 | CQ 6 | ndrange_kernel | running | KERNEL ID 9 | name=downsample_image
1644941822833845319 | EV ID 82 | DEV 1 | CQ 6 | ndrange_kernel | complete | KERNEL ID 9 | name=downsample_image
DEP | EV ID 81 -> EV ID 82
1644941822832023465 | EV ID 84 | DEV 2 | CQ 5 | migrate_mem_objects | queued |  # MEMS 1 | MEM 0 ID 13 | FROM DEV 1 | TO DEV 2 |
1644941822834066365 | EV ID 84 | DEV 2 | CQ 5 | migrate_mem_objects | submitted |  # MEMS 1 | MEM 0 ID 13 | FROM DEV 1 | TO DEV 2 |
1644941822834123544 | EV ID 84 | DEV 2 | CQ 5 | migrate_mem_objects | running |  # MEMS 1 | MEM 0 ID 13 | FROM DEV 1 | TO DEV 2 |
1644941822844460856 | EV ID 84 | DEV 2 | CQ 5 | migrate_mem_objects | complete |  # MEMS 1 | MEM 0 ID 13 | FROM DEV 1 | TO DEV 2 |
DEP | EV ID 82 -> EV ID 84
1644941822845853004 | EV ID 83 | DEV 2 | CQ 7 | ndrange_kernel | queued | KERNEL ID 11 | name=pocl.countred
1644941822845860303 | EV ID 83 | DEV 2 | CQ 7 | ndrange_kernel | submitted | KERNEL ID 11 | name=pocl.countred
1644941822845915177 | EV ID 83 | DEV 2 | CQ 7 | ndrange_kernel | running | KERNEL ID 11 | name=pocl.countred
1644941822861152721 | EV ID 83 | DEV 2 | CQ 7 | ndrange_kernel | complete | KERNEL ID 11 | name=pocl.countred
DEP | EV ID 82 -> EV ID 83
DEP | EV ID 84 -> EV ID 83
1644941822862294385 | EV ID 85 | DEV 2 | CQ 7 | read_buffer | queued | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
1644941822862300037 | EV ID 85 | DEV 2 | CQ 7 | read_buffer | submitted | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
1644941822862349225 | EV ID 85 | DEV 2 | CQ 7 | read_buffer | running | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
1644941822862588497 | EV ID 85 | DEV 2 | CQ 7 | read_buffer | complete | MEM ID 14 | size=8 | host_ptr=0x7ffc8c28cdf8
DEP | EV ID 83 -> EV ID 85
```

Figure 5: Example of an execution trace output from the CPS software execution.


