D5.1 GENERATION OF HW/SW RUNNABLE

Authors	ISI and UoP partners
Work Package	WP5 – CPSoSaware Integration and Cross-layer Optimization supporting design-operation continuum

Abstract

This report contains the output of Task 5.1 regarding the generation as well as the overall process of generation of the CPSoSaware hardware and Software runnables for the tow main application pillars of the project i.e the AI pillar and the Security pillar.

Funded by the Ho work Programme
of the European Union

Deliverable Information

Work Package	WP5 – CPSoSaware Integration and Cross-layer Optimization supporting design-operation continuum
Task	T5.1 SW/HW Generation of non-functional property enhancements
Deliverable title	Definition and planning of evaluation trials
Dissemination Level	Public
Status	Final
Version Number	1.0
Due date	30/04/2022

Project Information

Project start and duration	1.01.2020-31.12.2022
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) - 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
0.1	12/02/2022	Initial draft	*ISI*
0.2	15/03/2022	UoP input on DSM	*UoP*
0.3	20/03/2022	UoP updated input on DSM	*UoP*
0.4	15/04/2022	ISI input on NTT	*ISI*
0.5	25/04/2022	ISI input on introduction, and conclusions	*ISI*
0.7	28/04/2022	ISI proofreading and updates	*ISI*
0.8	16/05/2022	Deliverable ready for internal review	*ISI/UoP*
0.*9*	*23/05/2022*	I*nternal review has been provided*	*CRF/IBM*
1.*0*	*24/05/2022*	F*inal version of the deliverable ready*	*ISI*

	NAME
Prepared by	ISI
Reviewed by	CRF, IBM
Authorised by	ISI

DATE	RECIPIENT
20/05/2022	Project Consortium
30/05/2022	European Commission

Table of contents

List of figures

List of tables

1. Executive Summary

In this deliverable we provide the necessary details on several of the runnable that have been developed in the CPSoSaware project. The deliverable is a continuation of the activities of WP2, WP3 and WP4 on how to create solutions that exploit hardware – software codesign as well as making use of possible hardware reconfiguration. The deliverable includes a report of the main findings for the hardware and software runnable as well as the relevant code (found as in git repositories that are provided within the report text).

The report is split into two parts. The first part discusses the necessary hardware runnables that are designed in WP2, WP3 and WP4 and are mainly focused on the AI pillar of the CPSoSaware project as well as the security pillar of the project (since these are the main application pillars of CPSoSaware). More specifically, we focus our analysis on how to create a series of hardware runnables for driver state monitoring (DSM) when the driver is drowsy aiming to provide very fast and very efficient detection that can trigger a system's response. Apart from that based on the profiling activities and hardware – software partitioning approaches performed in WP4 and T4.1, we also report in D5.1 hardware runnables for the most computationally intensive arithmetic operation in quantum safe cryptography schemes (that constitute the focal point of cryptographic engineering research in the recent years) which is the Number Theoretic Transform (NTT). All the hardware runnable solutions use HLS tools (specifically the Xilinx HLS framework) to generate the hardware IP cores. The hardware runnable section includes a description of the functionality and key features of each runnable, the HLS pragmas that were used and reports the findings/results when the hardware runnable is deployed in a real system (typically some Xilinx MPSoC FPGA device).

The second part of the deliverable report follows a similar approach as the first part but it is focused on the software deliverables that have been developed in the CPSoSaware project. Similarly to the first part of the Deliverable, we focus on the AI pillar and the security pillar of the project and more specifically on the DSM and the NTT runnables as those are implemented purely in software. The second section include a description of each runnable, the OpenCL code that was developed (if applicable for the runnable) as well as the main results.

The final section of the deliverable is a conclusion paragraph that summarizes the main findings and the activities of the T5.1 and the relevant deliverable.

2. Introduction

In this Deliverable we focus on the hardware and software runnables that have been developed based on OpenCL and/or HLS C/C++ code using High level Synthesis tools (for hardware) or OpenCL compilers (for software). The analysis is focused on some of the components designed and analyzed in the WP2 and mostly WP3 activities after performing the necessary profiling done in WP4 (in T4.1) to determine what should be implemented in hardware and what on software. Since the main application pillars of the CPSoSaware project are the AI pillar and the security pillar, the provided runnable solutions are for those pillars. The described runnables are indicative of the overall design flow that is been followed in the project and aim to show how a specific algorithm (and overall design solution) can be realized in actual hardware and/or software runnables.

