Ref. Ares(2021)4077472 - 23/06/2021

D2.3 MODELS OF HW & SW COMPONENTS WITH NON-FUNCTIONAL REQUIREMENTS AND SECURITY EXTENSIONS (DEMONSTRATOR REPORT)

Authors	Pekka Jääskeläinen (TAU), Topi Leppänen (TAU), University of Peloponnese, Francesco Regazzoni (USI), Apostolos Fournaris (ISI), Aris Lalos (ISI), , Pavlos Kosmides (CTL)
Work Package	WP2 Virtual User/Physical Environment Models, CPS Models and orchestration support tools

Abstract

This deliverable is a *demonstrator report* that describes the output of Task 2.3 "CPS Models for HW & SW components" and Task 2.4 "Modeling of non-Functional Requirements and Security Functionality". The demonstrators of the D2.3 include 1) a running prototype of the AlmaIF v2, a HW component wrapper developed in the project for connecting HW components to the PoCL framework, 2) MATLAB based automated KPI modelling demo, 3) LLVM-based instruction mix analyzer, that is demonstrated on an OpenCL benchmark suite. In addition to the demos, this report provides model descriptions of the SW and HW (FPGA) components developed by the CPSoSAware partners for use in the pilots. The second part provides a methodology to extend the previous models (meta-models) with specific KPIs related to non-functional requirements including specific annotations for security extensions, a result from Task 2.4. It also describes the new version of the AlmaIF done in the project as well as the MATLAB-based automated KPI modelling approach.

Funded by the Horizon 2020 Framework Programme
of the European Union

Deliverable Information

Work Package	WP2 Virtual User/Physical Environment Models, CPS Models and orchestration support tools
Tasks	T2.3 [M1-M18] CPS Models for HW & SW components and T2.4 [M1-M18] Modeling of non-Functional Requirements and Security Functionality
Deliverable title	D2.3 MODELS OF HW & SW COMPONENTS WITH NON-FUNCTIONAL REQUIREMENTS AND SECURITY EXTENSIONS
Dissemination Level	PU
Status	F: Final
Version Number	1.1
Due date	30/06/2021

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	AUTHORS
1.1	17/06/2021	Addressed remarks and fixed from internal reviewers.	Pekka Jääskeläinen, Georgios Keramidas, Apostolos Fournaris, Nikos Piperigkos

	NAME
Prepared by	Pekka Jääskeläinen (TAU)
Reviewed by	Nikos Piperigkos (ISI), Apostolos Fournaris (ISI), University of Peloponnese
Authorised by	ISI

DATE	RECIPIENT
17/06/2021	Project Consortium
XX/XX/2021	European Commission

Table of contents

Executive summary 13

1. Introduction 14

    1.1 Document structure 14

    1.2 Relation to the Project 14

    1.3 Acronyms and descriptions 15

2. CPSoSAware HW and SW Component Model Properties 16

    2.1 Vector Add 16

        2.1.1 Metadata 17

        2.1.2 Measurement data 17

    2.2 Matrix Transpose 17

        2.2.1 Metadata 17

        2.2.2 Measurement data 17

    2.3 Matrix Multiplication 18

        2.3.1 Metadata 18

        2.3.2 Measurement data 18

    2.4 Elliptic Curve Operation Arithmetic Unit 18

        2.4.1 Metadata 19

        2.4.2 Measurement data 19

    2.5 Number Theoretic Transform (NTT) 19

        2.5.1 Metadata 19

        2.5.2 Measurement data 19

    2.6 Inverse Number Theoretic Transform (NTT) 20

        2.6.1 Metadata 20

        2.6.2 Measurement data 20

2.7 Pointwise Montgomery Multiplication 20
2.7.1 Metadata 20
2.7.2 Measurement data 21
2.8 2D Object Detection and Tracking Using Neural Networks (NN) 21
2.8.1 Metadata 21
2.8.2 Measurement data 21
2.9 Conv2D+ReLU+maxpool 22
2.9.1 Metadata 22
2.9.2 Measurement data 22
2.10 Face Detection 22
2.10.1 Metadata 22
2.10.2 Measurement data 23
2.11 Pose Detection 23
2.11.1 Metadata 23
2.11.2 Measurement data 23
2.12 Heart Rate Monitoring 23
2.12.1 Metadata 23
2.12.2 Measurement data 24
3. Modeling Extra Functional CPSoSAware KPIs 25
4. AlmaIF v2: Common Hardware Component Interface 28
4.1 AlmaIF v2 Changes in Comparison to AlmaIF v1 28
4.1.1Accelerator Memory Area Packing 28
4.1.2AXI Master Support 29
4.1.364b Addressing 29
4.1.4Example Command Queue Processor 29

4.1.5 Other Updates 29

4.2 Memory Addressed Registers 30

4.3 Software Emulation / C Hardware Model 34

4.4 AlmaIF v2 Demonstrator 34

4.5 AlmaIF-accelerator Implemented with High-Level Synthesis Tools 35

4.6 Future Work: AlmaIF v3+ Ideas 41

5. LLVM-Based OpenCL Kernel Instruction-Mix Characterizer 42

5.1 The LLVM IR Instruction Mix Profiler 42

5.2 Profiler Design 43

5.3 Demonstrator: Example Runs of the Profiler 44

5.4 Results 47

5.5.1 Approach 47

5.5.2 Kernel Characterization 48

6. Component Matlab Modeling Demonstrator 50

7. Conclusions 53

References 54

List of tables

Table 1: Kernel classifications. 48

List of figures

Figure 1: A comparison between AlmaIF V1 and AlmaiF V2 memory maps using an example accelerator with no need for configuration memory. 30

Figure 2: AlmaIF v2 with an example software programmable OpenASIP core that also handles command queue processing. 33

Figure 3: Generic HW accelerator with AlmaIF v2 where there is a command queue processing handling the control flow and launching Accelerator functionality implemented manually in RTL or generated from HLS. 34

Figure 4: Two AlmaIF accelerators placed in Vivado block design for Zynq-7020 FPGA. On the right are the physical addresses used for the driver to find the accelerator devices. 35

Figure 5: A screenshot from the AlmaIF demonstration with two accelerator devices and a computation with shared data between them. 35

Figure 6: The HLS implementation of the AddTwo accelerator. 36

Figure 7: The interface of the AddTwo accelerator. 37

Figure 8: The Vivado project of PoCL platform supporting FPGA accelerators. 37

Figure 9: The physical address of the BRAM controller module. 38

Figure 10: Export of the Vivado project to be used by the PetaLinux tools. 38

Figure 11: Successful build of the PetaLinux project. 39

Figure 12: Zedboard running PetaLinux. 40

Figure 13: Testing of the Pocl accelerator prototype in Linux running on the FPGA SoC platform. 41

Figure 14: Profiler overview. 43

Figure 15: Profiler block diagram. 44

Figure 16: Source code of doitgen kernel 2. 45

Figure 17: LLVM IR code of doitgen kernel 2. 46

Figure 18: Output of the profiler. 47

Figure 19: LLVM IR instructions classification. 49

Figure 20: MATLAB for Collision Risk KPI modeling. 51

Figure 21: MATLAB for Data Integrity Component KPI modeling. 51

Figure 22: MATLAB for Thread Severity KPI modeling. 52

Figure 23: MATLAB for Repeatability of algorithm speed KPI modeling. 52

Executive summary

The output of Task 2.3 of CPSoSAware are four practical demonstrators of the modelling and component wrapping work done in Work Package 2. This report accompanies the demonstrators, describing the demos and providing further technical information. The demonstrators of the D2.3 include 1) an FPGA prototype of the AlmaIF v2, a HW component wrapper developed in the project for connecting HW components to the PoCL framework demonstrated with two type of co-processor designs (OpenASIP and a Vitis HLS generated one), 2) MATLAB based demonstrators of automated KPI modelling, and 3) LLVM-based instruction mix analyzer, that is demonstrated on an OpenCL benchmark suite.

This is a written part of the deliverable that accompanies the demonstrators by providing the model metadata for the key components used in the project, as examples of the model metadata extracted. It also describes the new AlmaIF version which is used for interfacing to the hardware blocks from the OpenCL based software stack. In addition to the modelling aspects of T2.3, it includes descriptions of the security extensions as identified in T2.4. Finally, the LLVM-based profiling tool's implementation is described.

1. Introduction

This deliverable is the written part of the activities performed in Task 2.3 and Task 2.4. The main part of the D2.3 are the demonstrators which can be reproduced through access to source code repositories and pre-recorded videos.

A key target in Task 2.3 was to create a hardware component interface which can be used to efficiently and easily wrap accelerators such that they can be communicated with a common driver interface as well as perform peer-to-peer communication. This interface, called AlmaIF v2, was defined within the WP2 of this project, based on initial work in the previous project, and will be further developed and adopted in WP3 and WP5 of CPSoSAware. Another key aspect was to create tooling for capturing static profiles of kernel operation usage, which was implemented as a new LLVM compiler pass. Finally, the component meta data modelling data collection was performed within the project with the key components along with their metadata listed in this project.

As an outcome of Task 2.4, the deliverable provides model metadata for selected key components used in the project pilots as examples of model metadata extracted. It also includes descriptions of the security extensions as identified in T2.4.

1.1 Document structure

This document is structured into the following major sections:

- Section 2 provides meta data and measurement data for a list of components that are either already used in the pilot demonstrators, or are likely going to be.
- Section 3 describes the security extensions in the collect component meta data.
- Section 4 defines the extensions done for the AlmaIF as well as how the new features were demonstrated.
- Section 5 outlines the LLVM-Based OpenCL Kernel Instruction-Mix Characterizer.
- Section 6 overview the Matlab-based component modeling demo with examples of how KPIs were included in the Matlab model.
- Section 7 concludes the report. CPSoSAware HW and SW Component Model Properties

1.2 Relation to the Project

This deliverable reports the outcomes of Task 2.3 and Task 2.4 of the WP2. It provides input to WP3 Task 3.6 for the hardware component library by introducing an updated hardware wrapper abstracting the component details so they can be driven by a common OpenCL driver in order to efficiently integrate FPGA-based HW components at platform level. The OpenCL kernel profiler is utilized to provide feedback for producing the OpenCL-supporting soft cores in that task. Task 3.2 similarly utilizes the demonstrated component wrapper when integrating FPGA-based components to the distributed OpenCL platform. The modeling and meta data aspects feed to WP4 where the parameters are taken in account in the simulation runs of Task 4.4 and partitioning decisions of Task 4.1. It also feeds to Task 2.5, which provides the orchestration tool in close combination with the simulation tooling provided in WP4. The Vitis HLS generated proof-of-concept provides basis for Task 5.1.

1.3 Acronyms and descriptions

Below are listed the most relevant abbreviations used in the document and their expanded meanings:

Acronym / Term	Description
AlmaIF	ALMARVI Common Hardware IP Interface
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AQL	Architected Queuing Language
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CPS	Cyber-Physical System
CPU	Central Processing Unit
CPSoS	Cyber-Physical System of Systems
CQ	Command Queue
CNN	Convolutional Neural Networks
DSP	Digital Signal Processor / Digital Signal Processing
EC	Elliptic Curves
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High Level Synthesis
HSA	Heterogeneous Systems Architecture
HW	Hardware
IP	Intellectual Property
IR	Internal Representation
KPI	Key Performance Indicator
LLVM	Just LLVM (used to mean Low Level Virtual Machine).
LSU	Load-Store Unit
MEC	Multi-access Edge Computing
MMU	Memory Management Unit
NN	Neural Network
NTT	Number Theoretic Transform
OpenASIP	Open Application-Specific Instruction-set Processor
OpenCL	Open Computing Language
PoCL	Portable Computing Language
ReLU	Rectified Linear Unit
SFU	Special Function(al) Unit
SVM	Shared Virtual Memory
SoC	System-on-a-Chip
VHDL	VHSIC Hardware Description Language

2. CPSoSAware HW and SW Component Model Properties

In the CPSoSAware project, the component model properties were outlined as a set of interesting meta data and quantitative information which were considered adequate for serving the needs in other work packages (particularly *Tasks 3.6, 4.1, 5.1 and 5.2*) as well as other purposes such as developing alternative algorithms for the pilot demonstrators or new performance/cost estimation models. It also serves as a basis for the component library provided in D3.6 for the reliable components. This section describes the HW (FPGA and/or ASIC targeted realization) and SW (CPU and GPU based) components along with their model parameters developed by the middle of the project (M18).