```
cat":"pocl","id":14,"name":"migrate_mem_objects","pid":2,"tid":5,"args":{"mem_id":"13","src_id":"1","dst_id":"2"},"ph":"E","ts":161896}
cat":"pocl","id":15,"name":"read_buffer","pid":2,"tid":7,"args":{"mem_id":"14"},"ph":"B","ts":181012}
cat":"pocl","id":15,"name":"read_buffer","pid":2,"tid":7,"args":{"mem_id":"14"},"ph":"E","ts":181255}
cat":"pocl","id":16,"name":"write_buffer","pid":1,"tid":6,"args":{"mem_id":"12"},"ph":"B","ts":202087}
cat":"pocl","id":16,"name":"write_buffer","pid":1,"tid":6,"args":{"mem_id":"12"},"ph":"E","ts":202845}
cat":"pocl","id":17,"name":"downsample_image","pid":1,"tid":6,"args":{"kernel_id":"9","kernel_name":"downsample_image"},"ph":"B","ts":203068}
cat":"pocl","id":17,"name":"downsample_image","pid":1,"tid":6,"args":{"kernel_id":"9","kernel_name":"downsample_image"},"ph":"E","ts":203088}
cat":"pocl","id":18,"name":"pocl.countred","pid":2,"tid":7,"args":{"kernel_id":"11","kernel_name":"pocl.countred"},"ph":"B","ts":216634}
cat":"pocl","id":18,"name":"pocl.countred","pid":2,"tid":7,"args":{"kernel_id":"11","kernel_name":"pocl.countred"},"ph":"E","ts":231946}
cat":"pocl","id":19,"name":"migrate_mem_objects","pid":2,"tid":5,"args":{"mem_id":"13","src_id":"1","dst_id":"2"},"ph":"B","ts":203172}
cat":"pocl","id":19,"name":"migrate_mem_objects","pid":2,"tid":5,"args":{"mem_id":"13","src_id":"1","dst_id":"2"},"ph":"E","ts":214624}
cat":"pocl","id":20,"name":"read_buffer","pid":2,"tid":7,"args":{"mem_id":"14"},"ph":"B","ts":243307}
```


Figure 6: The trace can be converted to a JSON format that can be input to the Chromium browser for visualization.

Figure 7: The "swimlane" execution time profile visualization in Chromium.

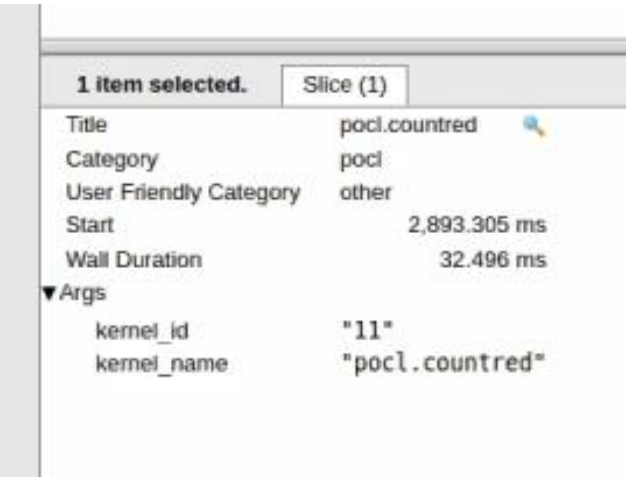

Figure 8: Details of a kernel execution when a kernel launch is selected from the "execution swimlane".

3.3 The implementation: Integration of PoCL-R to CARLA

The integration of PoCL was done on the simulator side. libpocl is built as static library and linked into the Carla plugin (see Figure 9). There is additional code that implements an example: a camera sensor that uses OpenCL to process images.

The OpenCL part of the sensor is located in file *LibCarla/source/carla/OpenCL/OpenCLcontext.cpp*, in class OpenCL_Context. This class contains the OpenCL code of two kernels; the count-red-pixels kernel, and the downsample-image kernel. There are two important methods: 1) initialize() which will try to find two OpenCL devices (with fallback to one), allocate the necessary buffers; and 2) processCameraFrame() which takes as input captured camera frame and outputs a red pixel count. In processCameraFrame(), first the data is uploaded to the GPU, then the downsample kernel is ran on the GPU, and finally the downsampled image is sent to the FPGA, where the second kernel (to count the red pixels) is executed.

The Carla part of the sensor is located in *Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor*, in files *SceneCaptureCameraOCL.{h,cpp}* and *PixelReader.h*. ASceneCaptureCameraOCL is a subclass of SceneCaptureSensor. It is an Actor similar to the SceneCaptureCamera Actor, but instead of capturing the frame and immediately sending data to the client stream, this sensor will capture frame data and run OpenCL kernels on it. Capturing of the data is done in the PixelReader class, with the OpenCLPixelsInRenderThread method, which captures the frame data and calls on an OpenCL_Context instance to process the captured frame. The result (red pixel count) is then serialized into PixelCountEvent and sent in the client stream.

PythonAPI/examples/camera_opencl2.py contains the python code that demonstrates the usage of SceneCaptureCameraOCL sensor. This demo sets up a specific scene in CARLA. It first creates two vehicles, sets their colors to red and blue respectively, spawns them at and drives in opposite direction in neighbouring lanes. The blue vehicle has a camera on top, an instance of the SceneCaptureCameraOCL. The camera has a callback registered, which will be called every time it receives PixelCountEvent in the client stream. The callback will then stop both cars when the red pixel count reaches a threshold that would indicate the red car is very near the blue car.

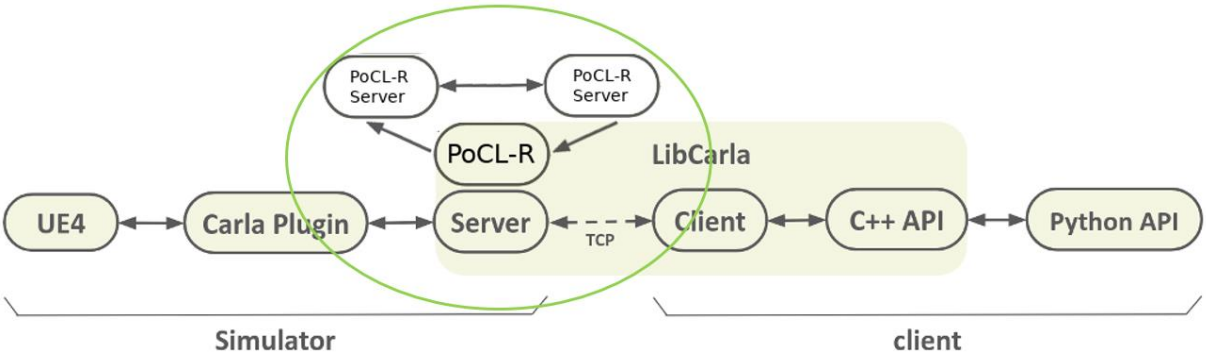

Figure 9: The integration point of PoCL-R to CARLA. The kernel launches were integrated to the simulation server side.

4 Portable Accelerator Utilization via OpenCL Built-in Kernels

Abstracting hardware accelerators in a portable fashion is still an open research challeng. The proposed CPSoSAware CPS oftware stack is based on OpenCL, which luckily has an abstraction for “black box” functionality that can cover non-programmable functionality in any kind of co-processors and input/output devices. A very underused feature in the standard introduced in OpenCL 1.2 [OpenCL] is the concept of a *custom device* which abstracts hardware accelerators which do not necessarily support online compilation of any OpenCL C functions, but instead present a set of *built-in kernels* for invoking interfacing with the functionality of the devices. In this project we build on this feature for both fixed function hardware acceleration purposes and for making sensors and actuators accessible from asynchronous application descriptions defined using OpenCL command queues.

To validate the remote offloading of Deep Neural Networks (DNN) via OpenCL built-in kernels for different backends implemented in this project, two demonstrators were built: An audio classification network for real-time environment classification in a tiny energy optimized edge processor and fast semantic segmentation for cityscapes utilizing a desktop scale GPU for acceleration. The audio classification demonstrator was used to demonstrate access to a real time input device as well as how the built-in kernels can drive fixed function FPGA-based hardware accelerators in the backend. The fast semantic segmentation case demonstrates accessing CUDA-based accelerators on Nvidia GPU’s Tensor Cores through the abstraction layer provided by the OpenCL built-in kernels.

4.1 End-to-End Environmental Sound Classification with an 1D Convolutional Neural Network

In this demonstrator, the idea was to show how the built-in functions can drive fixed function FPGA-based hardware accelerators in the backend. The demonstrator uses a small 1D convolutional neural network (CNN) as an example. The network is based on an end-to-end environmental sound classification using a 1D CNN. With an end-to-end approach the CNN learns the representation directly from the raw audio signal and outputs an environmental class label corresponding to the target prediction and is thus better suited for real-time applications. The target dataset used contained 4 second audio clips from 10 different environments and for the application the CNN was taught to classify the environment using half-second audio samples, so the input size was 8 000 samples with a sampling rate of 16kHz. The target hardware is a tiny DSP with only integer arithmetic support, so the model was further quantized using post-training 8-bit integer-only quantization. The final quantized model with an example quantized input and true and predicted value is shown in Figure 10.

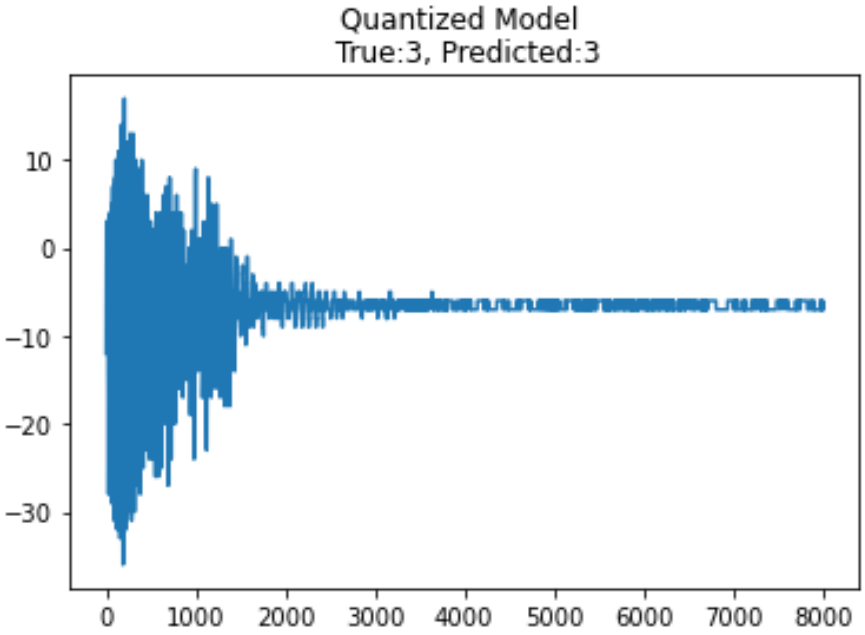

Figure 10: The End-to-end environmental sound classification 1D CNN with 8-bit integer quantization.

This network was offloaded to an FPGA implementation of the DSP using PoCL-R with the help of using TVM code generation for the kernels. The initial implementation used only optimized custom kernels for each layer of the network. For this demonstration, 3 built-in kernels were written for the convolutional, dense and maxpool layers of the network, which each utilize the FPGA-based hardware accelerators. An example built-in kernel description is shown in Table 1.

Table 1: Built-in functions for the 1D CNN with descriptions.

Built-in kernel name	Description
pocl.nn.conv2d_int8_strided_relu	2D strided convolution with 8-bit signed integers, quantization rescaling and relu.
	Grid dimensions:
	1. Input width 2. Input height 3. Convolution count
	Input and output descriptions:
	1. IN: char* Input read buffer in NCHW format 2. IN: char* Convolution read kernels in NCHW format 3. OUT: char* Output write buffer 4. IN: integer* 32-bit integer bias for each convolution kernel 5. IN: integer* 32-bit integer scale value calculated from the 32-bit floating point scale value for each convolution

	6. IN: integer* 32-bit integer shift value for each scale value 7. IN: char* 8-bit zero-point value for each convolution 8. IN: unsigned integer input width 9. IN: unsigned integer input height 10. IN: unsigned integer input depth 11. IN: unsigned integer convolution width 12. IN: unsigned integer convolution height 13. IN: unsigned integer stride in w direction 14. IN: unsigned integer stride in w direction

Each layer of the original implementation was replaced with a call to a built-in kernel which was then implemented by the soft DSP co-processor implemented on the FPGA fabric. The demonstrator was able to run on the simulation environment but validation and deployment for the FPGA is the planned next step.

4.2 Fast Semantic Segmentation with MobileNetV3

The second demonstration application for DNN offloading through OpenCL built-in functions used the MobileNetV3-Small architecture for semantic segmentation in cityscapes. Example segmentation input and output for the network can be seen in Figure 11.

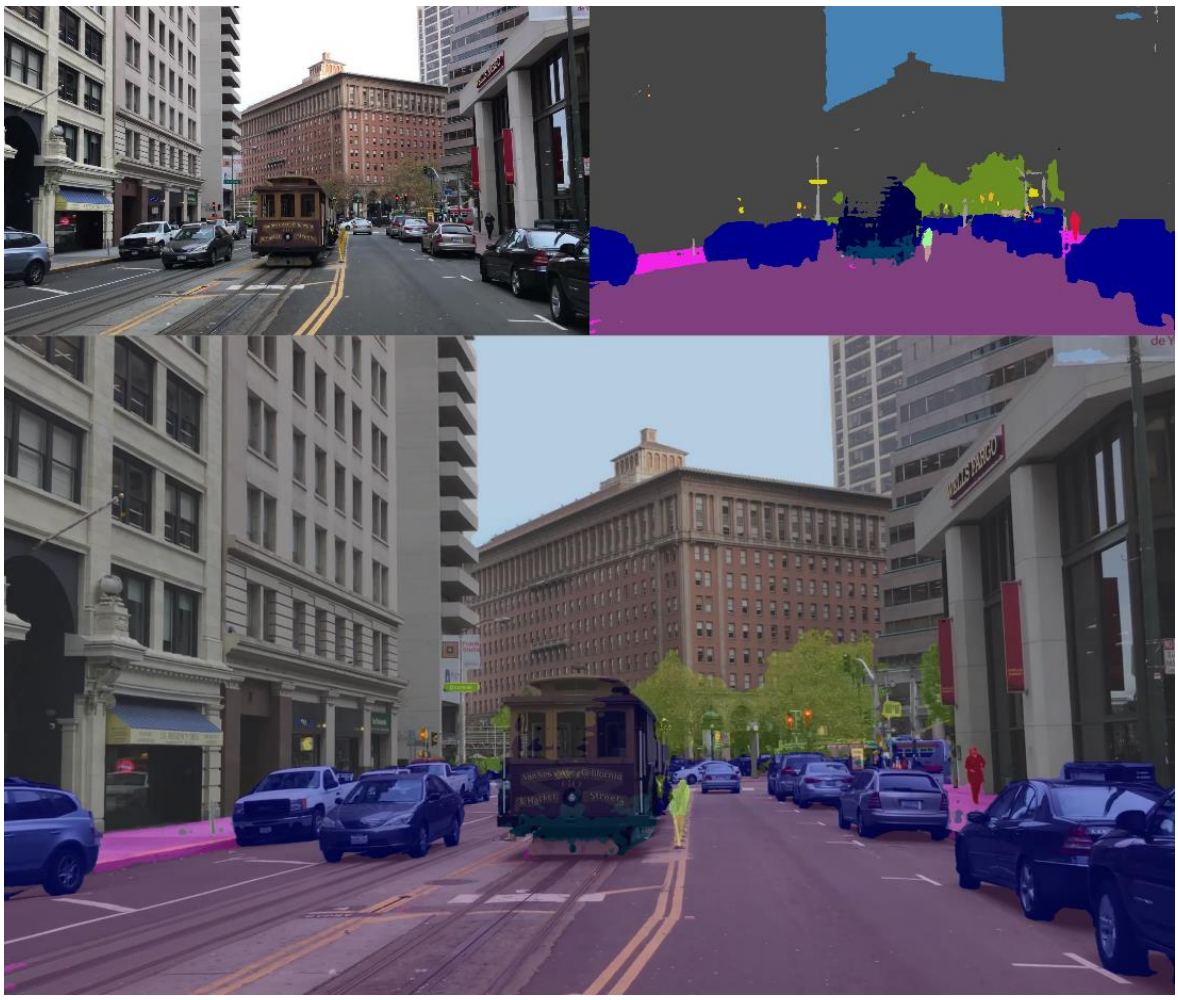

Figure 11: Fast semantic segmentation with MobileNetV3-Small model.

In this demonstrator, the target hardware was NVIDIA GPUs equipped with TensorCores for DNN acceleration. The model was first loaded to TVM through its ONNX-frontend. After this, OpenCL code was generated for the network with TVM. The DNN was then offloaded to PoCL with TVM with a modified runtime to replace the generated OpenCL kernels with the implemented built-in kernels for the device.

Layers without built-in kernel implementations available in the device used custom generated OpenCL C code as a fallback implementation. Better way for general purpose to replace the custom OpenCL kernels with built-in functions would be to handle the built-in kernel definitions already before generating the OpenCL code in TVM's intermediate representations with a custom tool. But at the time of this writing, for this demonstration just modifying the runtime was enough to showcase the prove that the remote offloading of DNNs works on top of the built-in kernel based approach. Example built-in kernel for demonstrating the MobileNetV3 architecture is in Table 2.

Table 2: Example built-in function for the MobileNetV3 model.

Built-in kernel name	Description
pocl.nn.conv2d_strided	2D convolution with 32-bit floating point values with strides

	Grid dimensions: 1. Input width 2. Input height 3. Convolution count