We present a Driver State Monitoring solution that uses a camera installed inside a car to monitor the face of the driver. Each frame of this camera is analyzed and facial landmarks are aligned. The position of these landmarks and their distance can be used to recognize whether the driver is yawning, or if his eyes are sleepy. Moreover, the pose of his head can also reveal information about the driver's drowsiness level. The 2D facial landmark alignment method used in the algorithm is implemented in C++ with the open source libraries DLIB and Deformable Shape Tracking (DEST), is used in several applications such as driver drowsiness detection, recognition of facial expressions, etc. The most challenging of these applications require fast processing of video frames. Therefore, the alignment of the facial landmarks in a single video frame has to be performed with the minimum possible latency without precision loss. The fast 2D facial landmark detection algorithm that has been presented by Kazemi and Sullivan in [1] where an Ensemble of Regression Trees (ERT) is used to estimate the position of the facial landmarks, has been adopted in the DSM runnables. The algorithms and the overall experiments on how to correctly design the runnables are presented in D4.8

For the security pillar, our analysis is focused on the next generation of security/cryptography primitives that are quantum safe. More specifically, we perform research on how to efficiently implement cryptography algorithms that will remain secure even when quantum computer-based attacks (like the Shor's algorithm attacks) are possible. This constitutes one of the most important recent cryptography engineering goals in the cryptography research field. Following the profiling activities of T4.1 we identified as the main bottleneck of most of the Lattice Based Cryptography schemes (that currently dominate the quantum safe cryptography solutions), the Number Theoretic Transform operation (NTT) taking place before and after each polynomial vector multiplication taking place during the all the algorithms' execution flow. Our cryptography scheme of reference has been the Dilithium Digital Signature, but the technique is generic enough to be applicable to any other algorithm that uses NTT. Our goal in implemented the relevant NTT hardware and software runnables was to achieve low latency under a fair tradeoff with utilized resources.

In the following section we describe the HLS process that was used for the implementation of the hardware runnables (to clarify and detail the design flow that was used) and in the sections after that we describe how the above two solutions (DSM and NTT) haves been realized as hardware and software runnables.

3. HLS workflow using Xilinx Vitis Toolset

The Xilinx Vitis High Level Synthesis (HLS) tool synthesizes a C or C++ function into Register-Transfer Level (RTL) code for acceleration in the programmable logic area of a Field Programmable Gate Array (FPGA) board. Vitis HLS is tightly integrated with the Vitis core development kit and the application acceleration design flow. Some benefits of using a high-level synthesis (HLS) design methodology include:

- Developing and validating algorithms at the C-level for the purpose of designing at an abstract level from the hardware implementation details.
- Using C-simulation to validate the design and iterate more quickly than with traditional RTL design.
- Controlling the C-synthesis process using optimization pragmas to create high-performance implementations.
- Creating multiple design solutions from the C source code and pragmas to explore the design space and find an optimal solution.
- Quickly recompile the C-source to target different platforms and hardware devices.

HLS includes the following stages:

1. Scheduling determines which operations occur during each clock cycle based on:
   - When an operation's dependencies have been satisfied or are available.
   - The length of the clock cycle or clock frequency.
   - The time it takes for the operation to complete, as defined by the target device.
   - The available resource allocation.
   - Incorporation of any user-specified optimization directives.
2. Binding assigns hardware resources to implement each scheduled operation, and maps operators (such as addition, multiplication, and shift) to specific RTL implementations. For example, a multiplication operation can be implemented in RTL as a combinational or pipelined multiplier.
3. Control logic extraction creates a finite state machine (FSM) that sequences the operations in the RTL design according to the defined schedule.