The following meta data was collected for the components:

- **Semantics:** A description of the implemented algorithm's functionality.
- **Linear execution time:** This indicates if the algorithm implementation has linear execution time according to the size of the inputs. This helps estimating the execution time of different size inputs based on a priori execution times with known size inputs.
- **Description language:** The input language used to define the implementation of the component. It can be a software language for programmable device targeted implementations or an hardware description language for fixed function implementations on FPGA devices.
- **Security functionality:** If the component implements security functionality.
- **Reliable component:** If the component implements features for reliability.

Many of the algorithm implementations were already ported, optimized (to some degree) and measured in one or more devices of interest to provide indicative data of performance and performance portability. Thereby, we collected the following data:

- **Target device:** The device used.
- **Device type:** Device type or "class" typically recognized in computer engineer's parlance (CPU, GPU, DSP or FPGA,).
- **Clock frequency:** The clock frequency used by the device in this measurement.
- **Input size:** Input size used for the measurement.
- **Runtime:** The execution time of the algorithm for the given input on the given device.
- **Power**: The average power consumption when executing the given task.

The following subsections highlight the key components used in the project's tasks and pilot implementations. Some of the components implement lower level, but more general-purpose functionality such as vector additions, matrix transposes or matrix multiplications. These can be used to hierarchically implement more complex functionality or algorithms such as the Kalman filter or Neural Network based 2D Object Detection, which further can be utilized for top level functionality such as Blinking Detection.

2.1 Vector Add

A generic component used, for example, in the Kalman filter developed in the Task 3.1. Meant for easy adoption to various OpenCL supported devices.

2.1.1 Metadata

- **Semantics:** Performs element-wise addition of two double precision floating point vectors and produces a similar size output of the sum.
- **Linear execution time:** Yes.
- **Description language:** OpenCL C++ (Python bindings - pyopencl).
- **Security functionality:** None.
- **Reliable component:** No.

2.1.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
Intel x64 i7-7700	CPU	3600	14 + 14 double	0,01
Intel x64 i5-650	CPU	3200	14 + 14 double	1,20
AMD Radeon HD 5570	GPU	650	14 + 14 double	3,78
NVIDIA GeForce GTX 1050Ti	GPU	1291	14 + 14 double	1,06
Intel x64 HD 630	GPU	350	14 + 14 double	0,3

2.2 Matrix Transpose

A generic component used, for example, in the Kalman filter developed in the *Task 3.1*. Meant for easy adoption to various OpenCL supported devices.

2.2.1 Metadata

- **Semantics:** Performs a matrix transpose. The kernel has an input of an array of NxM size with double precision floating points values and returns an array of MxN size with double precision floating points values.
- **Linear execution time:** Yes.
- **Description language:** OpenCL C++ (Python bindings - pyopencl).
- **Security functionality:** None.
- **Reliable component:** No.

2.2.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
Intel x64 i7-7700	CPU	3600	16x14 double	0,02

Intel x64 i5-650	CPU	3200	16x14 double	1,60
AMD Radeon HD 5570	GPU	650	16x14 double	4,67
NVIDIA GeForce GTX 1050Ti	GPU	1291	16x14 double	1,06
Intel x64 HD 630	GPU	350	16x14 double	0,20

2.3 Matrix Multiplication

A generic component used, for example, in the Kalman filter developed in the Task 3.1. Meant for easy adoption to various OpenCL supported devices.

2.3.1 Metadata

- **Semantics:** Performs a matrix multiplication of matrices with double precision floating point arithmetics. The kernel has an input of two matrices of NxM and MxK sizes and returns an array of NxK size.
- **Linear execution time:** Yes.
- **Description language:** OpenCL C++ (Python bindings - pyopencl).
- **Security functionality:** None.
- **Reliable component:** No.

2.3.2 Measurement data

Target device	**Device type**	**Clock frequency (MHz)**	**Input size**	**Runtime (us)**
Intel x64 i7-7700	CPU	3600	14x14 X 14x14 double	0,01
Intel x64 i5-650	CPU	3600	14x14 X 14x14 double	1,40
AMD Radeon HD 5570	GPU	650	14x14 X 14x14 double	4,78
NVIDIA GeForce GTX 1050Ti	GPU	1291	14x14 X 14x14 double	1,02
Intel x64 HD 630	GPU	350	14x14 X 14x14 double	0,10

2.4 Elliptic Curve Operation Arithmetic Unit

The Component performs point operations on Elliptic Curves (EC) defined on Binary extension Fields (GF(2k) where k is at most 233 bits. The arithmetic unit performs EC point addition as well as EC point doubling operations that are needed in all EC based public key cryptography schemes (using in security protocols like TLS/SSL, SSH, SMiME etc.). The Component's functionality can be extended to scalar multiplication which is the main operation of EC cryptography and used as part of T3.5.

2.4.1 Metadata

- **Semantics:** The component takes 3 EC point inputs of at most 233 bits (padded to 256 bit numbers) and produces one 233 bit result. Each input consists of two 233 bit numbers corresponding to polynomials defined on GF(2^k). There is software model implementation and also a hardware implementation modelled using VHDL
- **Linear execution time:** Yes.
- **Description language:** C (software implementation) and VHDL (hardware implementation)
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.4.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
ZCU104 –Arm Cortex A53 (Zynq Ultrascale+)	CPU	1.5 GHz	3x32 uint32	148250
ZCU104 –FPGA PL (Zynq Ultrascale+)	FPGA	100 MHz(bus clock)	3x32 uint32	1200 (at average)

2.5 Number Theoretic Transform (NTT)

A generic component used, for example, in the Dilithium Digital Signature developed in the *Task 3.5*. Meant for easy adoption to various OpenCL supported devices.

2.5.1 Metadata

- **Semantics:** Performs the Number Theoretic Transform (NTT) function on a number of polynomials. The kernel has an input vector that is comprised of N polynomials with 256 coefficients each in the time domain, size N*256, and returns a vector that is comprised of N polynomials of size 256 with the coefficients in the frequency domain, size N*256.
- **Linear execution time:** Yes.
- **Description language:** OpenCL C.
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.5.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
Intel x64 i7-7700	CPU	3600	5x256 uint32	8,009
Intel x64 i5-650	CPU	3600	5x256 uint32	14,628
ZCU104 – Arm Cortex A53	CPU	1500	5x256 uint32	116,326
ZCU104 – Xilinx VITIS	FPGA	150	5x256 uint32	1232,182
AMD Radeon HD 5570	GPU	650	5x256 uint32	73,484

NVIDIA GeForce GTX 1050Ti	GPU	1291	5x256 uint32	13,175
Intel x64 HD 630	GPU	350	5x256 uint32	25,838

2.6 Inverse Number Theoretic Transform (NTT)

A generic component used, for example, in the Dilithium Digital Signature developed in the *Task 3.5*. Meant for easy adoption to various OpenCL supported devices.

2.6.1 Metadata

- **Semantics:** Performs the Inverse Number Theoretic Transform (NTT) function on a number of polynomials. The kernel has an input vector that is comprised of N polynomials with 256 coefficients each in the frequency domain, size N*256, and returns a vector that is comprised of N polynomials of size 256 with the coefficients in the time domain, size N*256.
- **Linear execution time:** Yes.
- **Description language:** OpenCL C.
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.6.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
Intel x64 i7-7700	CPU	3600	5x256 uint32	9,285
Intel x64 i5-650	CPU	3600	5x256 uint32	16,683
ZCU104 - Arm Cortex A53	CPU	1500	5x256 uint32	116,110
ZCU104 - Xilinx VITIS	FPGA	150	5x256 uint32	940,955
AMD Radeon HD 5570	GPU	650	5x256 uint32	67,654
NVIDIA GeForce GTX 1050Ti	GPU	1291	5x256 uint32	11,668
Intel x64 HD 630	GPU	350	5x256 uint32	23,051

2.7 Pointwise Montgomery Multiplication

A generic component used, for example, in the Dilithium Digital Signature developed in the *Task 3.5*. Meant for easy adoption to various OpenCL supported devices.

2.7.1 Metadata

- **Semantics:** Performs a pointwise multiplication of polynomials in NTT domain representation performing a montgomerry reduction. The kernel has an input vector that is comprised of two N polynomials with 256 coefficients each in the NTT domain, size N*256, and returns one vector that is comprised of N polynomials of size 256 with the coefficients being the result of pointwise multiplication of the input polynomials, size N*256.

- **Linear execution time:** Yes.
- **Description language:** OpenCL C.
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.7.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
Intel x64 i7-7700	CPU	3600	2x5x256 uint32	0,982
Intel x64 i5-650	CPU	3600	2x5x256 uint32	1,676
ZCU104 – Arm Cortex A53	CPU	1500	2x5x256 uint32	20,482
ZCU104 – Xilinx VITIS	FPGA	150	2x5x256 uint32	282,696
AMD Radeon HD 5570	GPU	650	2x5x256 uint32	7,963
NVIDIA GeForce GTX 1050Ti	GPU	1291	2x5x256 uint32	3,238
Intel x64 HD 630	GPU	350	2x5x256 uint32	5,043

2.8 2D Object Detection and Tracking Using Neural Networks (NN)

A fully convolutional network to perform object detection and tracking.

2.8.1 Metadata

- **Semantics:** The component receives as input a three channel image of size 384x1248 and outputs a tensor with dimensions 24x78x9 corresponding to region proposals and class probabilities for three candidate classes, pedestrians, cars and cyclists.
- **Linear execution time:** Yes.
- **Description language:** C++/Python, Tensorflow, Keras.
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.8.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)
NVIDIA Jetson TX2	CPU (embedded)	CPU 1400 MHz	3x384x1248 uint8 RGB	670000
NVIDIA Jetson TX2	CPU (embedded)	CPU 1400 MHz	3x384x1248 float32 RGB	17000000

NVIDIA Jetson TX2	GPU (mobile)	Tegra X2 Compute Capability 6.2	3x384x1248 uint8 RGB	670000
NVIDIA Jetson TX2	GPU (mobile)	Tegra X2 Compute Capability 6.2	3x384x1248 float32 RGB	22000000

2.9 Conv2D+ReLU+maxpool

2.9.1 Metadata

- **Semantics:** Performs the most common operations of Convolutional Neural Networks (CNNs): 2D convolution, ReLU and max pool.
- **Linear execution time:** Yes.
- **Description language:** C.
- **Security functionality:** Secure implementation.
- **Reliable component:** No.

2.9.2 Measurement data

Target device	Device type	Clock frequency (MHz)	Input size	Runtime (us)	Power (W)
CPU, ARM A9	CPU	600	227X227 RGB	4696000	
FPGA, XC7Z020	FPGA	100	227X227 RGB	70000	2,20
FPGA, ZU3EG	FPGA	100	227X227 RGB	64000	2,60

2.10 Face Detection

A service of the Driver Monitoring System (DMS) Android application which includes the face detection, the extraction of the facial landmarks (i.e., eyes, mouth, nose, cheeks etc.) and the estimation of the following two possibilities: (i) the driver has his/her eyes closed, (ii) the driver is yawning.

2.10.1 Metadata

- **Semantics:** This component processes the frames coming from the smartphone’s front camera and analyses it, in order to extract information about whether the driver has his/her eyes closed, or he/she is yawning.
- **Linear execution time:** Yes.
- **Description language:** Kotlin/Java.
- **Security functionality:** None.
- **Reliable component:** No.

2.10.2 Measurement data

Target device	Device type	CPU (GHz)	Input size	Runtime (us)
Honor 10	Smartphone	4x2.4 & 4x1.8	30fps	3000
Huawei P30 Pro	Smartphone	2x2.6 & 2x1.92 & 4x1.8	30fps	2000
Samsung Galaxy A31	Smartphone	2x2.0 & 6x1.7	30fps	3000

2.11 Pose Detection

A service of the DMS application which includes the extraction of several body landmarks of the upper body of the driver (i.e., shoulders, elbows, wrists) and the estimation of his relevant position (i.e., if he/she has his/her hands on or off the wheel).