	Input and output descriptions: 1. IN: float* Input read buffer in NCHW format 2. IN: float* Convolution read kernels in NCHW format 3. OUT: float* Output write buffer 4. IN: float* bias value for each convolution kernel 5. IN: unsigned integer input width 6. IN: unsigned integer input height 7. IN: unsigned integer input depth 8. IN: unsigned integer convolution width 9. IN: unsigned integer convolution height 10. IN: unsigned integer stride in w direction 11. IN: unsigned integer stride in w direction

The built-in kernels can be written in CUDA with programmatic access to Tensor Cores through its Warp Matric Multiply Accumulate API to perform convolutions as matrix multiplications. This example demonstrates that the OpenCL built-in kernels can access also NVIDIA accelerators for Tensor Cores. At this point the demonstrator was able to run with NVIDIA GPU through PoCL libcuda and software OpenCL kernels from TVM.

4.3 Planned Work for Portable Programming Hierarchical Abstractions

The two demonstrators built were done as a proof-of-concept and still need to be evaluated. Moreover, the portability of the demonstrators is to be evaluated on multiple different backends with the same built-in kernel abstractions. Also, the proof-of-concept implementations on DL compiler TVM should be generalized to an abstraction layer with an intermediate representation (IR). This could be done for example utilizing custom operators on TVM. Another interesting implementation could be done by defining a Multi-Level Intermediate Representation (MLIR) dialect. Furthermore, here the built-in functions would work as the operations defined for the dialect but also able to define hierarchical structure of the operations e.g., the example built-in kernels describing Convolution-layers in the demonstrators would consists of multiply accumulation and addition operations. The abstractions could also use and extend the gpu dialect of MLIR for the definitions of the OpenCL built-in kernels as the dialect offers attributes and operations for defining Open CL kernels.

5 Distributed and Portable FPGA Deployment and Execution

FPGA is one of the devices we wanted to support to process the kernels offloaded from the CPS applications through command queues of OpenCL built-in kernels. To make it possible to integrate various different hardware IPs on the FPGA fabric, the common interface AlmaIF v2 was developed further in this work package. This work is direct continuation of the work initiated in D2.3 for hardware component wrapping purposes.

The main advancements made since D2.3:

- Support for the device to "pull" data from external physically contiguous memory
- On-chip event synchronization and data sharing support
- Microphone input device created with a combination of RTL and HLS.
- Support for an instruction-set simulator-based device

5.1 Carla Demonstrator Hardware

The hardware kernel for the Carla HitL demonstrator was executed on a small FPGA development board (Pynq-Z1). The task of the hardware accelerator was to read in the pixel data of the frame and count the number of red pixels in image. The red counting kernel was implemented as a custom operation of an Application Specific Instruction-Set Processor (ASIP) designed with the OpenASIP tools. The operation was described as a Directed Acyclic Graph (DAG) primitive operations. This DAG was used as input for the generation of a processor function unit. Figure 12 shows the system level view of the ASIP-based built-in kernel implementation and its interfacing.

Since the computation done here was so simple, the data transfer between the DDR memory of the PYNQ-Z1 board and the accelerator device on the FPGA had to be designed carefully to not become a bottleneck. A simple memcpy by the host CPU to the onchip-memory of the accelerator was deemed to be too inefficient, so basic DMA functionality was implemented. This requires for the data to be in a physically contiguous area of the memory, from where the accelerator can then pull it with burst transfers to its own wide FIFO.

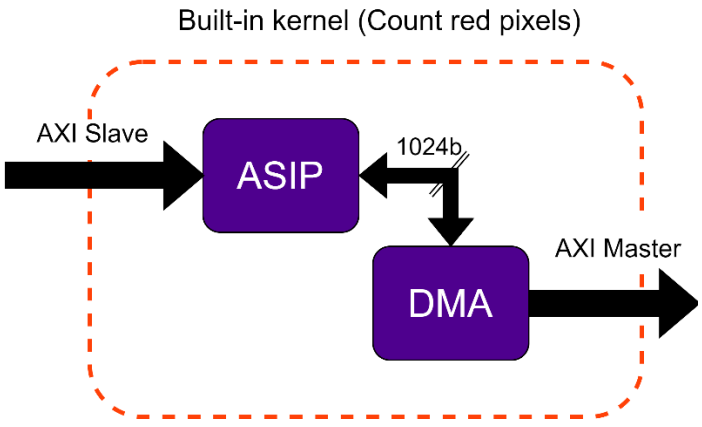

Figure 12: The system view of the ASIP-based built-in kernel implementation used in the HitL demonstrator.

In this demonstrator, the accelerator processed the data in the FIFO, counting the red pixels in the image data and outputting the red pixel count as a result value. The red counting kernel here was used merely as a demonstrator of any computation. Once the frame data has been pulled into the accelerator, any kind of computation is possible. In more realistic use cases, the computation/communication ratio would probably be higher.

5.2 On-chip Synchronization and Data Sharing

On a single FPGA programmable logic area, we can have multiple devices that work together as a task pipeline. The devices on the same FPGA have a very cheap way of accessing each other's memories through the on-chip bus. That can be exploited to allow the devices to share data with each other and progress independently without having to involve the host CPU for synchronization. This was implemented with barrier packets in the command queue. The host CPU will check for dependencies between the kernels it's launching on these different devices and write a barrier packet to describe the dependency. Then the later device that is waiting on the previous device to complete, will block on the barrier packet until the previous device has finished.

On-chip synchronization was tested with a simple dual-buffering configuration with two devices. First a microphone input device streams in audio data from a microphone and writes it to on-chip memory. Once the buffer is full, the first device sets a completion signal. A pointer to this completion signal is included in the barrier packet of the second device. Once the second device notices that the completion signal is set, it will continue past the barrier packet. After the barrier packet in the command queue, the host CPU has earlier put the data processing kernel, that the second device then executes. In this example it was a convolution kernel. Similar synchronization can be used to relaunch the microphone input device to rewrite the shared buffer once the input data is no longer needed by the second device. These dependencies are defined in the OpenCL application as event dependencies between the two command queues.

Data sharing between two on-chip devices was implemented by forwarding the knowledge that the two devices can access each other's memories to PoCL. Once PoCL knows that the devices share memory, it can skip the automatic memory migrations that would otherwise be generated for two devices sharing an OpenCL buffer. Therefore, the shared memory buffers are only allocated once to the device that first uses them, and both devices can directly use the same buffer.

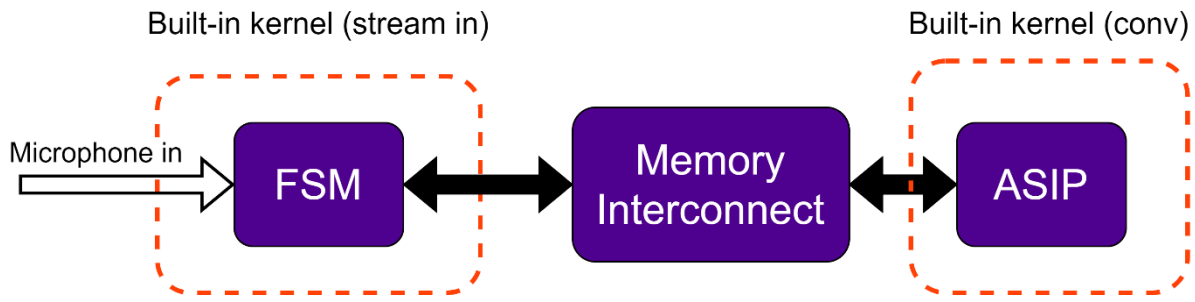

Figure 13: The system level view of the audio demonstrator with a built-in kernel used for data input.

The microphone in -signal in the Figure 13: The system level view of the audio demonstrator with a built-in kernel used for data input. is an AXI stream consisting of pulse-code modulated (PCM) 8-bit signed values at a sampling rate of 16kH. The PCM-stream was created with a custom RTL design to sample the PYNQ-Z1's MEMS pulse-density modulated (PDM) microphone. Then Vivado-provided ips were used to perform a PDM-to-PCM-conversion by low-pass filtering and decimating the signal to the desired sampling rate and value range. For the purposes of this demonstrator the microphone sampling and PDM-PCM-conversion was left out from the built-in kernel abstractions. From the point-of-view of the OpenCL driver and the OpenCL application, the incoming PCM AXI stream is thought to be already present on the device always producing microphone values to the input stream port of the FSM-device.

The blue FSM-device in Figure 13: The system level view of the audio demonstrator with a built-in kernel used for data input. was implemented using Vitis HLS tool with C-language input. The HLS tool creates a simple finite-state machine (FSM) that reads in the AXI stream and writes it to on-chip memory. In addition, it implements the barrier packet processing protocol described above. The FSM device can be thought of as a stream-to-memory-mapped-component that implements AlmaIF-interface. The AlmaIF interface is important here because it allows for the device to be directly plugged to to the PoCL's accel driver.

5.3 Design of a Bitstream Database with Automatic FPGA Programming Support

Previously the FPGA devices were presumed to be already running the bitstream that implemented the built-in kernels desired in the program. A list of built-in kernels running on the device was required to be given by the user as an environment variable. This is not a very scalable way of doing it, so an idea for a bitstream database was drafted and an implementation for a simple proof-of-concept was started. The database structure is shown in Figure 14.

This database would be accessed either remotely or locally. At the moment only the local database was considered, but the thought is to make it easily transformable into a remote database. The remote database is important since the bitstream-files can consume significant amount of data storage when considering different built-in kernel variations for various FPGA devices.

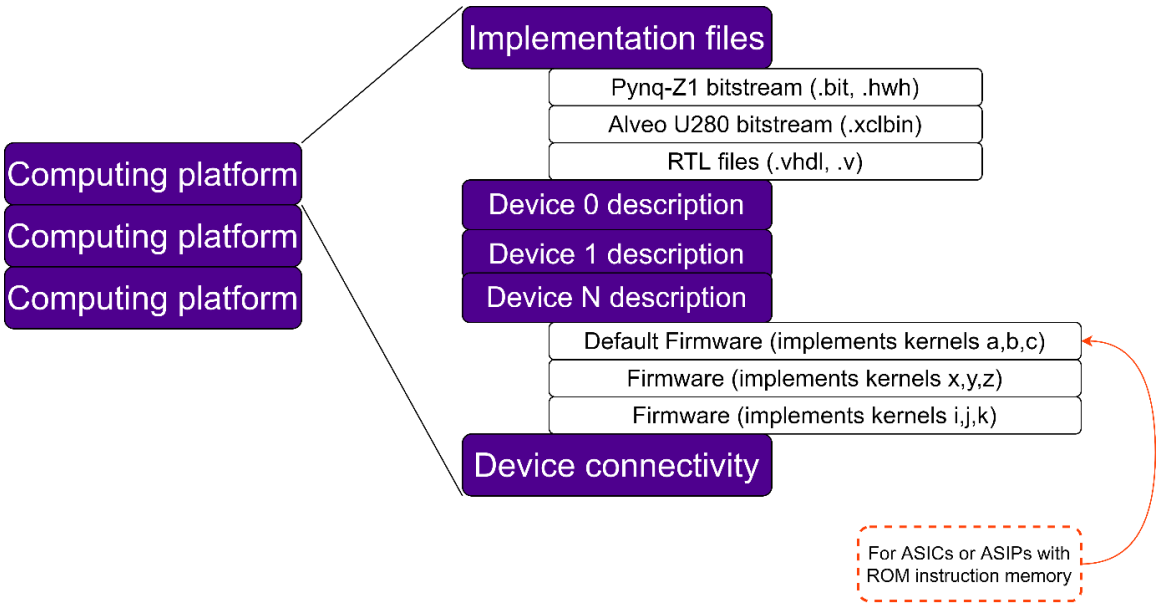

Figure 14: The FPGA bitstream registry structure.

The database consists of a number of computing platforms, which can be visualized as a block design of one or more interconnected devices. The computing platform has associated implementation files for various FPGA devices and possibility to include a collection of RTL-files to be used for RTL-simulation. All these different implementations of the same computing platform should have exactly the same structure and the functionality, the only difference being the FPGA device or simulator they have been synthesized for.

When the user wants certain built-in kernels, the PoCL's accel-driver will parse through this database and find the computing platform with the correct devices to implement those kernels. The computing platform here can include one or multiple devices. In the future, it will be worth it to implement partial reconfiguration support to be able to switch out the devices in the computing platform independently. At the time of writing, this feature was not yet developed, so the driver will always switch out a complete bitstream file.

In the database every device has some firmware. Even non-programmable, fixed-function accelerators are described here to have a "default firmware", which just means that the driver doesn't have to upload anything to the device instruction memory. Therefore the "default firmware" doesn't contain any program files, but rather just a listing of the built-in kernels that device implements by default. For programmable accelerator use-case we implement support for multiple different device firmwares that can be switched in depending on which kernels we want to execute. This allows for the device firmwares to be highly optimized (e.g. hand-optimized assembly) for the built-in kernel they implement without having to be compiled from more generic input language, e.g. OpenCL C or SPIR-V [SPIRV].