3.1. Generic Vitis HLS workflow

Vitis HLS is project based and can contain multiple variations called **solutions** to drive synthesis and simulation. Each solution can target either the Vivado IP flow, or the Vitis Kernel flow. Based

on the target flow, each solution will specify different constraints and optimization directives, as described in Enabling the Vivado IP Flow and Enabling the Vitis Kernel Flow. [1]

The following steps are present in a typical design flow which is comprised of synthesis, analysis, and optimization:

1. Creation of a new Vitis HLS project.
2. Verification of the source code with C simulation.
3. Executing high-level synthesis to generate RTL files.
4. Analyzing the results by examining latency, initiation interval (II), throughput, and resource utilization.
5. Optimization and repeat from step 2 as needed.
6. Verification of results using C/RTL Co-simulation.

Vitis HLS implements the solution based on the following:

- Target flow
- Default tool configuration
- Design constraints
- Any optimization pragmas or directives.

A developer can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

3.2. Workflow followed in the development of the DSM application implemented in FPGA

The Driver State Monitoring (DSM) framework implemented by UoP consists of 2 platforms that are shown in Figure 1. The Deformable Shape Tracking (DEST) applications, particularly the video tracking, had been initially ported to an Ubuntu platform replacing the slow Eigen library calls with fast C implementations.

[1] Refer to Default Settings of Vivado/Vitis Flows for a clear list of differences between the two flows.

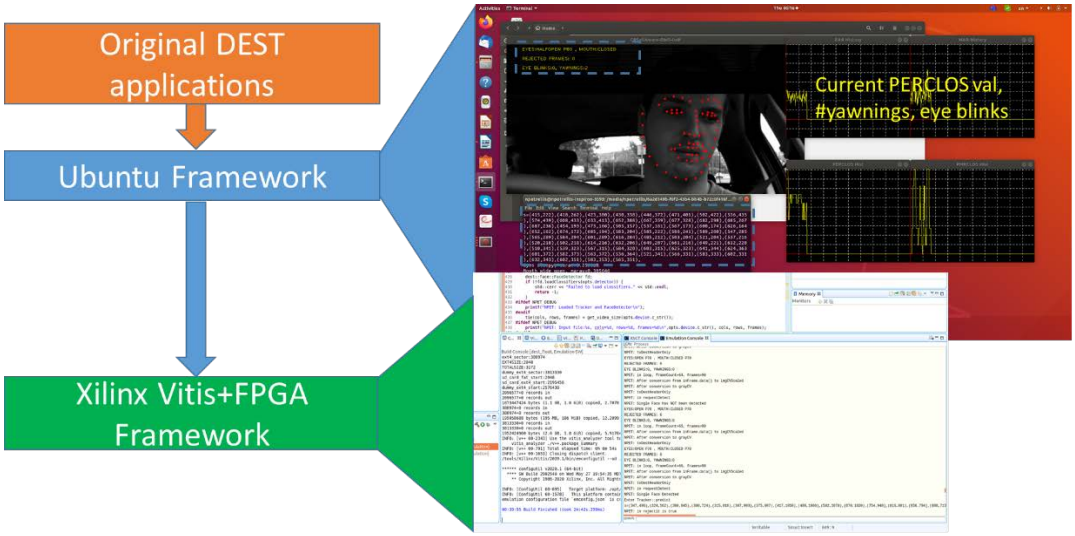

Figure 1: *The Ubuntu and Vitis+FPGA frameworks developed by UoP.*

In the Ubuntu platform, alternative shape alignment algorithms and modifications can be tested. The output of these algorithms can be directly displayed on screen for easier debugging. The input in the Ubuntu platform can either be from a camera or from a stored video. The compilation process is performed with GNU compilers that are compatible with the ones used in the target platform of the ZCU102 FPGA board.

The ZCU102 FPGA board can be directly used in the environment of the vehicle with its input being normally a camera pointed at the driver. The results of the frame processing performed by the DEST video tracking application (DSM) that has been ported to the Vitis+FPGA platform are used by higher level applications that can generate alarms if driver drowsiness is detected. In this case no video output is needed. However, for full functionality, debug and comparison reasons, the DSM application can also read input from a video file stored in the SD card of the FPGA platform and output video similar to the one produced by the Ubuntu environment which can be stored in the SD card and viewed offline.