2.11.1 Metadata

- **Semantics:** This component processes the frames coming from the smartphone's front camera and analyses it, in order to extract information about driver's relevant position.
- **Linear execution time:** Yes.
- **Description language:** Kotlin/Java.
- **Security functionality:** None.
- **Reliable component:** No.

2.11.2 Measurement data

Target device	Device type	CPU (GHz)	Input size	Runtime (us)
Honor 10	Smartphone	4x2.4 & 4x1.8	30fps	1000
Huawei P30 Pro	Smartphone	2x2.6 & 2x1.92 & 4x1.8	30fps	1000
Samsung Galaxy A31	Smartphone	2x2.0 & 6x1.7	30fps	2000

2.12 Heart Rate Monitoring

A service of the DMS application which extracts driver's heart rate through his/her smartwatch while he is driving.

2.12.1 Metadata

- **Semantics:** This component is responsible for retrieving driver's heart rate from his/her smartwatch in order to achieve better results regarding driver's drowsiness level. To achieve that, the component is communicating with Google *Fit* app, while the rate of retrieving driver's heart rate, depends on the smartwatch's capabilities.
- **Linear execution time:** Yes.
- **Description language:** Kotlin/Java.
- **Security functionality:** None.

- Reliable component: No.

2.12.2 Measurement data

Target device	Device type	Sensor	Runtime (s)
Xiaomi Amazfit Bip	Smartwatch	PPG	12

3. Modeling Extra Functional CPSoSAware KPIs

This section summarizes the methodology followed to model the extra-functional KPIs within the CPSoSAware project, reports the list of all the KPIs currently defined, and presents some of them in details. We finally concentrate on methods to extend the component models (meta-models) with specific KPIs related to non-functional requirements with a focus on specific annotations for security.

The first step is to define the extra functional KPIs (or non functional KPIs). They are defined as all the other KPIs. To define them, we decided to use the methodology conceived within the framework of the H2020 CERBERO project [1]. We selected this methodology because of its generality and its capability of expression were perfectly suiting the needs of the CPSoSAware project. Here we recall the principles of that KPI definition methodology. The pillar of the methodology is the definition of the concept of KPI, that is done in the following way: "A KPI is a quantifiable parameter associated with a metric. A KPI evaluates one critical parameter of a CPS and evaluate the discrepancies from its long term goals". KPIs are expected to have the following properties:

- KPIs are always defined with a metric: the metric depends on the system
- KPIs are measuring a specific CPS: the set of KPIs used to measure a system are specific to that system
- Each KPI belongs to a family: KPIs is tailored to the CPS, but it will belong to a family. Properties of family of KPIs are the reusable element of the library of KPIs
- KPIs are always defined with structured/common formalism: formally defined KPIs automatically inherit all the properties of a family
- KPIs drives the evaluation of the system during the whole live cycle: KPIs are used at design time, but they also be used to drive the adaptation

To collect the KPI definition, we followed an iterative procedure that involved all the partners. The procedure we followed to collect and formalize the KPIs was the following. Firstly, we prepared a form to collect the description of the KPI in English language. The form contained the following fields: KPI Name, Class, Description, Metric, Metric Computation. The form was sent to all the partners of the project. All the forms have been collected and revised. From the definition in English, it was extracted a formal definition expressed in mathematical language, and other characteristics of the KPI were highlighted (such as addressed properties). The mathematical definition of each KPI and the updated forms were sent back to for review. This definition will be used for the realization of MATLAB models, described in Section 6.

By their nature, KPIs are a thing that evolves and changes. We thus did not close the list of KPIs. On the contrary, we expect that such list will be update and modified during the whole evolution of the project. Currently, we have identified and defined more than 30 KPIs (24 of them have been also formalized), while 10 other are currently undergoing the process of definition. For each KPI, we define the following fields: KPI Name, KPI Class, Goal, Addressed Property, Description, Metric, Metric Computation, KPI Mathematical Definition. We discuss one example in details to explain in details each of these fields and we report few of them in the rest of the section as demonstrative example.

Example: Algorithm Update Speed

- *KPI Name: Algorithm Update Speed*

This is simply the name of the KPI

- *KPI Class: Additive*

This is the class of the KPI. The concept of class (or family) of KPI was part of the CERBERO KPI methodology that we decided to use to model the KPIs of CPSoSAware [1]. In a nutshell, a class of KPI is a group of KPIs having the same properties, in particular, a group over which the same algebraic operations are valid. For instance, the elements belonging to the class "additive" are all elements where the operation addition is valid and that, for instance, they are all monotone.

- *Goal: Minimize*

This is the goal of this KPI. In this case, we want to achieve the minimum time needed to update the algorithm, so the goal is to minimize the value of the KPI.

- *Addressed Property: Algorithm Speed*

This is the properties that the KPI is covering (algorithm speed, safety, security, reliability, …). In this case, the property is algorithm speed.

- *Description*: Time needed by the algorithm to update location and vehicle behaviour should be within real time constraints (e.g., max 100ms)

This is the English description of the KPI, in a text that is understandable by a human being.

- *Metric*: Algorithm's worst case execution time

This is the metric that KPI is computed. In this case we are computing the worst case execution time, since the algorithm speed update is equal to the worst case execution time needed by the algorithm to update the location of the vehicle.

- *Metric Computation*: Measured time between update of sensor inputs till response to updated inputs

This is description, in English, of the way in which the metric described in the section "Metric" is computed. The algorithm execution time is computed measuring the time that passes between the update of the sensors used to locate the vehicle and the response that updates the inputs.

- *KPI Mathematical Definition*:

MAX(T_Alg_input_update(i) - T_sensorResponse(i)) for all i measurements for all sensor input

This is the mathematical formulation of the computation of the KPI. This is the formulation that is then directly modelled using the matlab language. In the case of this example, the worst-case execution time is computed as the maximum of the differences between the time in which a sensor respond to a change and the time after which the algorithm reacts to the change in the input values coming from the sensors.

The list of extra-functional KPIs is maintained in an excel document that is stored in a repository shared among partners and constantly updated. Changes in the KPIs formulation, removal or addition to the list are promptly recorded in the file.

The KPIs defined here are used also to extend system models. At high level, the whole method used in CPSoSAware to annotate all the non-functional requirements (including some very specific, such as the security related ones) consists of three main steps. The first is to identify and define, following the methodology previously described, all the non-functional KPIs related with the use case. Once the KPIs are defined, they are modelled, using the same approach used of all the other KPIs. Finally, the models of the non-functional extension are instantiated within the model of the components where non-functional requirement are needed. We will discuss how this process works in details in the rest of this section.

In the CPSoSAware project, we defined all the KPIs using the methodology described above. As mentioned, one of the most important characteristics of the selected methodology for modelling KPIs is that is general, namely it can be used to define any type of KPI. This includes, in particular, also the non-functional and the security related KPIs which we need to define in this project. We thus define non-functional requirements and security extension in the same way as other KPIs. Non-functional requirements and security requirements could be difficult to measure, as often they are simply a requirement such as “the system should be resistant against side channel attacks” or “the data between these components should be encrypted”. In the first case, we changed the definition of the KPI into “The system should be resistant against side channel attacks carried out using up to K number of traces”, introducing a threshold (K, the number of traces available for the attacker) that allows us to quantify the resistance. In the second case, we used a boolean metric that indicates if functionality to encrypt the communication is present or not. As such, they should be defined with a mathematical formalism and they should be defined together with a metric to evaluate them.

All the KPIs (and, as part of them, the metric to measure them) identified so far within the CPSoSAware project have been described in English language, defined by means of a mathematical function and modelled in matlab language (the description of the models is detailed in Section 6 of this deliverable). The final step is to integrate the models of the non-functional and security KPIs with the model of the component. This is done by calling the functions that evaluates the KPIs from the model of the component. The function call should include all the parameters of the model, including relevant thresholds, to correctly evaluate the required KPIs. The task of correctly passing all the needed parameters is left to the designer of the model. However, this is not a problem, since the selection of matlab as modelling tool for KPIs allows a large flexibility in interfacing. The matlab model allows us to demonstrate KPI evaluation in practice with the input variables fed in.

4. AlmaIF v2: Common Hardware Component Interface

The main goal for the ALMARVI Common Hardware IP Interface (AlmaIF) is to enable plug'n play style of customization of the hardware platform at the system level, by allowing adding accelerators from different partners to a new hardware platform design with an easy integration to a common OpenCL-based heterogeneous system software platform. The AlmaIF concept was created originally in ALMARVI ECSEL JU project, where the v1 of the interface was defined and tested on FPGA in a laboratory setting [2]. Over the CPSoSAware project, the key issues in the AlmaIF v1 were identified with an improved v2 proposed to provide IP wrapping for the purposes of the CPSoSAware project.

The key motivation for the AlmaIF is to provide a hardware interface that is generic enough for sharing driver code in the OpenCL implementation layer. A hardware abstraction layer such as AlmaIF makes it easier to plug in different AlmaIF-compliant devices in a compute platform with only minimal changes to the driver code. This helps in making the CPSoSAware execution platform composable: as devices share a common control interface, it reduces the integration effort at multiple levels due to not requiring device-specific knowledge at the simplest control functions for verification and execution purposes, for instance. The first version of AlmaIF and the general concept is more thoroughly described in [2].

Ultimately, standardizing the interfaces of the accelerators enables accelerators to communicate between each other in a peer-to-peer manner more efficiently without needing special communication and control means with each accelerator type. This philosophy is similar to the Heterogeneous System Architecture (HSA) specification with a light-weight packet processor interface [3].

A key goal of AlmaIF v2 in comparison to its first version was to provide better interfacing to fixed function accelerators, by utilizing a common OpenCL driver that understand that AlmaIF memory mapped interface with a registry of built-in kernels and associated integer IDs that the accelerator provides. The OpenCL driver, called PoCL-accel, is already upstreamed to the open source PoCL project [4].

4.1 AlmaIF v2 Changes in Comparison to AlmaIF v1

In the following, we present the hardware interface v2 that allows launching programs on different accelerators and sending results back to the host processor. The interface is designed to be sufficient to support OpenCL 1.2 programs, and thus also OpenCL 3.0 which is its minimal superset.

4.1.1 Accelerator Memory Area Packing

The HW accelerator IP can now use accelerator memory area packing (see Figure 1) to avoid wasting the address space. It gives the starting addresses and sizes for configuration/instruction, command queue and data memory regions.

The hardware decoder can become somewhat more complex if we define whatever starting address and size for the memories. However, the memory area packing can be defined by the accelerator, so it can still use a static decoder. It can even choose to use the basic decoder that utilizes the two most significant address bits used in AlmaIF v1. Addresses are set design time since the memories are design time, so it can simplify the implementation overheads.

For the memory mapped AlmaIF, this requires new data registers for the memory region starting addresses and sizes.

The flexibility makes it possible for the accelerator to implement the memory regions inside a single memory component (a single load-store unit, LSU), or split it into parallel memories etc. (3 LSUs). The driver doesn't need to know the details, it's enough that it can access it through the AXI slave using the starting addresses and sizes as defined in the AlmaIF interface.

4.1.2 AXI Master Support

The PoCL driver should know if the IP can access any physical address for directly loading/storing buffer data without copying to the on chip memory first. This is now a flag in the property space that tells that the accelerator can access AXI bus directly (the physical address space).

4.1.3 64b Addressing

Support for both 32b and 64b hardware platforms was added by making all addresses 64 bits.

4.1.4 Example Command Queue Processor

This example shows a generic small command queue processor based on either OpenASIP or a C-based wrapper meant for HLS (see Figure 2) that:

- Waits until doorbell-signalled.
- Parses AQL command queues and passes the arguments and the work-item counts etc. to an function unit that actually performs the kernel function (defined with whatever HW design means, hand-made RTL, HLS tools, ...) and blocks until finished. The FU could have I/O that exactly matches the kernel argument list. E.g. vecadd would get 3 pointers in where two are pointers to input buffers and one to the output buffer.
- Signals the listeners
- Understands the barrier packet semantics etc.