The proof-concept implementation for the database was planned to be a hierarchical set of directories with associated json-files describing the directory contents. The PoCL's accel-driver will parse the json-files to get the same structural view of the database as shown in Figure 14: The FPGA bitstream registry structure. The initial implementation will then look for a computing platform that could execute the set of built-in kernels the user has requested and upload the correct bitstream for the FPGA device in use. It's clear that

there might be a possibility of multiple computing platforms each being able to execute the desired kernels. Some of these options could be better in terms of performance or power. In the future a more intelligent heuristic will be developed that will be able to map the task pipeline to the best possible computing platform.

Implementation of this proof-of-concept database is still in progress with the goal to finish it for WP5 integration.

5.4 Simulating built-in kernels

To save developer headaches, it is a good idea to confirm the functionality of the built-in kernels on a simulation before deploying it on an ASIP running on an FPGA. For this reason, a simple way of plugging an ASIP instruction set simulator into the driver was developed. From the point of view of the driver, there is very little difference to the actual device. The memory reads and writes done by the driver are simply redirected to inside the simulator. Supporting multiple simulator devices with direct communication between them was left out for now. To be able to transparently support simulation of built-in kernels, all the necessary architecture description files will have to be included in the FPGA bitstream registry for all programmable devices.

6 Deploying the Software Stack to Distributed CPSs

In this section, the prototype containerised environment is described. The prototype is suited to hosting PoCL-R client and PoCL-D that can access underlying hardware in remote nodes. The selection of containerized environment, instead of the popular Docker, was due to its lightweight nature. The prototype has been actualized via Linux containers (LXC/LXD). The setup was tested successfully in remote configurations, where the controller node was located in Athens and the target nodes were located in Cyprus, in 8Bells' computing cluster. OpenCV has been used to check the current PoCL setup. Out-of-lab testing of the distributed OpenCL runtime has been tested successfully.

6.1 Selection of LXC containers over Docker

Linux Containers (LXC) [LXR] in master-slave configurations were chosen for the development, instead of the popular Docker [DOCKER], primarily as a lightweight alternative. LXC is an OS-level virtualization technology that enables the creation and execution of multiple Linux operating systems (OS) simultaneously on a single Linux machine (LXC host), providing at the same time a set of tools for container management, on top of available templates for the creation of a virtual environment of Linux OS. On the other hand, Docker focus is on providing an isolated environment for running single applications, provided as an open-source containerization technology. It comes with the Docker Engine that enables the creation, execution, and distribution of containers. There are commonalities between LXC and Docker, as regards to their architecture and usage. However, they also differ in many ways.

LXC falls under the umbrella project linuxcontainers.org, which includes also LXD and LXCFS. LXC [LXR] is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers. LXD [LXR] is a next generation system container and virtual machine manager. It offers a unified user experience around full Linux systems running inside containers or virtual machines. LXD is image based and provides images for a wide number of Linux distributions. It provides flexibility and scalability for various use cases, with support for different storage backends and network types and the option to install on hardware ranging from an individual laptop or cloud instance to a full server rack. Finally, LXCFS is a simple userspace filesystem designed to work around some current limitations of the Linux kernel.

Docker containers do not behave like lightweight VMs and can’t be treated, while they are restricted to one application (Php, python, lamp, Nginx), in contrast with LXC containers. LXC containers allow SSH connections for log in, they can be treated as an OS, and ensure the expected performance of the applications and services installed. These are not offered as options from Docker containers, since Docker base OS template is pared right down to one app environment and doesn’t have a proper init or support things like services, daemons, Syslog, corn, or running multiple applications. Key differences between LXC and Docker are summarised in Figure 15.

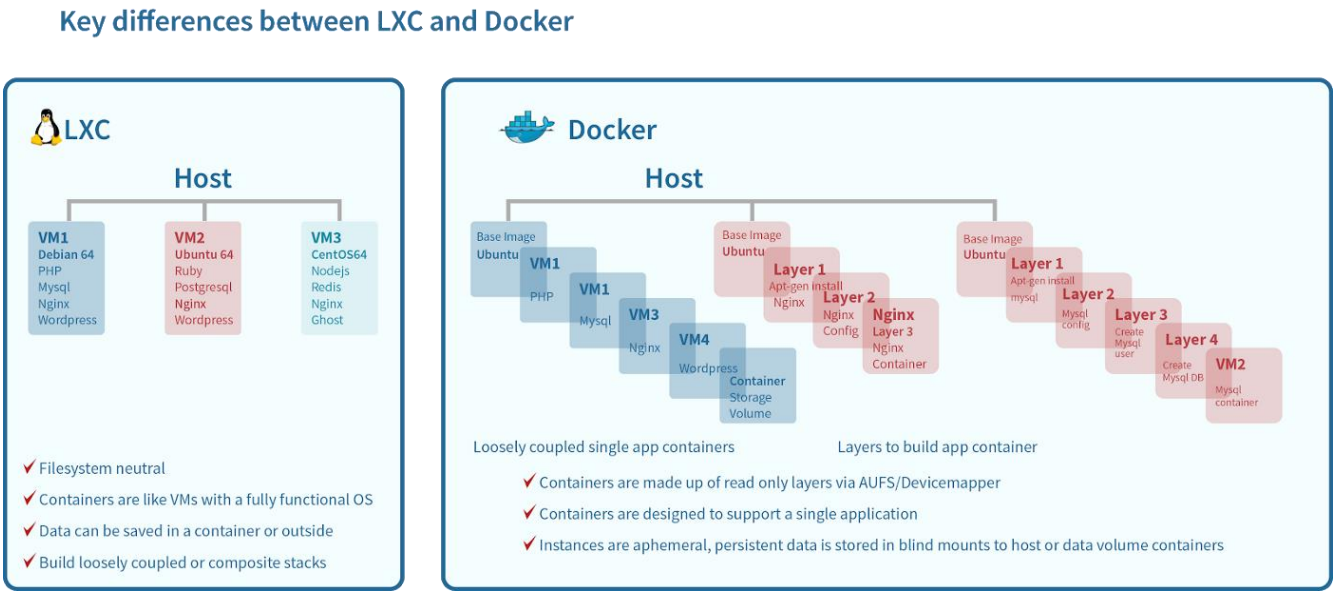

Figure 15. Key differences between LXC and Docker. Source: https://medium.com/@harsh.manvar111/lxc-vs-docker-lxc-101-bd49db95933a.

Table 3 below presents a comparison between LXC and Docker containers, in terms of host machine utilisation, simplicity, speed, security, ease of Use, scalability and tooling.

Table 3: Comparison of LXC containers vs Docker containers.

	LXC containers	Docker containers
Host Machine Utilisation	Using Linux kernel features to create isolated processes and file systems	Defined by sharing the kernel of the host system, offering a better alternative to VMs
Simplicity	More flexible in design, making simple the configuration and installation of anything that could be installed also in VMs. Leverages Kernel features like chroot, cgroups, and namespaces, to create an LXC virtual environment.	Docker containers were designed specifically for microservices applications, which differentiates them significantly from VMs.
Speed	LXC features fast boot times when compared to a virtual machine – it doesn't need to package an entire OS and a complete machine setup with network interfaces, virtual processors, and a hard drive.	Docker containers are also lightweight, which contributes significantly to their speed. Since Docker can run on an existing OS that is already initialized, it is possible to boot a container from its image almost instantly.

	LXC containers	Docker containers
Security	LXC is enriched with security configurations such as group policies and a default AppArmor profile to protect the host from accidentally misusing privileges inside the container.	Docker separates the operating system from the services running on it to ensure secure workloads, but the fact that Docker runs as root can increase exposure to malware.
Ease of Use	Installing LCX is pretty easy, as it only requires recent versions compatible with the existing Linux distribution. Additionally, LXC runs a standard operating system unit for each container, thus making it easy and manageable to migrate from bare metal or virtual machine servers, instead of moving to Docker containers.	Getting started with Docker is straightforward since it is an open source container platform, however, OS that support containers natively already installed.
Scalability	LXC is less scalable compared to Docker. Its images are not as lightweight as those of Docker.	With Docker, functionality of applications can be divided into individual containers. Then, linking these containers together and creating an application, allows for the independent scaling of components.
Tooling	Available LCX tools for managing tasks like creating, launching, deleting LXC containers, as well as reusing automation scripts already utilised on bare metal, VMs or other virtual environments, thus providing enhanced portability and applications migration.	Docker's command-line interface (CLI) provides control over containers, allowing to list and manage images, while using Docker registry allows accessing and distributing images for frequently used applications.

Both LXC and Docker are effective container technologies, and the choice between them depends on the development needs, as suggested by recent research [WRKP19, MSKU20, SLDV21]. The LXC containerized environment was chosen in this case mainly because it provides operating-system-level virtualization via a virtual environment, which contains its own userspace and processes. Linux containers depend on the functionality of the Linux kernel to isolate containers. The host CPU divides memory allocations into namespaces to control RAM and CPU usage, thus we have improved performance since LXC bundles use only the required application/OS. Furthermore, LXC excels in running multiple Linux distributions on a single server, while gives a lot of control over the features it is based on (namespace, cgroup, chroot, etc). Another advantage is that LXC makes easy to control the virtual environment, while provides increased app portability and makes easy their distribution inside containers. Finally, test upgrades and changes to a suite of applications can be done easily, while launching new instances is quick.

6.2 Deployment

The prototype containerized environment developed is suited to hosting PoCL-R client and PoCL-D that can access underlying hardware in remote nodes, which consists of two LXC containers that communicate seamlessly. The first LXC container contains a GPU that the second container "slaves", thus providing a master-slave configuration. In the following paragraphs details regarding system configuration, connection details, and the PoCL script are presented.

6.2.1 System configuration

The system's configuration includes:

- One target node with CPU Intel(R) Core (TM) i5-9600K CPU @ 3.70GHz and GPU GeForce GTX 1060 3GB (henceforth referred to as **slave node**) – currently located in Cyprus.
- One app-running node without access to GPU (henceforth referred to as **controller node**) – currently located in Athens.

Slave node spawns Ubuntu-based LXC container called slave1. Slave node has a publicly exposed IP and slave1 is on a bridge network with the slave node. Controller node spawns Ubuntu-based LXC container called controller1. Controller1 can access (via bridge with controller node) the outside world.

6.2.2 Connection details

Slave1 listens to all incoming traffic on ports 10998, 10999 and 11000. Slave node also listens on ports 10998, 10999, 11000 and forwards all incoming traffic to the slave1's respective ports. Controller1 sets environmental variables so that PoCL-remote searches for the slave node's publicly exposed IP – traffic will be redirected to slave1. Port forwarding rules are presented in the table below.

Table 4: Port forwarding rules.