The efficiency of the hardware acceleration techniques has been evaluated using Xilinx Vitis/Vivado HLS tools. The features that can be estimated include latency, required resources and power consumption. The facilities offered by the Xilinx XRT are used to dynamically select the optimal pair of ERT model and HW kernel according to environmental conditions as described in D3.2 and D5.2. Moreover, XRT offers real time monitoring techniques e.g., about the speed of the hardware of software functions that were also taken into consideration.

4. HW Runnables

4.1. FPGA HW runnables for the DSM application

4.1.1. Description

The algorithm for the DEST application that aligns landmarks in faces detected in the frames of a video or camera stream we used is depicted in Figure 2. In the original application, the operations in the green boxes did not exist. Frames from the video stream are retrieved and if face landmark alignment is needed in the specific frame, face detection takes place using the OpenCV library.

The coordinates of the face bounding box returned by OpenCV are used in the next step that performs landmark alignment using the *predict()* function[2]. The Similarity Transform (ST) process has to be applied on the detected face bounding box to adapt its coordinates to the ones used by the mean shape stored in the trained model. This model consists of a number of regression trees in each cascade stage, and the tree node values are available from the training.

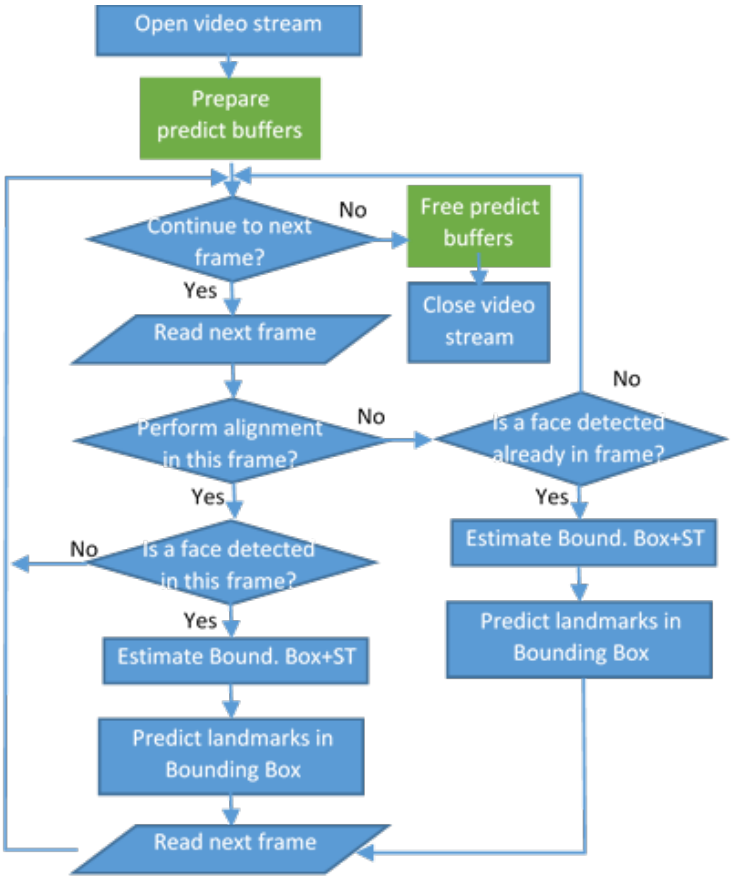

Figure 2: *DEST Application for face shape alignment in video frames*

[2] The *predict()* function is implemented in the DEST library and performs landmark position estimation.

The most computational intensive operation that is the *predict()* function as seen in Figure 3. The top level *Tracker::predict()* accepts as input the image frame and the position of the bounding box of the face that has been recognized and returns the estimated face landmark positions. A loop in this function executes the cascade stages. Within each cascade stage, the *Regressor::predict()* is called. The *Regressor::predict()* accepts as input the image pixel matrix Img, the face bounding box coordinates and the current shape estimation. In the *Regressor::predict()* function the pixel intensities of a sparse image representation are read and the *Tree::predict()* is called for all the stored binary regressor trees. In each tree node that is visited, the intensities of a pair of pixels indexed in the trained model are compared. The right or left direction of the binary tree is followed depending on whether the intensity difference is larger than a threshold that is also stored in the trained model.