The OpenASIP implementation of the command queue processor implements the following memory map for the data memory:

1. Program data first (not controlled through AlmaIF interface)
    - Global data
    - Uninit global data
    - Heap (controlled by the _end symbol)
    - Stack (grows downwards towards heap, can be controlled with --init-sp of tcecc)
2. Global buffers (the data memory allocated by the bufalloc a.k.a. DATA memory region.)
3. AQL command queue (a.k.a. CQ memory region)

4.1.5 Other Updates

- Keep the queue format specified (HSA AQL-inspired: other devices than hosts can send work too).

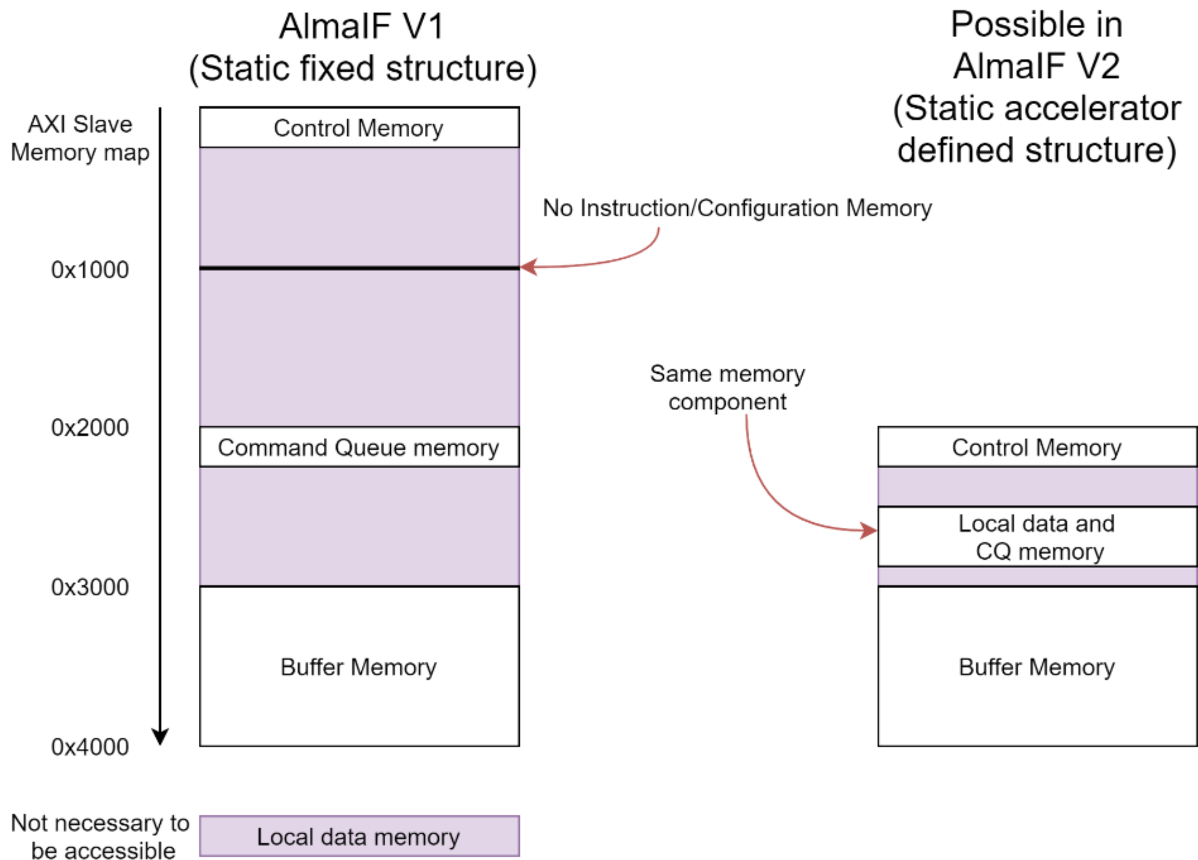

Figure 1: A comparison between AlmaIF V1 and AlmaiF V2 memory maps using an example accelerator with no need for configuration memory. In V2 the accelerator can define the starting addresses for the memory regions and use a more optimal memory area packing strategy than the static address decoding proposed in V1. In V2 the accelerator can also combine the memory components and provide a pointer to the address where the memory region begins. Naturally, in V2 the accelerator could also define the V1-style memory map if it wanted to.

4.2 Memory Addressed Registers

The base address is the only thing necessary in AlmaIF v2 to discover and control an AlmaIF-device. For this purpose, the information register segment holds at least the following read-only registers that are presented with offsets from the device's physical AlmaIF starting address.

Offset	Name	
0x000	STATUS	Status of the accelerator. Bit 0 is high when the execution is stalled due to any reason, bit 1 is high when the external stall signal is active, and bit 2 is high when the accelerator reset is active.
0x100	CQ_READ_IDX_LOW	Read index of the command queue (low 32 bits). Read only.

Offset	Name	
0x104	CQ_READ_IDX_HIGH	Read index of the command queue (high 32 bits). Read only.
0x108	CQ_WRITE_IDX_LOW	Write index of the command queue (low 32 bits). Writing to this register increments the 64-bit value.
0x10C	CQ_WRITE_IDX_HIGH	Write index of the command queue (high 32 bits). Read only.
0x200	COMMAND	Command register to control execution. Writing 1 to this register resets the accelerator, writing 2 lifts reset and external stall, and writing 4 enables the external stall signal, pausing execution (4 is an optional feature).
0x300	DEVICE_CLASS	Device class (vendor ID) of the accelerator. Currently unused by the driver.
0x304	DEVICE_ID	Device ID of the accelerator. Currently unused by the driver.
0x308	AlmaIF version	AlmaIF v1 = 1 AlmaIF v2 = 2
0x30C	Core count	Number of OpenCL Compute Units in the device.
0x310	Control memory size	32b size.
0x314	Configuration/instruction memory size	32b size. This memory is used for instructions (software programmable co-processors) or configuration bits (configurable hardware accelerators).
0x318	Configuration memory starting address.	64b starting address of the device's memory that maps instructions or the configuration bits. The actual memory can be onchip or offchip, in the latter case the device likely has a memory hierarchy with a dynamic cache. ⭐ In AlmaIF v2 this is a physical address, in v3 we want to move to virtual addressess. If we want to support partial interfacing with the virtual memory, this address should be writable by the host: It would update the starting address of the (contiguous) page for instructions after asking it from the memory

Offset	Name	
		manager of the OS. Top level 64b generic is needed for setting this for the onchip memory case: the physical address mapping should be design-time known.
0x320	Command queue memory size	64b size. The command queue fills the entire region, so the size of this memory is Max_number_of_packets * Packet_size. Max_number_of_packets must be a power-of-two.
0x328	Command queue memory starting address.	64b starting address of the device's memory that maps the AQL queue. See ⭐ above.
0x330	Data memory size	64b size.
0x338	Data memory starting address.	64b starting address of the device's memory that maps global buffers and perhaps other device-specific shared data and is visible to the AXI bus. The actual memory can be onchip or offchip, in the latter case the device likely has a memory hierarchy with a dynamic cache. See ⭐ above.
0x340	Feature flags	64b size register with boolean feature flags. Bit semantics: Bit 0 = AXI master support. Bit 1 = writable instruction memory starting address. Bit 2 = writable command queue memory starting address. Bit 3 = writable data memory starting address. Bit 4 = whether 'pausing' is supported (COMMAND 4).

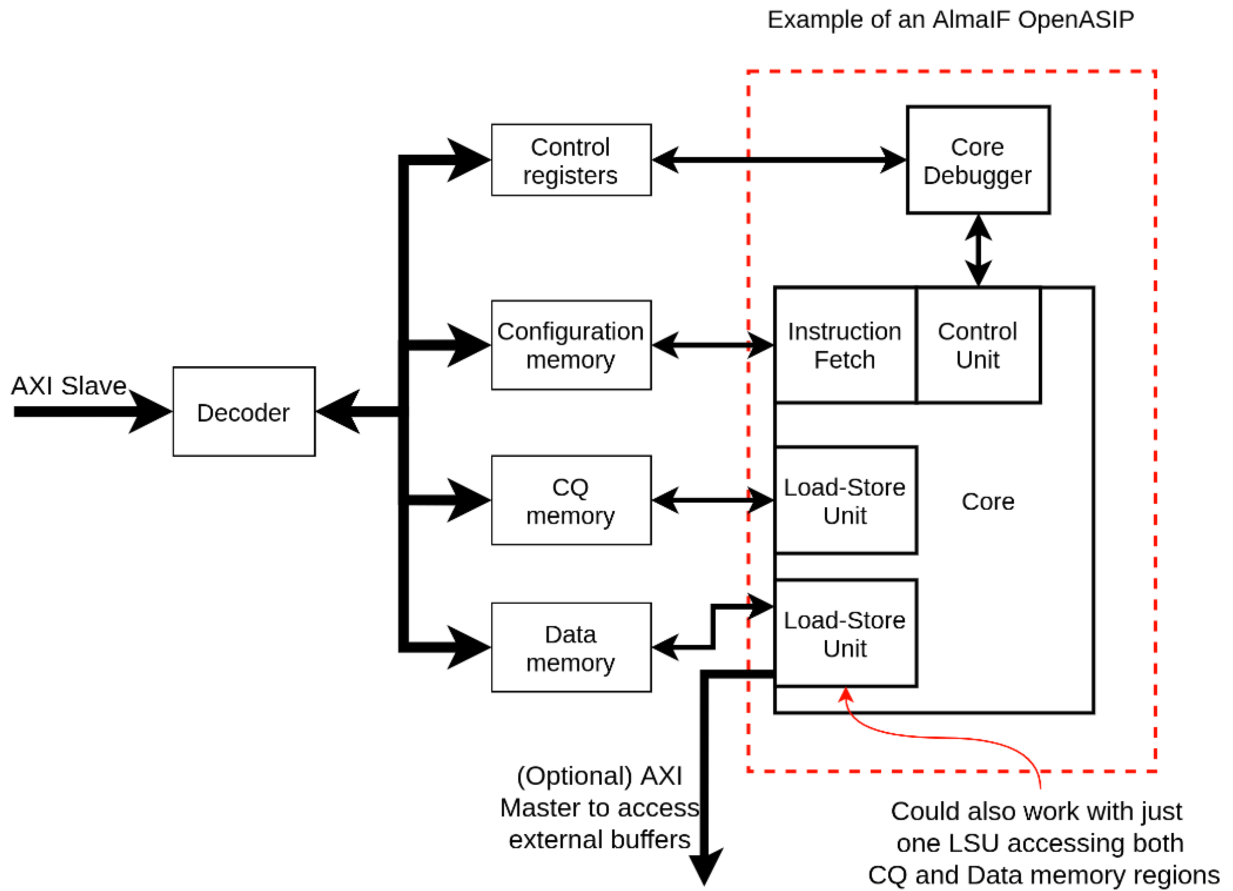

Figure 2: AlmaIF v2 with an example software programmable OpenASIP core that also handles command queue processing.