```
sudo iptables -t nat -L -n -v

sudo iptables -t nat -A PREROUTING -p tcp --dport 10998 -j DNAT --to-
destination 10.140.146.96:10998

sudo iptables -t nat -A PREROUTING -p tcp --dport 10999 -j DNAT --to-
destination 10.140.146.96:10999

sudo iptables -t nat -A PREROUTING -p tcp --dport 11000 -j DNAT --to-
destination 10.140.146.96:11000

sudo iptables -t nat -L -n -v
```

6.2.3 PoCL script

The instance is launched from an nvidia-ready image. To launch PoCL, the following commands are used.

Table 5: PoCL launch commands.

```
lxc launch name-of-nvidia-working-image container-name -c
nvidia.runtime=true

lxc config device add container-name gpu gpu
```

Then, the launch is tested with the following command.

Table 6: Testing PoCL launch.

```
lxc exec container-name nvidia-smi
```

The following table includes the PoCL script.

Table 7: PoCL script details.

```
## Fix DNS problems

echo "nameserver 10.0.3.254" | sudo tee /etc/resolv.conf > /dev/null


## Set env variables

GIT_COMMIT=master

CLANG_VERSION=6.0

LLVM_VERSION=6.0

GH_SLUG=cpc/pocl-remote


## Get system necessities/dependencies
```

```
apt update

apt upgrade -y

apt install -y software-properties-common wget

wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -

add-apt-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-${LLVM_VERSION} main" && apt update

apt install -y build-essential ocl-icd-libopencl1 cmake git pkg-config libclang-${CLANG_VERSION}-dev clang-${CLANG_VERSION} llvm-${LLVM_VERSION}

apt install -y make ninja-build ocl-icd-libopencl1 ocl-icd-dev ocl-icd-opencl-dev libhwloc-dev zlib1g zlib1g-dev clinfo dialog apt-utils



## Get CUDA drivers and toolkit

cd /home

wget https://developer.download.nvidia.com/compute/cuda/11.1.1/local_installers/cuda_11.1.1_455.32.00_linux.run

sh cuda_11.1.1_455.32.00_linux.run --silent --drivers ## !!easily breakable, driver passthrough from host better option!!

sh cuda_11.1.1_455.32.00_linux.run --silent --toolkit

PATH=/usr/local/cuda-11.1/bin:${PATH}

LD_LIBRARY_PATH=/usr/local/cuda-11.1/lib64:${LD_LIBRARY_PATH}



## Clone and build

cd /home; git clone https://github.com/$GH_SLUG.git

cd /home/pocl; git checkout $GIT_COMMIT

mkdir build; cd build

cmake -G Ninja -DWITH_LLVM_CONFIG=/usr/bin/llvm-config-${LLVM_VERSION} -DCMAKE_INSTALL_PREFIX=/usr -DENABLE_CUDA=ON ..
```

```
ninja install


## Run useful tests and diagnostics

llc --version

clinfo

ctest -j4 --output-on-failure -L internal

cd /home/pocl; ./tools/scripts/run_cuda_tests
```

6.3 Demonstration

In this section, we present a demonstration of the containerized environment.

Initially, Slave1 accesses the underlying hardware GPU (and CPU).

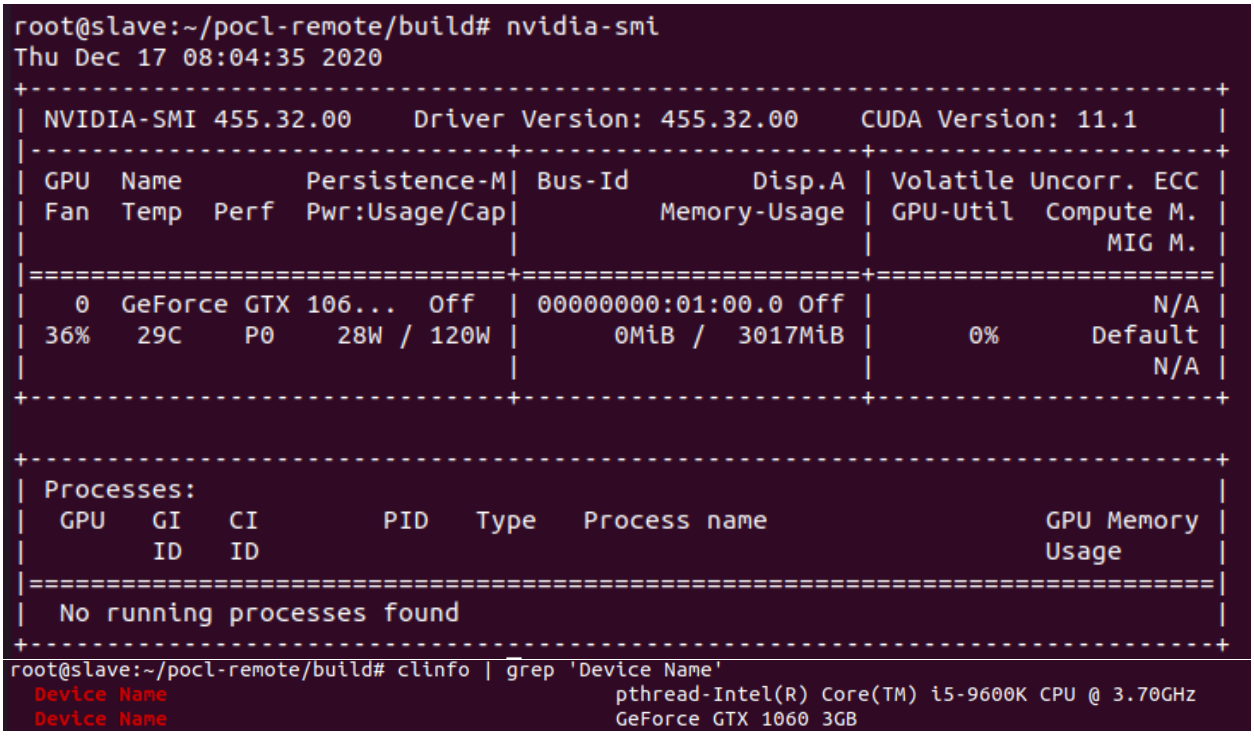

Figure 16: Slave1 accesses the underlying hardware GPU (and CPU).

Following, Slave 1 utilises CUDA to natively run GPU apps in the containerized environment.