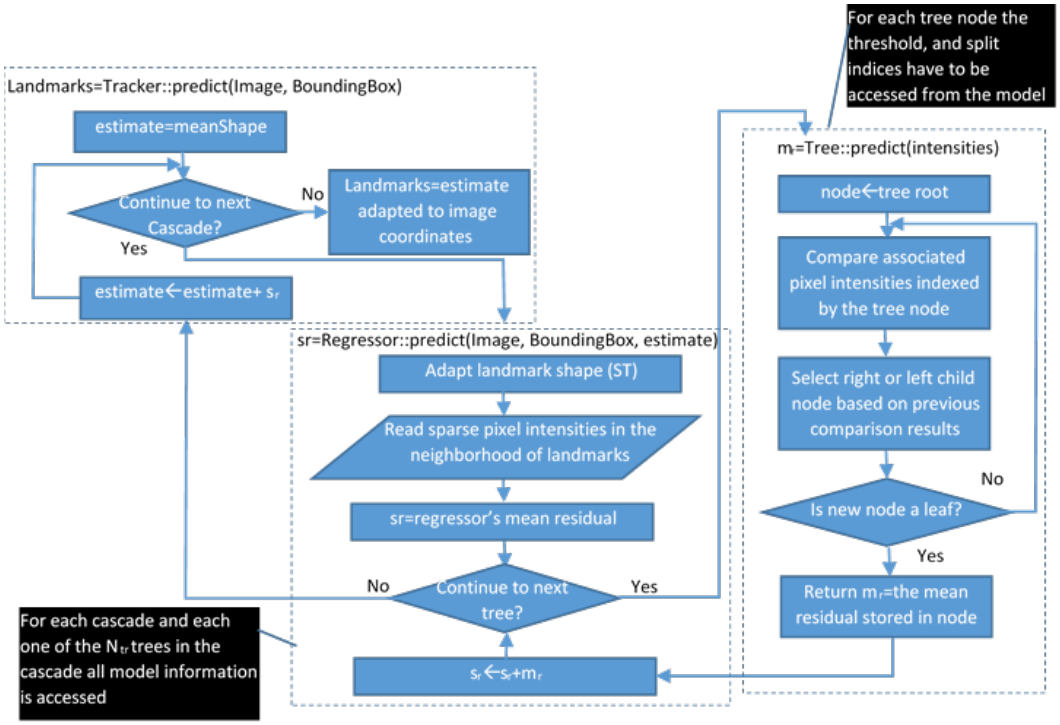

Figure 3: *The operations of the predict() routine that performs the landmark position estimation*

In our implementation, the functionality of all the three nested *predict()* routines shown in Figure 3, was initially included in flat Kernel_*predict()* routine that has been developed in ANSI C, in order to be portable to hardware. All the numerous parameters of the trained model that were accessed in the original DEST implementation from *predict()* routines, are now loaded during initialization into contiguous buffers from the new *predict_prepare()* routine, seen in the corresponding green box in Figure 2.

The initial implementation in hardware of the complete functionality of the *predict_kernel()* routine resulted in a kernel that required 16 large buffers to be passed as arguments and transferred from the main DRAM memory. Complicated operations that were consuming a large

number of resources were only executed once for each *predict_kernel()* call. Inside these operations there were no iterations and other operations appropriate for parallelization that would benefit from hardware implementation.

For this reason, a profiling was performed in the internal operations of the *kernel_predict()* routine. It was found that 85% of the latency of this routine was spent in the iterations of the *Regressor:predict()* and Tree::predict() routines. Therefore, it was decided to implement the iteration that visits the binary regression trees and the nested one that visits the levels of these trees in hardware as a single hardware kernel named *predict_trees().* For more details, the reader is referred to the Deliverable D4.8 where more details can be found.

4.1.2. Employed HLS pragmas

Several profiling methods supported in the Xilinx Vitis environment have been used to assess the acceleration techniques that have been applied in the *predict_trees()* kernel. Three levels of profiling were employed:

- Min/max latency estimation in the Vitis/Vivado High Level Synthesis (HLS) tool,
- XRT profiling on the target ZCU102 board
- Profiling performed within the application code by printing the time intervals measured to complete a *predict_kernel* and a *predict_trees* routine. The *predict_kernel* latency corresponds to a single frame processing and the one of the *predict_trees* is the latency of a hardware kernel execution.