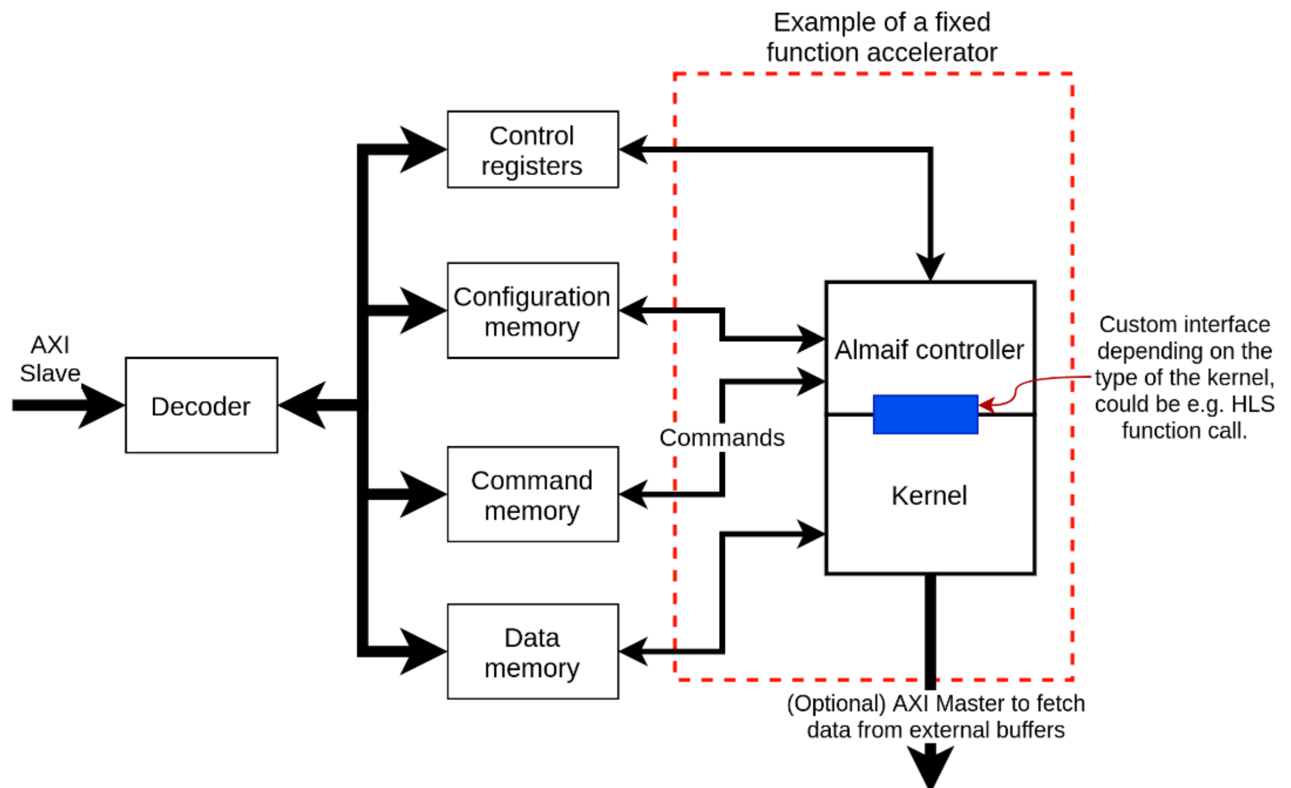

Figure 3: Generic HW accelerator with AlmaIF v2 where there is a command queue processing handling the control flow and launching Accelerator functionality implemented manually in RTL or generated from HLS.

4.3 Software Emulation / C Hardware Model

Instead of memory mapping of a fixed-function accelerator, the PoCL-accel driver also supports emulating an accelerator in software. The driver creates a software thread, which pretends to be a fixed-function accelerator with the AlmaIF interface.

There are at least three motivations for this software emulation:

1. Easier debugging of OpenCL applications, as one does not need an FPGA for testing accelerator-based programs.
2. Easier debugging of the PoCL's accel-driver.
3. The emulation function (written in C) serves as a good starting design point for HLS-based AlmaIF accelerators.

4.4 AlmaIF v2 Demonstrator

In order to ensure that modifications to the interface work in a practical environment, two OpenASIP-based accelerators with read-only arrays as instruction memories are created with the right-most memory map from Figure 1: A comparison between AlmaIF V1 and AlmaiF V2 memory maps using an example accelerator with no need for configuration memory. In V2 the accelerator can define the starting addresses for the memory regions and use a more optimal memory area packing strategy than the static address decoding

proposed in V1. In V2 the accelerator can also combine the memory components and provide a pointer to the address where the memory region begins. Naturally, in V2 the accelerator could also define the V1-style memory map if it wanted to.. The accelerators can execute a set of built-in kernels (vector addition, multiplication and blinking a led). The accelerators are simultaneously placed on a Zynq-7020 FPGA and controlled using PoCL running on 32b ARM Cortex-A9. A led blinking and a simple vector arithmetic OpenCL programs are evaluated on it and the output result is verified. A demo video of the system in action can be viewed in YouTube: https://www.youtube.com/watch?v=pBQsOfalKZk. Two screenshots of the demonstrator are shown in Figure 4 and Figure 5.

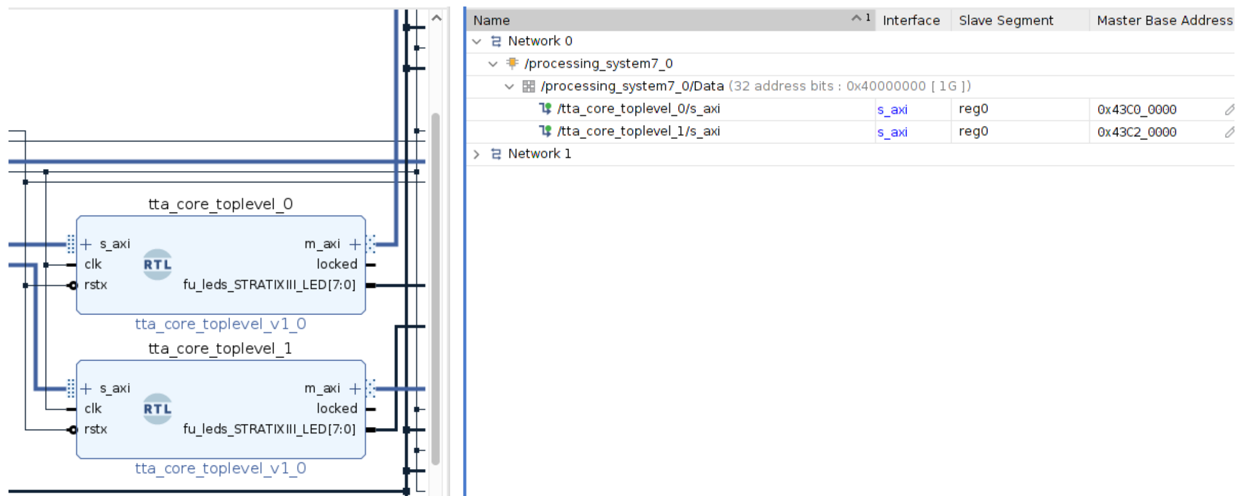

Figure 4: Two AlmaIF accelerators placed in Vivado block design for Zynq-7020 FPGA. On the right are the physical addresses used for the driver to find the accelerator devices.

```
ynq:~/pocl_accel_clean/examples/accel$ sudo POCL_DEVICES="accel accel" POCL_ACCEL0_PARAMETERS=0x43C00000,1,2,3 POCL_ACCEL1_PARAMETERS=0x43C20000,1,2,3 POCL_BUILDING=1 ./accel_duo
evice 0 initialized
evice 1 initialized
ect result!
ynq:~/pocl_accel_clean/examples/accel$ sudo POCL_DEVICES="accel accel" POCL_ACCEL0_PARAMETERS=0xE,1,2,3 POCL_ACCEL1_PARAMETERS=0xE,1,2,3 POCL_BUILDING=1 ./accel_duo
mulation device 0 initialized
mulation device 1 initialized
ect result!
ynq:~/pocl_accel_clean/examples/accel$ sudo POCL_DEVICES="accel accel" POCL_ACCEL0_PARAMETERS=0xE,1,2,3 POCL_ACCEL1_PARAMETERS=0x43C00000,1,2,3 POCL_BUILDING=1 ./accel_duo
mulation device 0 initialized
evice 1 initialized
ect result!
ynq:~/pocl_accel_clean/examples/accel$
```

Figure 5: A screenshot from the AlmaIF demonstration with two accelerator devices and a computation with shared data between them. The physical address 0xE triggers the driver to create an emulation thread, which can be used by itself or together with FPGA accelerators.

4.5 AlmaIF-accelerator Implemented with High-Level Synthesis Tools

This section presents the work performed by UoP in building a methodology to connect PoCL with HLS-based accelerators based on the ALMAIF interface. This is a collaborative work between UoP and TAU and it is considered an important part of the project, since through the proposed step-by-step approach it is possible to offload specific computational kernels to fixed-logic accelerators generated by the Xilinx HLS tool, which proves the portability of the approach.

More specifically, for the interface to be usable for other than OpenASIP-cores, there should be an easy and systematic way to modify existing accelerators to implement the AlmaIF interface so they can be seamlessly integrated to the PoCL platform.

Figure 3 shows a high level design in which a separate AlmaIF controller is used to handle the AlmaIF control interface and command queue processing. This AlmaIF controller needs to be easily available and modifiable depending on the type of the accelerator. Therefore, defining it in an HLS tool makes it possible to easily customize it and to plug it directly into HLS-based accelerators.

A prototype example of this implementation consists of a simple accelerator, called AddTwo accelerator, which adds a constant number (number in the example) to all elements of an array data if the content of the first element of the given array is equal to one. Figure 6 shows the HLS implementation of the AddTwo accelerator.

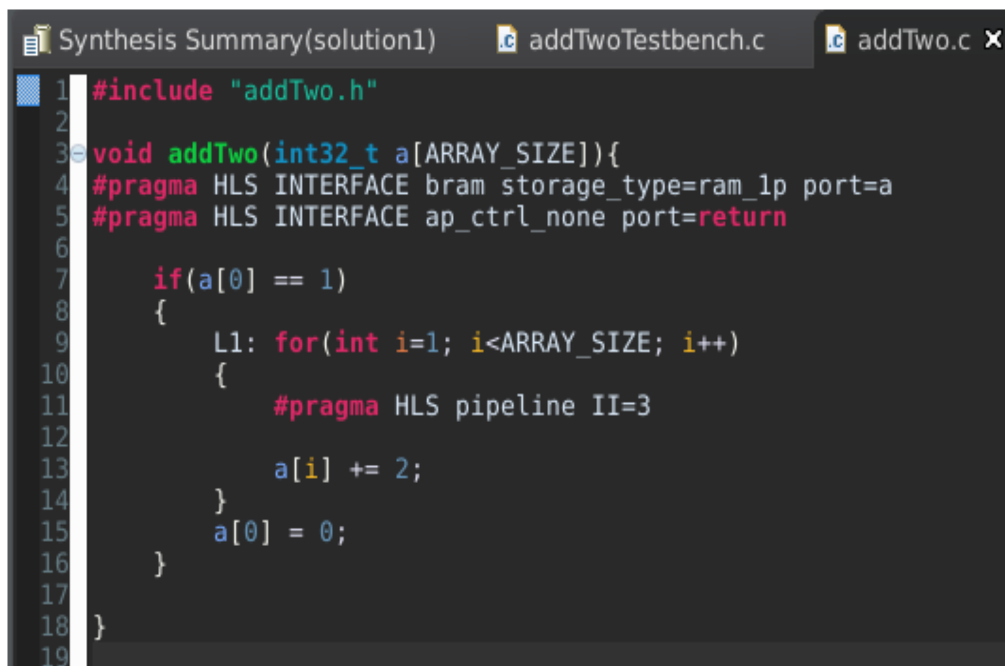

Figure 6: The HLS implementation of the AddTwo accelerator.

This accelerator is designed to have a single BRAM interface. A dual port BRAM is used for implementing the memory part of the AlmaIF interface. The first port of this BRAM is connected to the AddTwo accelerator and the second port is connected to the host system; that is an ARM processor in a Zynq FPGA SoC. Figure 7 depicts the interface of the AddTwo accelerator.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	addTwo	return value
ap_rst	in	1	ap_ctrl_none	addTwo	return value
a_Addr_A	out	32	bram	a	array
a_EN_A	out	1	bram	a	array
a_WEN_A	out	4	bram	a	array
a_Din_A	out	32	bram	a	array
a_Dout_A	in	32	bram	a	array
a_Clk_A	out	1	bram	a	array
a_Rst_A	out	1	bram	a	array

Figure 7: The interface of the AddTwo accelerator.

The accelerator is exported as an IP to be used in a Vivado project which implements the processing system that will run the PoCL middleware, the interface to the accelerator, and the AddTwo accelerator. Figure 8 shows the block diagram of the AddTwoPlatform Vivado project. In this figure, the AddTwo accelerator is part of the PoCL platform which supports FPGA accelerators.