```
root@slave:~/NVIDIA_CUDA-11.1_Samples/0_Simple/vectorAdd# ./vectorAdd
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

Figure 17: Slave 1 utilises CUDA to natively run GPU apps in the containerized environment.

It then uses PoCL-remote to wait for incoming traffic.

```
root@slave:~/pocl-remote/build# ./pocld
SERVER: PID 686 Listening on 10.140.146.96:10998, 10.140.146.96:10999 and 0.0.0.0:11000
SERVER: checking for zombies....
....DONE
```

Figure 18: Utilisation of PoCL-remote.

Controller1 then sets environmental variables to instruct PoCL-remote to search for slave node.

```
root@controller :~# export OCL_ICD_VENDORS=/root/pocl-remote/build/ocl-vendors/pocl-tests.icd
root@controller :~# export POCL_DEVICES=remote
root@controller :~# export POCL_REMOTE0_PARAMETERS=${HOST_IP}:10998/0
```

Figure 19: Controller1 then sets environmental variables to instruct PoCL-remote to search for slave node.

Then Controller1 can use the resources of the slave node.

```
root@controller :~# clinfo | grep 'Device Name'
  Device Name                                     pthread-Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
    Device Name                                   pthread-Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
    Device Name                                   pthread-Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
    Device Name                                   pthread-Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
root@controller :~# clinfo | grep 'Device Version'
  Device Version                                  OpenCL 1.2 pocl HSTR: pthread-x86_64-pc-linux-gnu-skylake HSTR: pocl-remote
```

Figure 20: Controller1 uses the resources of the slave node.

The remote GPU can be accessed like presented in the following figure.

```
root@controller :~# export POCL_REMOTE0_PARAMETERS=${HOST_IP}:10998/1
root@controller :~# clinfo
Number of platforms                               1
  Platform Name                                   Portable Computing Language
  Platform Vendor                                 The pocl project
  Platform Version                                OpenCL 1.2 pocl 1.5-pre/master-0-g7002e63e  Linux, Debug+Asserts, RE
  Platform Profile                                FULL_PROFILE
  Platform Extensions                             cl_khr_icd cl_pocl_buffer_size
  Platform Extensions function suffix             POCL

  Platform Name                                   Portable Computing Language
Number of devices                                 1
  Device Name                                     GeForce GTX 1060 3GB
  Device Vendor                                   NVIDIA Corporation
  Device Vendor ID                                0x10de
  Device Version                                  OpenCL 1.2 pocl HSTR: CUDA-sm_61 HSTR: pocl-remote: GeForce GTX 1060
```

Figure 21: Access of the remote GPU.

Finally, by running a couple of the PoCL-remote sample tests on the CPU, we get the following positive results, as presented in the figure below.

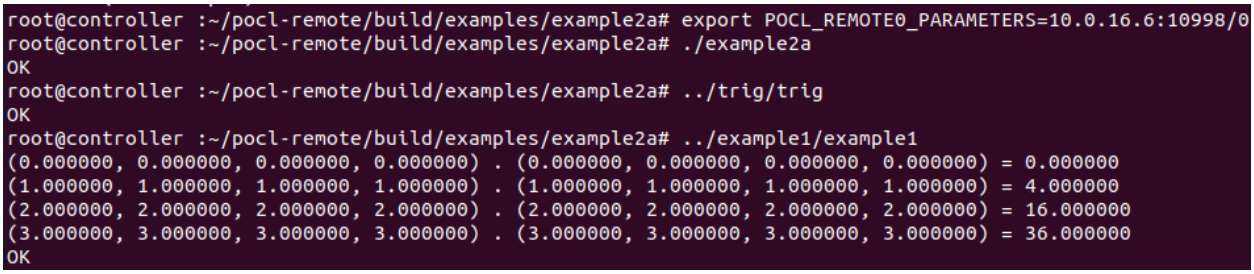

Figure 22: Positive results after running a couple of the PoCL-remote sample tests on the CPU.

These sample tests have been written with a CPU in mind, as such we run them on slave1's CPU (notice the export command before running the examples targets device 0, which is the CPU).

7 Dynamic Reconfiguration Support

In this section the dynamic reconfiguration support tools developed by UoP in the CPSoSAware project are described. Although we focus on the Driver Status Monitoring (DSM) application, the workflow can be generalized for similar use cases such as pose estimation, object detection, etc. Initially, the DSM application architecture is described briefly focusing on the computationally intensive parts that can be implemented in hardware. Then, we describe two ways to select alternative hardware kernels that implement the same function i.e., dynamically. Based on these two approaches, the dynamic reconfiguration mechanism for the DSM application is proposed.

7.1 Architecture of the DSM application

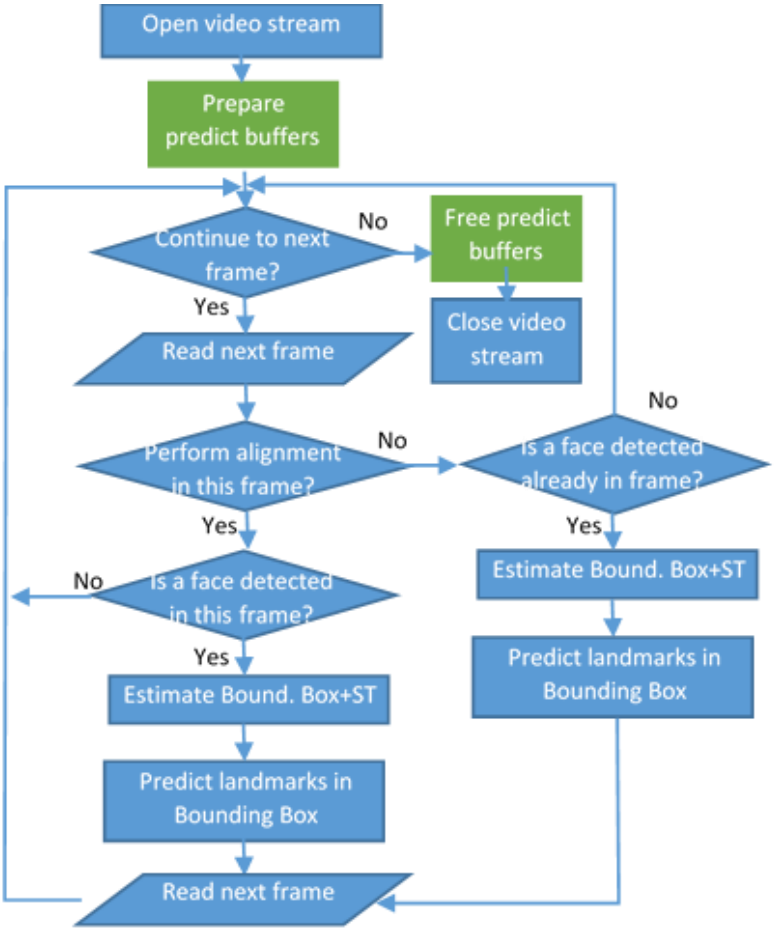

Figure 23: Flowchart describing the DSM video processing application.

The developed DSM application is based on the Deformable Shape Tracking (DEST) open package. The DEST package offers a number of applications related with face alignment: training using an annotated dataset, evaluation of a trained model, image and video processing. The implementation of the DEST package is based on Ensemble of Regression Trees (ERTs), a method proposed by Kazemi and Sullivan [ERT].

As shown in Figure 23, the DSM video tracking application opens an input video stream (a camera or a stored video) and for each frame that is analysed, the OpenCV library face recognition utility is used to detect the face bounding box. Then, the ERT algorithm has a number of cascade stages and uses a number of regression trees where the mean shape landmark positions are compared with reference pixels from a sparse representation of the image in order to select appropriate correction factors for the position of the landmarks. A shape warping is also performed using a routine called Similarity Transform (ST) as shown in 3.

Profiling showed that the process of "predicting" the landmark position within the bounding box is the most time-consuming operation and for this reason it was selected for hardware implementation as described in detail in the context of Task 4.1.

Using the DEST package utilities different models have been implemented based on different ERT parameters (number of cascade stages, regression trees, reference pixels, depth of the trees, etc). These different models are implemented with different landmark prediction hardware kernels that show different features (accuracy, speed, power consumption, resources). According to environmental conditions (male or female driver, night time or daytime) the system should be able to dynamically reconfigure switching between different predict kernels. A number of these alternative hardware kernel implementations are described in Task 5.1.

7.2 Dynamic hardware kernel loading in Xilinx Vitis XRT

The Xilinx Real Time library (XRT) offers several options concerning the dynamic reconfiguration of an FPGA. Two of these options have been investigated using alternative implementations of the DSM video tracking application. We developed two projects where we assume that we need to switch between different hardware kernels i.e., the FPGA has to be dynamically reconfigured by the software in order to use alternative implementations of the same function based on a decision taken by the top level software. Although in this sense both of these hardware kernels have to take the same arguments, this is not necessary. To understand how alternative hardware kernels can be dynamically loaded, the preparation steps of this procedure have to be considered (see Figure 23).

Initially the device and platform have to be detected and a context has to be created based on the detected devices. Then a command queue is created where the kernel arguments and the kernel task itself will be added. After creating the command queue, the external bitstream file that will configure the programmable logic of the FPGA has to be defined. These are the so called xclbin files in Xilinx environment. These files may contain multiple kernels as we see in one of the two alternative implementations examined here. The top level software is not obliged to use all of the kernels defined in a xclbin file. The kernel that will be called by the software is defined next.