The following acceleration techniques were tested:

4.1.2.1. Local data processing

Accessing local BRAM is much faster than accessing the RAM that is common to the PS and PL of the FPGA. Copying the data needed by the PL from RAM to BRAM is a very efficient acceleration technique. However, this technique can be employed in applications where the size of the data used by the hardware kernel is small and the data are reused multiple times. In this way, the initial data copy does not take much time and then all the accesses are performed extremely fast.

In our DSM application the model is too large (approximately 50Mbytes) to fit in the local BRAM. The size of the data needed in each cascade stage is also too large to fit in the local BRAM. Even if the model data fitted in local BRAM it would be needed to repeatedly swap between the data of each cascade stage. The overhead required for this continuous data swapping would exceed the expected gain from the higher speed data processing.

Another problem with the use of this technique in the DSM application is that only log2(number of tree nodes) of the data stored in the pretrained model are used within the kernel but the values that are going to be used are not known before the kernel execution. This is due to the fact that

the parameters stored in the model concern all the nodes of the regression trees, but only the values of the nodes that are visited are used in the calculations performed within the kernel and these nodes are only 5 as the tree is traversed from the root to the leaf in a binary tree of 32 nodes. For these reasons, this acceleration method was not applicable.

4.1.2.2. Pipeline and Loop Unrolling

Loop unrolling was applied to the loops of the *Regressor::predict* and *Tree::predict* routines of Figure 3. The system was forced to use pipeline wherever applicable. The loop in the *Regressor::predict* routine traverses the trees of the current cascade stage and the loop in the Tree::predict traverses the nodes in each tree from the root to the leaf. After the loop in the *Tree::predict* routine ends, a second loop updates the correction factors.

Unrolling can be performed by several factors leading to implementations of different speed and resource requirements. The factors that maximized the speed with respect to the resource limitations of the specific FPGA of the ZCU102 target board lead to an overall predict_kernel latency of 36ms, i.e., a speed improvement by approximately 15%.

4.1.2.3. Kernel argument passing through wider ports

As already described, transferring large parts of the data model to local BRAM was not feasible. Although only about 15% of the parameters stored in the data models are used in the operations of the hardware kernel, if these values are transferred through wide ports a significant acceleration can be achieved. The model parameters are passed as arguments to the hardware kernel. The initial hardware implementation of the *predict_kernel()* routine, required 18 large buffers to be passed as arguments. In the implementation of the *predict_tree()* routine as hardware kernel the number of buffers that have to be passed as arguments was reduced to 5 but still it is not possible to store their values in local BRAM. However, transferring in BRAM only the data needed by the loop in the *Tree::predict()* routine, using a wide port (of 1024 bits) is feasible. This technique in conjunction with pipeline and loop unrolling allowed a further reduction of the *predict_tree()* kernel latency to 33ms (using the default ERT model).

By modifying the default ERT model parameters such as the cascade stages, number of trees per cascade stage, regression tree sized, or the number of reference pixels in the sparse image representation used, the resulting hardware kernels may have different resource requirements and different speeds. Using comparable ERT model parameters both in the software and the hardware implementations an acceleration in the order of 22% was achieved.

4.1.3. Demonstration/ Usage example/Validation

The full code of the *predict_tree()* HW kernel is the following:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ap_int.h>
#include "common_decl_fl.h"
```

```
//// THE FOLLOWING DEFINITIONS ARE REQUIRED FOR ARGUMENT PASSING THROUGH WIDE PORTS
////  KR_INTENSITIES  ////
#define NUM_OF_VALUES 600
#define WIDTH_PER_VALUE 32
#define DATA_PER_PKT 30
#define DATAWIDTH 960
#define NUM_OF_PKT (NUM_OF_VALUES/DATA_PER_PKT)
typedef ap_int<DATAWIDTH> u_intensities_t;