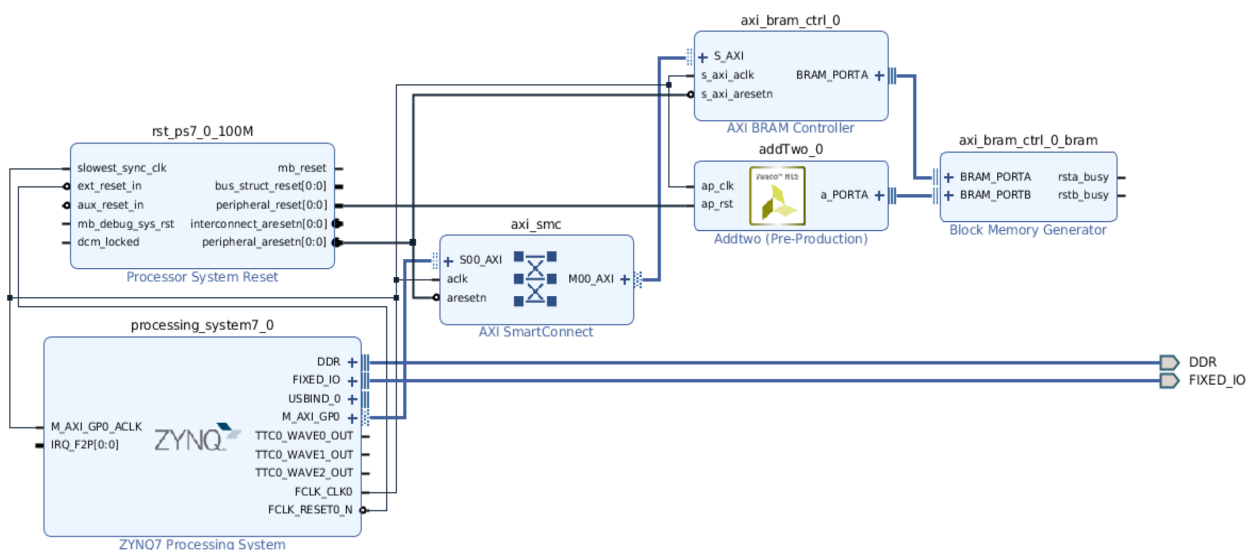

Figure 8: The Vivado project of PoCL platform supporting FPGA accelerators.

The platform shown in Figure 8 consists mainly of an ARM processing system, a BRAM controller, the accelerator, and the BRAM. The BRAM can be accessed by both the processing system and the accelerator. The processing system has access to the BRAM through the physical system address of the BRAM controller block as shown in Figure 9.

Diagram | Address Editor | IP Catalog

Assigned (1) | Unassigned (0) | Excluded (0) | Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF

Figure 9: The physical address of the BRAM controller module.

The Vivado project of Figure 8 is built and exported along with the corresponding FPGA bitstream to be used by the PetaLinux tools. The PetaLinux tools will generate the boot files and the filesystem of a Linux distribution which will run PoCL and it will provide the software interface to the accelerator. Figure 8 shows the Vivado project export and Figure 10 the Petalinux build output.

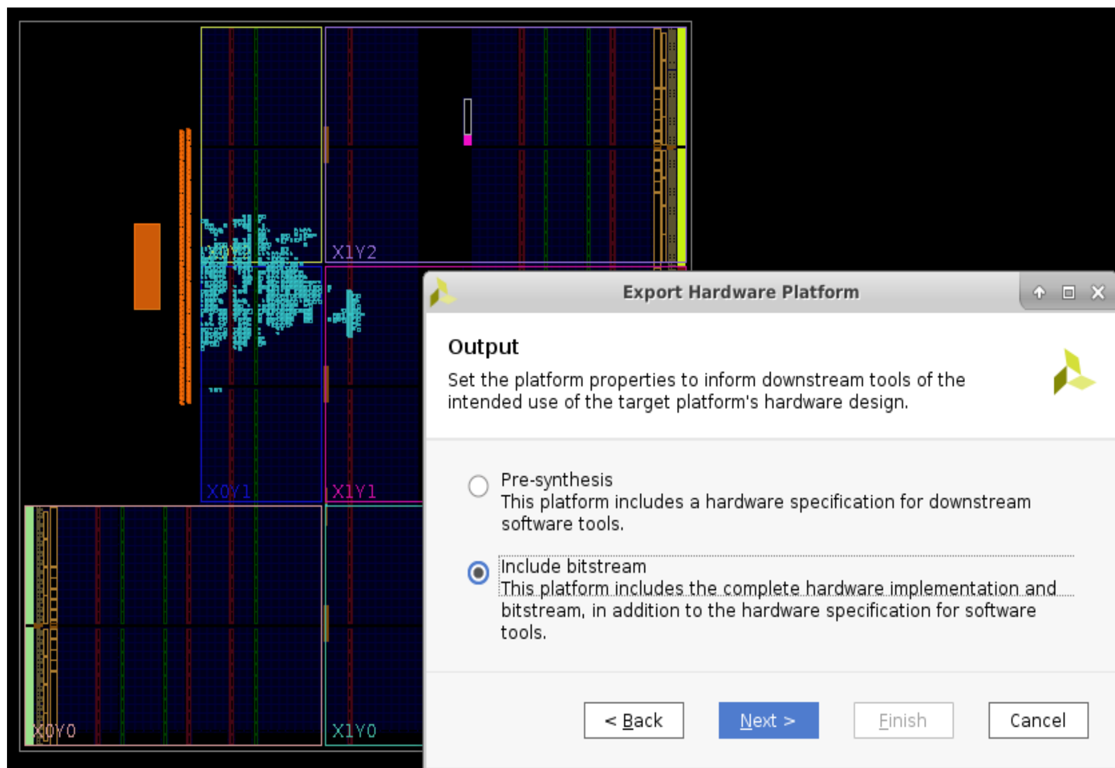

Figure 10: Export of the Vivado project to be used by the PetaLinux tools.

```
pmousoul@GPGPU:/mnt/terabyte/pmousoul_data/projects/petalinux_2020.1/addTwoLinux$ petalinux-build
INFO: sourcing build tools
[INFO] building project
[INFO] sourcing build environment
[INFO] generating user layers
[INFO] generating workspace directory
INFO: bitbake petalinux-image-minimal
Parsing recipes: 100% |#########################################################################################
Parsing of 2961 .bb files complete (0 cached, 2961 parsed). 4230 targets, 190 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
NOTE: Fetching uninative binary shim from file:///mnt/terabyte/pmousoul_data/projects/petalinux_2020.1/addTwoLinux
1f216/x86_64-nativesdk-libc.tar.xz;sha256sum=9498d8bba047499999a7310ac2576d0796461184965351a56f6d32c888a1f216
Initialising tasks: 100% |######################################################################################
Checking sstate mirror object availability: 100% |#############################################################
Sstate summary: Wanted 2247 Found 1874 Missed 373 Current 0 (83% match, 0% complete)
NOTE: Executing Tasks
NOTE: Setscene tasks completed
NOTE: Tasks Summary: Attempted 7276 tasks of which 5647 didn't need to be rerun and all succeeded.
INFO: copy to TFTP-boot directory is not enabled !!
[INFO] successfully built project
```

Figure 11: Successful build of the PetaLinux project.

In order to validate the correct operation of the PoCL HLS prototype, the PetaLinux project result files are written to an SD-card which is used for booting the Zedboard which hosts a xc7z020 FPGA SoC device. Figure 12 shows the board and the PetaLinux login screen.

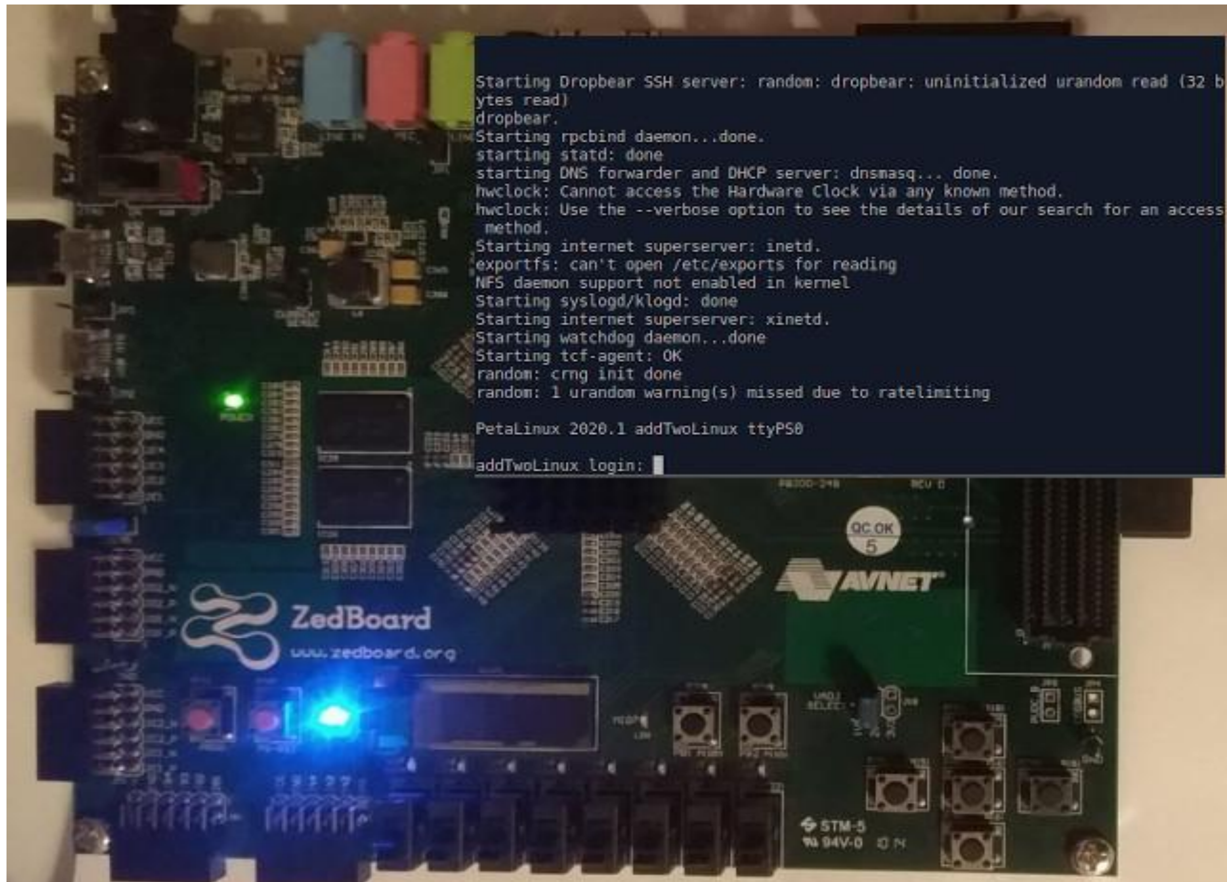

Figure 12: Zedboard running PetaLinux.

After PetaLinux boot process is finalized, we use the devmem utility to test the AddTwo PoCL prototype accelerator. After boot, the BRAM contents are all zero. Numbers 1, 2, and 3 are written in the 1st, 2nd, and 3rd 32-bit addresses. When the content of BRAM address 0 becomes equal to 1, the accelerator adds 2 to every content of the BRAM (up to the address defined by the AddTwo accelerator). Figure 13 shows the testing of the operation of the AddTwo PoCL prototype accelerator in PetaLinux.

```
root@addTwoLinux:~# devmem 0x40000004 32 1
root@addTwoLinux:~# devmem 0x40000008 32 2
root@addTwoLinux:~# devmem 0x4000000C 32 3
root@addTwoLinux:~# devmem 0x40000004
0x00000001
root@addTwoLinux:~# devmem 0x40000008
0x00000002
root@addTwoLinux:~# devmem 0x4000000C
0x00000003
root@addTwoLinux:~# devmem 0x40000000 b 1
root@addTwoLinux:~# devmem 0x40000000
0x00000000
root@addTwoLinux:~# devmem 0x40000004
0x00000003
root@addTwoLinux:~# devmem 0x40000008
0x00000004
root@addTwoLinux:~# devmem 0x4000000C
0x00000005
root@addTwoLinux:~# devmem 0x40000010
0x00000002
root@addTwoLinux:~#
```

Figure 13: Testing of the PoCL-accelerator prototype in Linux running on the FPGA SoC platform.