Table 8: The steps describing how a hardware kernel is loaded in the FPGA using Xilinx XRT.

1	Detect device, platform and create Command Queue for the context of this device
2	Define and load the xclbin file with the bitstream of the hardware kernels

3	Create a Program from the device, context and the bitstream
4	Define the kernel name in the Program
5	Prepare the memory buffers with the kernel arguments
6	Enqueue and set the kernel arguments
7	Start the kernel (enqueue the kernel task)
8	Prepare the buffer for the return values of the kernel
9	Wait for the kernel to finish (if InOrder execution is used)

Then, the input arguments of the kernel have to be prepared along with the necessary memory buffers where they are stored and pointers to these buffers have to be added to the command queue. The kernel starts by performing a kernel task add in the command queue. The system can wait for the kernel to finish and store the results in other buffers that have also been prepared (in order execution). However, it is possible to perform out of order execution by adding to the command queue multiple kernel tasks before waiting each one of them to finish. All of the steps described above are implemented in OpenCL.

The two alternative dynamic reconfiguration scenarios tested are described in the following subsections.

7.3 Multiple kernels in the same xclbin

Two kernels with the same arguments have been developed and declared to the Xilinx Vitis system project. The project was built and one xclbin file was generated. Then in real time operation the software loads the single xclbin file in step 2 of Table I and then it selects the kernel that it will call in step 4. In our case as said earlier both kernels use the same arguments. However kernels with totally different arguments can be selected and in this case the steps 5 and 6 have to be adapted accordingly.

If for example, the developed two kernels (A and B) have to be called as follows: A in the “if-then” part of a conditional statement in the top level software and B in the “else” part then steps 1-3 could be executed during initialization and steps 4-9 can be repeated in the “if-then” and the “else” part of a conditional statement in the top level software. In this case the name and the arguments of each kernel A or B have to be adapted accordingly.

7.4 Multiple xclbin files with different kernels

In this case two different kernels can be built with dfferent Xilinx Vitis system projects generating two different xclbin files with discrete names. Then in real time operation the software loads the appropriate

xclbin file in step 2 of Table I and then it selects the kernel that resides within this xclbin file. If, for example, different kernels has to be selected in the “if-then” part of a conditional statement in the top level software and the “else” part then all the steps from 2 to 9 have to be repeated in the “if-then” and the “else” part of a conditional statement in the top level software. In this case the name and the arguments of each kernel have to be adapted accordingly. In this case only the 1st step can be executed during initialization.

7.5 Dynamic reconfiguration in the DSM application

The proposed dynamic hardware kernel reconfiguration scheme for the DSM application is described in Figure 24. It is based mainly on the multiple xclbin files with the multiple different kernels approach described in Section 7.2 which is more general and allows the build of independent kernels and xclbin files by different, distant agencies. In Figure 24 the yellow boxes display the steps listed in Table 8. These step numbers are included in the parentheses in these boxes.

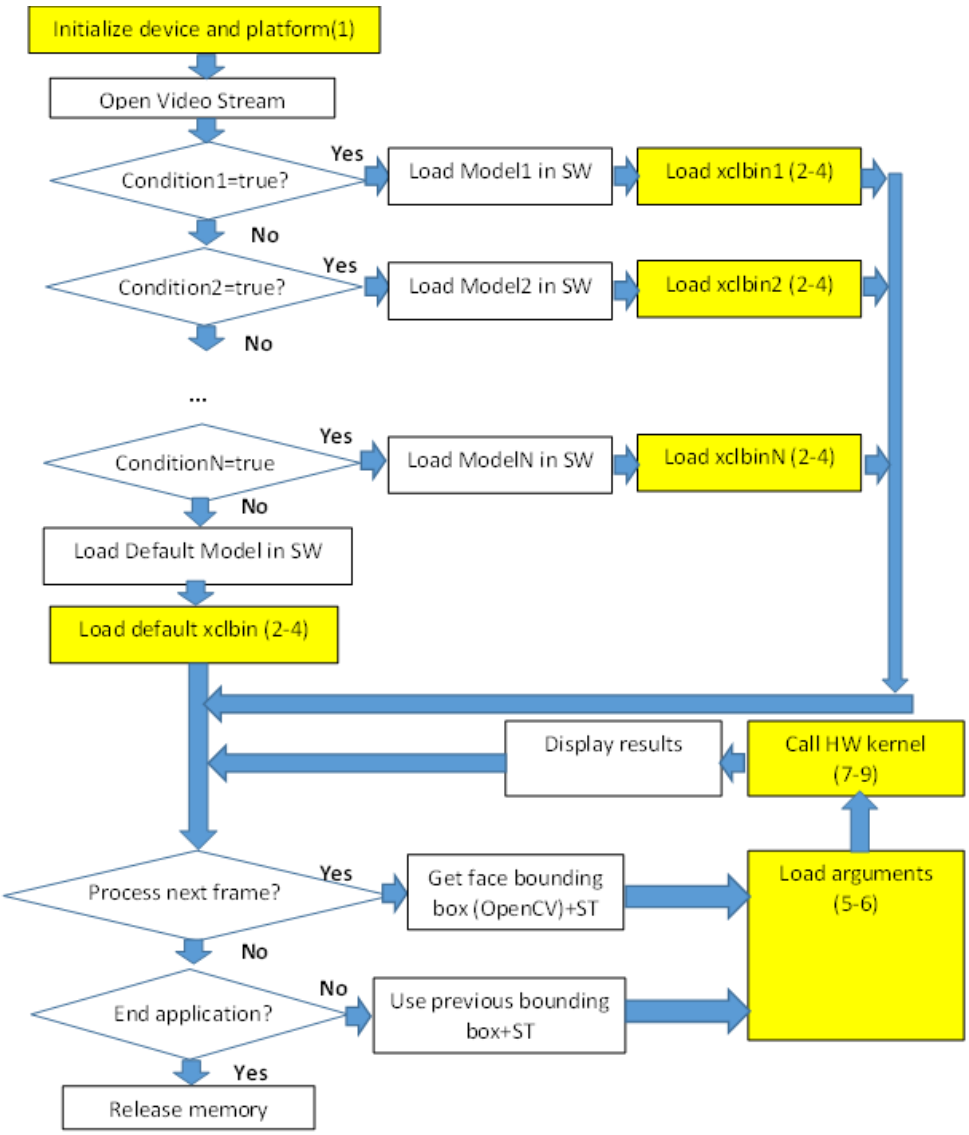

Figure 24: The proposed dynamic reconfiguration scheme for DSM.

The DSM video tracking application starts by initializing in OpenCL the underlying hardware device and platform. The input video stream is also opened during initialization. Then external conditions determine which pair of model and hardware kernel will be loaded. Conditions 1, 2,…, N can be sensor values (e.g., the recognize if it is dark outside the vehicle) or other inputs (e.g. recognition of the driver gender). According to the condition that is true the appropriate trained DEST model and the corresponding hardware kernel implementation is loaded. The DEST model is loaded in software. The hardware kernelxclbin is loaded by OpenCL. Then the loop starts as presented in Figure 24. If a frame has to be analyzed (some frames may be omitted to avoid the time consuming OpenCV face bounding box recognition), the bounding box recognized by OpenCV is used, ST is performed and the hardware kernel arguments are prepared. The hardware kernel implementing the landmark prediction operation is called. The results returned by the hardware kernel are used to display the results or are returned to higher level

software layers that can infer about the drowsiness of the driver from the distance between specific landmarks.

The pair (model, hardware kernel) is selected once as shown in Figure 24. However, the dynamic reconfiguration means that the system should be able to switch between pairs of (model, hardware kernel) if the external environmental conditions change. This can be implemented by periodically returning to the 1st condition check after displaying the results, instead of the “Processing next frame?” condition. Loading a new pair of (model, xclbin file) is a process that requires several seconds. This is the reason why we do not always return to “Condition1=true?” check point.

More details on the dynamic reconfiguration will be given in the context of WP5.

8 Networking Platform Reconfiguration

To support the required data transfers and to make possible the execution of kernels in a distributed manner, a network infrastructure enabling it should be available. Following the general architecture depicted by the project, the network platform should span to reach all the levels, from the cloud system layer down to the lowest layer, which is the cps level, in which the kernels would be executed. That networking platform should be flexible, scalable, and reconfigurable.

In order to achieve that level of flexibility, the project proposes to have at the cloud system layer an SDN controller based in ONOS. ONOS is the leading Open-Source software to build NFV and SDN solutions. In this case, ONOS will act as a controller for the different virtual switches hosted in the edge servers at the edge layer. The controller will be assisted by the optimizer which will reconfigure the network according to the temporary needs of the different phases (normal operation, cyber thread detected, mitigation phase, etc). Also, in the cloud system layer the different applications are hosted, as for example the application tasked with managing the commissioning/decommissioning signalling.

Note, that for the rest of the chapter we focus more on the scenario of the Autonomous Vehicle, but similar setup could be configured to be used in other scenarios, by simply adapting the services running at the edge.

At the edge layer, in the different edge servers, we can find one or more virtual switches, depending on the scale of the network and the requirements of the services running at the edge. These virtual switches are implemented with Open vSwitch which is a production level, open-source multilayer virtual switch. Open vSwitch is designed to enable network automation using programmatic extensions or applications. In the edge servers there are one or more hosts connected to the switch. In these hosts, services will be instantiated to enable the edge computing capabilities, required by the different scenarios. As examples of these services, we can find in the edge server some of the following services: an mqtt broker service belonging to the v2x stack, or the full SRMM service (a cybersecurity analysis tool). Connected to the edge server, we can find cheap and small of-the-shelf switches that will allow the connexion of physical devices belonging to the RSU (Roadside Unit) layer to the edge layer.

Lastly from the RSU layer, there should be wireless communication from the RSU layer to the CPS layer, in the case of the autonomous car scenario, one of the CPS on boarded in the car, or OBU (On Board Unit). These OBU are the ones in charge of sending and receiving for example CAM or DEM messages or notify from the SRMM agent that there is a cybersecurity incident.

All the previously depicted infrastructure, will be described in more detail in D4.7 as a part of the documentation produced for the interconnectivity layer which is used in the demonstrator implemented in the context of Task 4.2 and Task 6.2.

8.1 Intra Communication Platform Reconfiguration

In the context of T3.2, UoP designed and developed a dynamic and scalable mechanism for the management of two critical functionalities of the intra – communication layer:

1. the deployment / commissioning: This component is responsible to deliver to the target system the configurations of intra – communication wireless interfaces.

2. the execution mechanism: This component is responsible to handle RX/TX of data over the available intra – communication wireless interfaces.

In both functionalities, an event – driven approach has been followed. In that context, the operation of the mechanisms is based on trigger events that are published to message queues. The main modules that are used in this environment are:

- An MQTT broker configured to handle the message queues of the system.
- The broadcast module that is responsible for the tx operations
- The listener module that is responsible for the rx operation
- On the system layer, the simulation environment issues the network configurations commands based on the executed simulations for each of the application scenarios.

This implementation is implemented and demonstrated on a single board computer (raspberry pi) utilizing the linux drivers for wifi and bluetooth (bluez). The implementation can be ported to any device that can run a Linux OS.

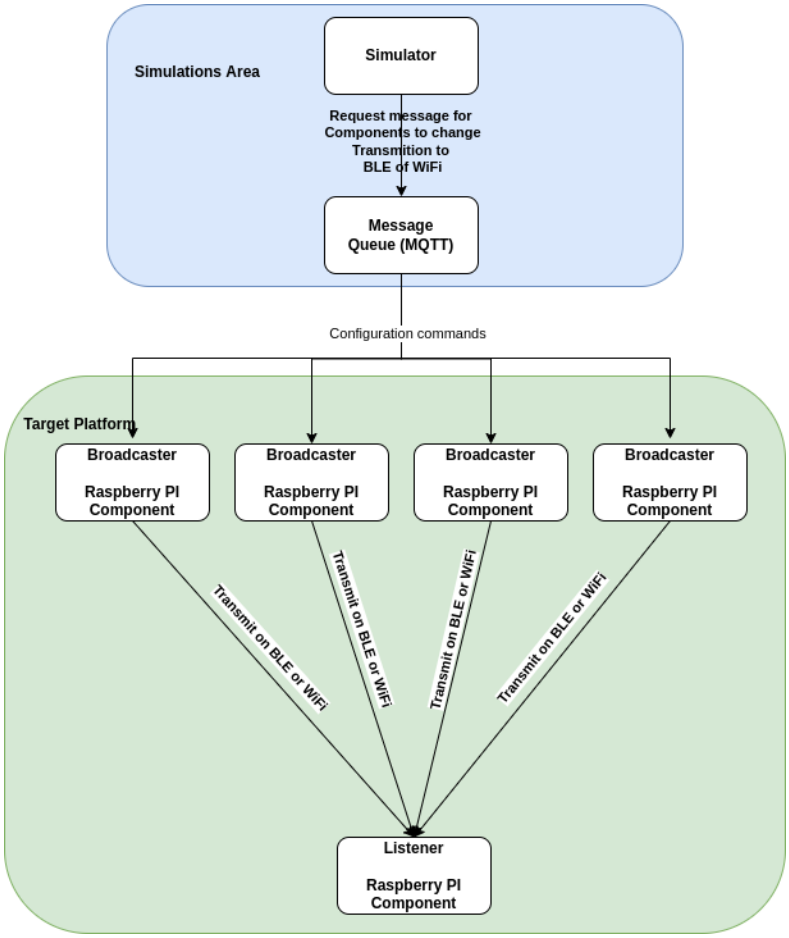

Figure 25: Environment Diagram.

8.1.1 Deployment/Commissioning Mechanism

In principle, after the completion of each simulation and the respective result evaluation, the system decides the communication technology (BLE/WiFi) to be used based on the application requirements. Sequentially, a message is emitted (e.g. RunOnBle, RunOnHttp) to a message queue, to which the broadcast components are listening to. This process is repeated every time the simulation results are processed and evaluated with respect to the application requirements. This flow can be discriminated in 2 pillars. The first pillar selects the transmission channel of the data at the application layer. This channel can be either the TCP/IP network or an emulation of a serial communication over BLE. This option allows as to handle back-pressure conditions where data are generated but cannot be served due to network configuration switching. At the second pillar a low level access to the available wireless interfaces is given. Especially for the WiFi option where more options are available, the driver of the interface is used to update several configuration options. An example of a first set of WiFi and TCP parameters that can be configured through the implemented mechanism is given below.

rtsHandshaking	Adds a handshake before each packet transmission to make sure the channel is clear
packetFragmentationSize	Sets the maximum fragment size, which is always lower than the maximum packet size
retryLevel	Set the maximum number of times the MAC can retry transmission.
txPowerLevel	Sets the transmit power in dBm
socketReceiveBuffer	Data buffered on the recipient's end
socketSendBuffer	Data buffered on the transmitter end
socketReceiveBufferMax	Max data buffered on the recipient's end
socketSendBufferMax	Max data buffered on the transmitter end
windowScaling	Option to increase the receive window size allowed in Transmission Control Protocol above its former maximum value

Moreover, the option for completely enabling/disabling the wireless interfaces is given. This option can be beneficiary for cases where an optimization on the power consumption is required based on the application requirements and the hardware resources.

8.1.2 Broadcast module

The broadcaster component is loaded and executed as a python service. This service is connected on the message queue and it is consuming the messages from the queue which contains the interface of preference to be used (BLE or WiFi). When the service is starting, it is in status on-Hold, it enables the WiFi and BLE interfaces and then it creates two transmission subprocesses. Both subprocesses are set on hold for both interfaces, and no data are transmitted. Upon a configuration message is received, it “un-holds” the related subprocess and transmission is started. Upon a new configuration message reception, the service changes the transmission interface or updates the current configuration. Sequentially, a handover to the other interface is performed by unfreezing its subprocess and switching the state.

8.1.3 Listener Module

The listener operates on the other end of 2-peer communication. This module is again a python service which runs an http server and a BLE dump service. The http server is exposing a rest api which receives the data from the broadcaster(s) when the transmission is through WiFi. On the other hand, the BLE dump service creates a white list of broadcasters and it is listening data only from them when they are transmitting from this interface. The same configuration options for the wireless interfaces can be tuned as in the broadcaster.

9 Conclusions

This report delivered the output of Task 3.2: *"Design and Develop CPS Layer CPSoSaware Deployment/Commissioning and Execution Mechanism."*. Associated with the deliverable is a *demonstrator* video available here.

In addition to describing the demonstrator that showcases the OpenCL-based CPS sofware stack using a hardware-in-the-loop example, it gave technical details on other the key relevant aspects of the mechanism such as the OpenCL-based distributed execution environment itself, accelerator utilization layer using the built-in kernel feature of the specifications, distributed use of FPGA, deployment of software stack (containerization), as well as reconfiguration of the FPGA fabric and the network.

We believe the goals for this task were met in every way, thanks part to solid background work on PoCL-R that was brought to the project which allowed us the deliver the required new features on top of it within the allocated researcher resources. Some of the features such as the dynamic reconfiguration of the FPGA utilizing a firmware library and the simulation support will be finished with work packages 4-6.

References

[CARLA] Alexey Dosovitskiy and German Ros and Felipe Codevilla and Antonio Lopez and Vladlen Koltun, “CARLA: An Open Urban Driving Simulator”, Proceedings of the 1st Annual Conference on Robot Learning, 2017.

[DOCKER] Docker Engine Overview. https://docs.docker.com/engine/, accessed: 2022-03-23

[ERT] V. Kazemi and J. Sullivan, "One millisecond face alignment with an ensemble of regression trees," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 2014, pp. 1867-1874, doi: 10.1109/CVPR.2014.241

[LXR] Linux containers. https://linuxcontainers.org/, accessed: 2022-03-23

[MSKU20] M. Moravcik, P. Segec, M. Kontsek, J. Uramova and J. Papan, "Comparison of LXC and Docker Technologies," 2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA), 2020, pp. 481-486, doi: 10.1109/ICETA51985.2020.9379212

[OpenCL] Khronos® OpenCL Working Group, “The OpenCL™ Specification,” https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, accessed: 2022-03-22

[PoCL] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, “pocl: A performance-portable OpenCL implementation,” Int. Journal of Parallel Programming, vol. 43, no. 5, 2015

[PoCL-R] J. Solanti, M. Babej, J. Ikkala, V. Kumar Malamal Vadakital, and P. Jääskeläinen, “PoCL-R: A scalable low latency distributed OpenCL runtime,” in Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation XXI, 2021. [Online]. Available: https://samos-conference.com/wp/wp-content/uploads/2021/07/R_28_PDF.pdf

[SLDV21] M. Sollfrank, F. Loch, S. Denteneer and B. Vogel-Heuser, "Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation," in IEEE Transactions on Industrial Informatics, vol. 17, no. 5, pp. 3566-3576, May 2021, doi: 10.1109/TII.2020.3022843

[SPIR-V] Khronos® Group, “The SPIR-V™ Specification,” https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.pdf, accessed: 2022-03-22

[SYCL] Khronos® SYCL™ Working Group, “SYCL™ Specification,” https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, accessed: 2022-03-23.

[Vulkan] Khronos® Vulkan™ Working Group, “Vulkan™ Specification,” https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html, accessed: 2022-03-23

[WRKP19] J. Watada, A. Roy, R. Kadikar, H. Pham and B. Xu, "Emerging Trends, Techniques and Open Issues of Containerization: A Review," in IEEE Access, vol. 7, pp. 152443-152472, 2019, doi: 10.1109/ACCESS.2019.2945930