The next steps of this development process are to use the actual PoCL sofware to control the HLS accelerator and also to improve the AlmaIF implementation. More specifically, UoP will continue to work in this direction in order to build a demo of the above approach showcasing the connection of PoCL middleware to a more complex HLS-based fixed logic accelerator within the WP3 and 5.

4.6 Future Work: AlmaIF v3+ Ideas

Some of the features that were discussed while conducting the Task 2.3 that will be useful and might be partially implemented within the Task 3.6 in relation with the work on the OpenCL programmable soft cores as follows:

- MMU support for passing virtual addresses around the system with paged memory management. This would enable implementing the **OpenCL 2.0 SVM** and make accelerators much more flexible since they would not be reliant on continuous buffer regions.
- **OpenCL 2.0 pipes** support. Task pipeline based streaming is essential on getting good performance out of FPGAs, thus good support for FIFO-based I/O should be added to the interface.
- Support for **RDMA-like** technologies for directly accessing the local memories of accelerators *across network*.

5. LLVM-Based OpenCL Kernel Instruction-Mix Characterizer

Profiling based techniques have gained much attention on computer architecture and software analysis communities. The target is to rely on one or more profiling tools in order to identify specific code pieces of interest e.g., code pieces that slowdown a given application. The extracted code pieces can be further modified and optimized. In general, the profiling tools can be classified as deterministic, statistical-based, or hardware performance counter-based.

A common characteristic of the available profiling tools is typically based on analysing or even manipulating (in case of binary instrumentation tools) machine-level code. This approach come with two main drawbacks. First, a lot of information (even GBytes of data) needs to be gathered, stored, post-processed, and visualized. Second, the performed analysis of the gathered data is platform-specific and it is not straightforward to categorize the given applications/program phases/kernels into distinct categories that have the same or almost the same behaviour (e.g., the same percentage of computational vs. control instructions). The latter stems from the fact even small changes in the source code of the applications might lead to significantly different machine code implementations. Therefore, even two specific program kernels exhibit the same behaviour (e.g., they have the same number of instructions, but with a different ordering), it is very difficult for a machine-code level profiling tool to assess their similarity, simply because the generated machine level code might have significant differences resulting in many missing opportunities for the available profiling tools. To address this issue, as part of Task 2.3 of the CPSoSaware project, we developed a profiling tool that is able to operate on the machine independent intermediate representation (IR) level. The input to the profiler can be either C/C++ or OpenCL code.

The goal of developed profiler is to classify different OpenCL kernels wrt. the number of IR-level instructions. The profiler relies on the LLVM API and it is able to hierarchically (at various levels of the call stack) and recursively parse the IR code and extract various useful statistics. As part of this deliverable, we showcase the design and implementation of the LLVM-based profiler by analyzing a subset of the PolyBench benchmarks. The output of the profiler will act as a “signature” of the OpenCL kernels that will be executed on the end-device. The outputs will be provided to the WP4 to be used for the modelling related activities performed as part of the work package.

5.1 The LLVM IR Instruction Mix Profiler

The LLVM toolchain [6] ships with a number of passes that can operate at various levels. These passes are divided into two main categories: analysis passes and transformation passes. Analysis passes only extract information from the source code of the input application, while transformation passes can also apply specific program transformations, modifying functions, basic blocks or whole programs. For the profiler created for the project, we developed an analysis pass that is built on top of existing LLVM passes.

The profiler pass is modular and it can be easily integrated in the LLVM codebase, since it relies on the LLVM API. As a first step, the LLVM pass takes as its input an LLVM IR module and outputs a call graph. The call graph information is a useful representation of the input IR code (thus, the input application) in order to understand the basic structure and organization of the code. However, it is not suitable for more complex purposes such as power consumption analysis, security analysis etc. To accomplish the latter goals, our proposed LLVM pass is extended to record and classify the various IR opcodes of the input source codes.

The developed profiler pass includes a classification step that is able to classify the IR opcodes into specific categories. These categories are given as inputs to the classifier. A high level design of the developed profiler is shown in Figure 14: The LLVM-pass takes as input the LLVM IR code and a mapping between IR instructions and their corresponding category and outputs: i) The different IR instructions organized in categories for the input application and ii) the callgraph. This means that the IR level statistics can be given at a hierarchical manner wrt. the functions of the program.

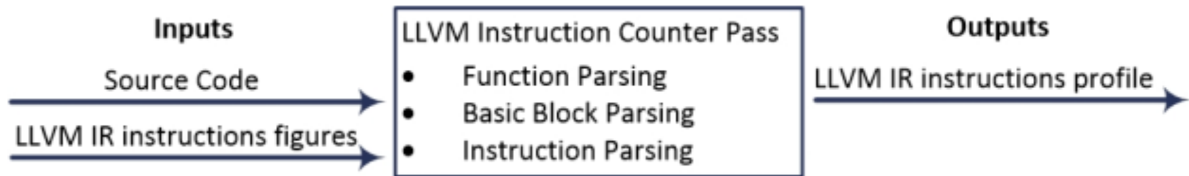

Figure 14: Profiler overview.

5.2 Profiler Design

The LLVM project provides a well-organized API that allows to add additional LLVM transformations with new features. In the context of the profiler that produces statistics of the different instruction types, the key API is the *Instruction* class of LLVM. We implemented new hooks and features in order to classify the IR opcodes of the input application into different IR categories. More specifically, the LLVM toolchain provides a class called *Function* that represents a function in the LLVM IR. The class *Function* provides an iterator for each *BasicBlock* class and in each basic block an iterator for its *Instruction* classes. In Figure 15, these steps are annotated as (4) and (5). Therefore, we have at least three abstraction layers in our source code to begin with.

Moreover, all LLVM passes are subclasses of the LLVM *Pass* class and their functionality comes from overriding the virtual methods inherited from the *Pass* class. For this work, we overrode the function *runOnFunction* (step 1 in Figure 15). The pass iterates through the various functions in the IR, the basic blocks of each function, and finally the instructions of each basic block. During this process, the profiler holds a record for every visited instruction and updates a data structure with their total numbers of occurrences (step 6 in Figure 15). In addition, it operates recursively when the instruction ”call” is visited (step 7 in Figure 15) in order to create a call graph for the function and accordingly count all the instructions in the created call tree. Finally, the classification of each instruction, based on the predefined (input) categories, is performed. The output of the profiler is a call graph for each function, an analysis of all its opcodes, and a power figure analysis (as a result of the classification of the LLVM IR instruction types).

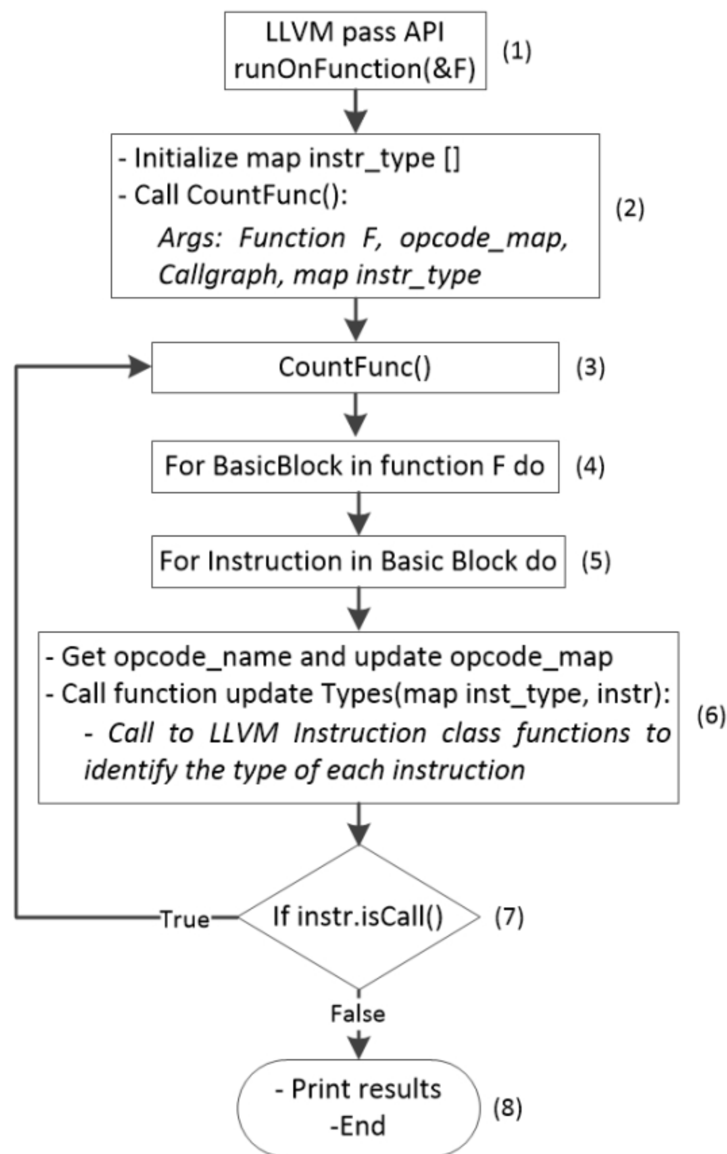

Figure 15: Profiler block diagram.

5.3 Demonstrator: Example Runs of the Profiler

For validation purposes we present an example of an OpenCL kernel (see Figure 16), on which we applied our LLVM pass.

```
1   __kernel void doitgen_kernel2(int nr, int nq, int
       np, __global DATA_TYPE *A, __global DATA_TYPE *
        C4, __global DATA_TYPE *sum, int r)
2   {
3       int p = get_global_id(0);
4       int q = get_global_id(1);
5
6       if ((p < np) && (q < nq))
7       {
8           A[r * (nq * np) + q * np + p] = sum[r * (nq
        * np) + q * np + p];
9       }
10  }
```

Figure 16: Source code of doitgen kernel 2.

This OpenCL kernel is part of the doitgen benchmark from the Polybench Suite [POLY]. This kernel is used for Multiresolution Analysis or MADNESS and is typically is used in linear algebra for solving integral and differential equations of many dimensions (more details about this kernel is out of the context of this work). This source code corresponds to the LLVM IR code (shown in Figure 17) as it is generated by the clang compiler.

```
1  @doitgen_kernel2(i32 %nr, i32 %nq, i32 %np, float*
       nocapture
2   %A, float* nocapture readnone %C4, float* nocapture
        readonly %sum,
3   i32 %r)  {
4   entry:
5     %call = tail call i64 @_Z13get_global_idj(i32 0)
       #3
6     %conv = trunc i64 %call to i32
7     %call1 = tail call i64 @_Z13get_global_idj(i32 1)
        #3
8     %conv2 = trunc i64 %call1 to i32
9     %cmp = icmp slt i32 %conv, %np
10    %cmp4 = icmp slt i32 %conv2, %nq
11    %or.cond = and i1 %cmp, %cmp4
12    br i1 %or.cond, label %if.then, label %if.end
13
14  if.then:                                          ;
        preds = %entry
15    %mul6 = mul i32 %r, %nq
16    %reass.add = add i32 %mul6, %conv2
17    %reass.mul = mul i32 %reass.add, %np
18    %add8 = add i32 %reass.mul, %conv
19    %idxprom = sext i32 %add8 to i64
20    %arrayidx = getelementptr inbounds float, float*
       %sum, i64 %idxprom
21    %0 = load float, float* %arrayidx, align 4, !tbaa
        !8
22    %arrayidx15 = getelementptr inbounds float, float
       * %A, i64 %idxprom
23    store float %0, float* %arrayidx15, align 4, !
       tbaa !8
24    br label %if.end
25
26  if.end:                                           ;
        preds = %if.then, %entry
27    ret void
```

Figure 17: LLVM IR code of doitgen kernel 2.

As we can see in lines 5 and 7 in Figure 17, there are two call instructions that correspond to the *get_global_id()* function calls in the source code. In addition, the if statement in source code (line 6 in Figure 16) is translated to the IR opcode depicted in lines 9, 10, and 11 in Figure 17. Line 12 is the branch instruction coming from the *if...then* label or the *if...end* label. The body of the if statement corresponds to the lines 15-24 inside the *if...then* label.

```
Function: doitgen_kernel2
add :: 2
and :: 1
br :: 2
call :: 2
getelementptr :: 2
icmp :: 2
load :: 1
mul :: 2
ret :: 1
sext :: 1
store :: 1
trunc :: 2

Power type profiling
NoType -> 5
type1 -> 3
type2 -> 2
type3 -> 4
type4 -> 5

Callgraph for doitgen_kernel2
Null
```

Figure 18: Output of the profiler.

The Figure 18 illustrates the output of the proposed profiler. The top part of the listing enumerates the IR level statistics for the two functions of the studied kernel (lines 1-5 and lines 6-13). The bottom (lines 15-20) depicts the IR level instruction classification statistics. More details about the classification process will be given in the next subsection.

5.4 Results

5.5.1 Approach

To generate the LLVM IR bitcode from the OpenCL kernels, we use the clang compiler version 12.0.0. We also use the opt tool from the LLVM toolchain. To perform the IR instruction classification, we accordingly focus on the Instruction class of the LLVM codebase. Finally, we apply our profiler to the Polybench-ACC benchmark suite [5] and more particularly in the linear algebra OpenCL applications. Each application consists of 1 or 2 OpenCL kernels (annotated as K1 or K2 hereafter).

The goal of this profiler is to classify different OpenCL kernels wrt. the number of IR-level instructions. Therefore, the next step was to classify each IR opcode to five distinct categories. Obviously, each category contains opcodes with similar or almost similar power profiles. As part of this work, the following categories were used (we will extend the categorization as part of WP4 activities):

The type1 group includes all the simple ALU operations e.g., additions, subtractions, and bitwise operations. type2 group contains the long-latency, thus more power consuming, ALU operations like multiplications

and divisions instructions. The type3 group consists of the load-store memory operations and the type4 includes all the control instructions e.g., branches, function calls, function returns etc. Finally, there is a last category (NoType) with all the remaining LLVM IR instructions.

5.5.2 Kernel Characterization

This section presents our profiler-based characterization of the following OpenCL applications from the Polybench-ACC [POLY] benchmark suite: 2mm, 3mm, atax, bicg, doitgen, gemm, gemver, mvt and syr2k. In total, 17 OpenCL kernels were used. Our goal, as explained, was to create groups of similar kernels depending on their IR instruction mix, thus groups with similar power behavior. Our classification algorithm works as follows: two kernels are grouped together if the difference in the same type instructions is lower than a predefined threshold.

In the current implementation, we set two threshold values: 10% and 20%. Table 1 contains the extracted kernel groups. As we can see from Table 1, when the first threshold value is enforced, we end up with 12 groups, while only eight groups are extracted for the second threshold value.

Table 1: Kernel classifications.

kernel clustering into groups	10%	20%
group 1	2mm k1, gemm k1	2mm k1, gemm k1
group 2	2mm k2	2mm k2, 3mm k1, 3mm k2, doitgen k1, gemver k3
group 3	3mm k1, 3mm k2, doitgen k1	atax k1, bicg k1, mvt k1
group 4	atax k1, mvt k1	atax k2, bicg k2, mvt k2
group 5	atax k2, mvt k2	doitgen k1
group 6	bicg k1	gemver k1
group 7	bicg k2	gemver k2
group 8	doitgen k2	syr2k k1
group 9	gemver k1	-
group 10	gemver k2	-
group 11	gemver k3	-
group 12	syr2k k1	-

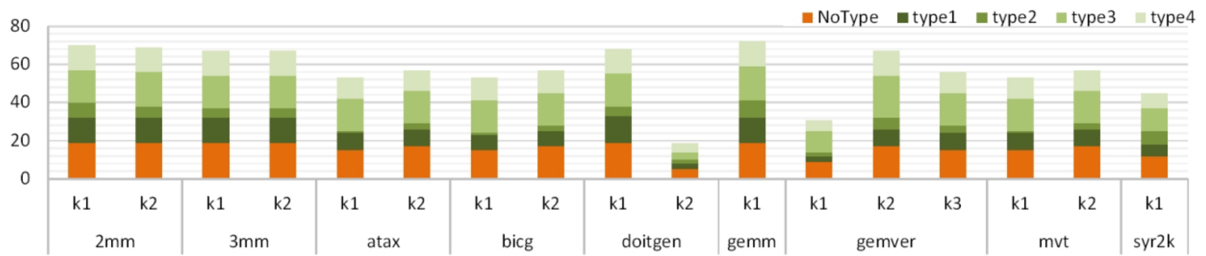

Figure 19: LLVM IR instructions classification.

Finally, Figure 19 presents the whole range of the profiler outputs for each OpenCL application and kernel (shown in the horizontal axis). The y-axis shows the absolute values of the number of instructions of each instruction type. It is obvious from Figure 19 that specific kernels (belonging to the same or to different applications) exhibit the same behavior (equal or almost equal number of instructions per instruction type) e.g., the 2mm k1 and gemm k1 kernels, while other kernels show different instruction statistics (e.g., doitgen k2 and 2mm k2), thus different run-time behaviour (to be confirmed in WP4).

6. Component Matlab Modeling Demonstrator

As a result of Task 2.4, a Matlab program based demonstrator of the component modelling methodology is described in this section. All the KPIs currently identified within the CPSoSAware project have been defined using the methodology discussed in Section 2 and modelled in MATLAB. They are then included in a library of modeled KPIs that is available to the project members in the internal project repository. We currently have 18 KPIs modelled in the library. We do not consider KPIs as static thing, but as a dynamic thing, that thus will evolve and can change during whole duration of the project. It is thus possible that few KPIs could be added or modified in the near future. The updates would be anyway made immediately available in the same repository.

We selected to use MATLAB to model the KPIs for several reasons. The first, is that, per our definition, KPIs are defined with mathematical formalism. The use of a formalism suitable for defining mathematical function was the most natural choice. The second is the flexibility and easy to interface capability of MATLAB. MATLAB programs can be written in a very flexible and very powerful language, and they can be executed also on openly available tools such as octave. Furthermore, KPIs could require the evaluation of complex mathematical functionalities to be measured. MATLAB is very suitable for these tasks. Finally, programs written in MATLAB can be easily interfaced with other components of a tool chain, since they can access and parse easily text files generated by other programs.

The modelling of KPIs starts from the mathematical definition obtained after the iterative process of KPIs definition, described in Section 2. After the confirmation of correctness of such definition, each KPI was then modelled with the language used by the MATLAB tool. Specifically, each KPI model is defined as a function in MATLAB. Each function takes the parameters needed to compute the specific KPI, a specific threshold where it is needed, and returns the computation of the KPI itself, as defined in the collection phase.

The library of all modelled KPIs is available for each partner in the internal repository of the project and will be made publicly available on the project website together with this deliverable. In the rest of this section, we report as example the code of few KPIs.

Collision_Risk KPI (Figure 20). The definition of Collision Risk is the maximum risk of collision between the current car and all the neighbouring cars. This value depends on the distance between cars and a parameter, called collision factor, that can be determined empirically or analytically. The component that models this KPI is a function that computes the Collision Risk between each of the surrounding cars. Each models contains, in the form of comments within the code, the definition of the KPI and the way in which the function has to be used.

```
%Compute the distance between cars, infer the collision risk from it
%MAX(ABS(C(i) - C(j)) x CF) for i and j different car indexes (i != j) CF:Collision Factor

%C, an 1-D matrix with n elements, with C(i)-C(j) the distance from car i to car j.
%CF, the collision factor depending on weather/illumination conditions
%Returning an nx2 matrix, i.e. M(i,1) the collision risk of i car
%                               M(i,2) the index of other related car

function M = Collision_Risk(C,CF)

    n = size(C,2);
    for i = 1:n
        m_val = intmax;
        for j = 1:n
            if(i==j)
                continue
            else
                tmp = abs(C(1,i)-C(1,j))*CF;
                if(tmp<m_val)
                    m_val = tmp;
                    ind = j;
                end
            end
        end
        M(i,1) = m_val;
        M(i,2) = ind;
    end
end
```

Figure 20: MATLAB for Collision Risk KPI modeling.

Data_Integrity_Component KPI (Figure 21). This KPI checks the presence, in each module, of a component to enforce data integrity. This is verified by verify that, for all the modules in the system, the matrix storing the presence of a component returns the value true.

```
function M = Data_Integrity_Component(D)
%Let the columns be the components of each CPS
%Returns a boolean vector, true values means
%        that all components supports data integrity

    [r,c] = size(D);
    M = ones(r, 1);
    for i = 1:r
        for j = 1:c
            if D(i,j) > 0
                continue
            else
                M(i,1) = 0;
                break
            end
        end
    end

end
```

Figure 21: MATLAB for Data Integrity Component KPI modeling.

Threat_Severity KPI (Figure 22). This KPI measures the severity of a threat, defined as the normalized maxium impact that a threat could have. The normalized value of the threat severity is computed multiplying the threat severity with the impact that the threat can have on the requirements, divided by the maximum value that the impact can have. All these values have to be stored in a matrix for each threat.

```
% S,  Threat Severity
% k,  impact of this threat on the requirements
% km, maximum impact (max value that k can obtain)
function R = Normalized_Criticality(S,k,km)

    R = S*k/km;

end
```

Figure 22: MATLAB for Thread Severity KPI modeling.

Repeatability_of_Algorithm_speed KPI (Figure 23). This KPI monitors that the algorithm speed under the same conditions is highly similar. It does so by verifying that the absolute difference between execution times of the same algorithm is smaller than a given threshold.

```
%ABS(T_Delay(i) - T_Delay(j)) < Threshold for i and j different executions (i != j)
%T_D is a vector of n different executions of an algorithm
%Threshold is the threshold of the time difference between the Algorithm
%executions
%Output:
%M is an nxx matric of all possible ABS teim differences betweeen
%executions e.g M(1,3) is the ABS of the dealy of algorithm execution 1 and 3
% T is an nxn matrix of the Threshold fails for each time difference check.
function [M,T] = Repeatability_of_algorithm_speed(T_D,Threshold)
    n=size(T_D,2);
    T=zero(n,n);
    M=zero(n,n);
    for i=1:n
        for j=1:n
            M(i,j)=abs(T_D(1,i)-T_D(1,j));
            if M(i,j)>Threshold
                T(i,j)=1;
            end
        end
    end

end
```

Figure 23: MATLAB for Repeatability of algorithm speed KPI modeling.

7. Conclusions

This deliverable described the outcomes of Task 2.3 and 2.4 of the CPSoSAware project. A key demonstrated outcome from Task 2.3 was a new version of AlmaIF, a hardware component interface that is essential in integrating hardware accelerators to common open heterogeneous platform that is utilized in the CPSoSAware project. The work on this component will continue in Task 3.2 and Task 3.6 where it will be extended and adapted for the OpenCL platform optimization.

The document also presented metadata for various hardware and software components that are going to be used in the pilot demonstrators as a result of Task 2.3 and will be used in WP4 for driving the adaptation process. The KPI collection process was outlined and examples given on how security related aspects were given. Finally, a new profiling tool, as planned in Task 2.3, was demonstrated. It is capable of analysing OpenCL kernel instruction mixes and thus will be usable in providing feedback to the soft core generation automation that is being created in Task 3.6.

References

[1] CERBERO Modelling of KPIs. D3.1 and D3.2, available at: https://www.cerbero-h2020.eu/wp-content/uploads/2020/06/D3.1.pdf and https://www.cerbero-h2020.eu/wp-content/uploads/2018/12/D3.4.pdf

[2] Hoozemans, J., van Straten, J., Viitanen, T. *et al.* ALMARVI Execution Platform: Heterogeneous Video Processing SoC Platform on FPGA. *J Sign Process Syst* **91,** 61–73 (2019). https://doi.org/10.1007/s11265-018-1424-1

[3] Heterogeneous Systems Architecture Specification version 1.0. Chapter 2: System Architecture Requirements: Requirement: Architected Queuing Language (AQL). http://www.hsafoundation.com/. Date checked: 2021-06-17.

[4] Portable Computing Language: Fixed Function Accelerators (The „accel" Driver) http://portablecl.org/docs/html/accel.html. Date checked: 2021-06-17.

[5] The Polyhedral Benchmark Suite Targeting Multicore CPUs, GPUs, and Accelerators. http://cavazos-lab.github.io/PolyBench-ACC/. Date checked: 2021-05-22.

[6] LLVM Project https://llvm.org/