////  KR_SPLIT1/2  ////
#define SPLIT_NUM_OF_VALUES 16000
#define SPLIT_WIDTH_PER_VALUE 16
#define SPLIT_DATA_PER_PKT 64
#define SPLIT_DATAWIDTH 1024
#define SPLIT_NUM_OF_PKT (SPLIT_NUM_OF_VALUES/SPLIT_DATA_PER_PKT)
typedef ap_int<SPLIT_DATAWIDTH> u_split_t;

////  KR_NODE_THRES  ////
#define THRES_NUM_OF_VALUES 16000
#define THRES_WIDTH_PER_VALUE 32
#define THRES_DATA_PER_PKT 32
#define THRES_DATAWIDTH 1024
#define THRES_NUM_OF_PKT (THRES_NUM_OF_VALUES/THRES_DATA_PER_PKT)
typedef ap_int<THRES_DATAWIDTH> u_thres_t;

#define learningRates 0.15

extern "C" {

        void predict_trees(
                float*                  kr_sr_tg,
                int                             kr_tg_len,
                int*                    kr_casc,
                u_split_t*              kr_split1,
                u_split_t*              kr_split2,
                u_intensities_t* kr_intensities,
                u_thres_t*              kr_node_thres,
                float*                  kr_node_mean)
                //float*                kr_learningRates)
        {
                #pragma HLS INTERFACE m_axi latency=1 depth=39168 bundle=gmem_node_mean
port=kr_node_mean
                #pragma HLS INTERFACE m_axi latency=1 depth=136   bundle=gmem_sr_tg port=kr_sr_tg
                #pragma HLS INTERFACE m_axi latency=1 depth=16000 bundle=gmem_split1
port=kr_split1
                #pragma HLS INTERFACE m_axi latency=1 depth=16000 bundle=gmem_split2
port=kr_split2
                #pragma HLS INTERFACE m_axi latency=1 depth=1             bundle=gmem_casc
port=kr_casc
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_node_mean
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_node_thres
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_sr_tg
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_split1
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_split2
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_intensities
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_kr_tg_len
                #pragma HLS INTERFACE s_axilite bundle=control port=kr_casc

                int indx, k;
                int numCasc = kr_casc[0];

                int split_index=numCasc*SPLIT_NUM_OF_PKT;
                int thres_index=numCasc*THRES_NUM_OF_PKT;

                // PREPARATION OF BRAM BUFFERS TO STORE PARAMETERS TRANSFERRED THROUGH WIDE PORTS
                float kr_intensities_bram[NUM_OF_VALUES];
                #pragma HLS array_partition variable=kr_intensities_bram complete dim=0
                #pragma HLS resource variable=kr_intensities_bram core=RAM_2P_LUTRAM
```

```
		// USE OF WIDE PORT WITH LOOP UNROLLING/PIPELINE
		loop_kr_intensities_pkt:for(int i=0; i<NUM_OF_PKT; i++)
		{
			#pragma HLS loop_tripcount min=20 max=20
			#pragma HLS pipeline II=1
			loop_data_per_pkt:for(int k=0; k<DATA_PER_PKT; k++)
			{
			kr_intensities_bram[i*(NUM_OF_PKT)+k]          =          kr_intensities-
>range((k+1)*WIDTH_PER_VALUE-1, k*WIDTH_PER_VALUE);
			}
		}

		// PREPARATION OF BRAM BUFFERS TO STORE PARAMETERS TRANSFERRED THROUGH WIDE PORTS
		int kr_split1_bram[SPLIT_NUM_OF_VALUES];
		#pragma HLS array_partition variable=kr_split1_bram complete dim=0
		#pragma HLS resource variable=kr_split1_bram core=RAM_2P_LUTRAM

		int kr_split2_bram[SPLIT_NUM_OF_VALUES];
		#pragma HLS array_partition variable=kr_split2_bram complete dim=0
		#pragma HLS resource variable=kr_split2_bram core=RAM_2P_LUTRAM

		// USE OF WIDE PORT WITH LOOP UNROLLING/PIPELINE
		loop_kr_split_pkt:for(int i=0; i<SPLIT_NUM_OF_PKT; i++)
		{
			#pragma HLS loop_tripcount min=250 max=250
			#pragma HLS pipeline II=1

			loop_split_data_per_pkt:for(int k=0; k<SPLIT_DATA_PER_PKT; k++)
			{
				kr_split1_bram[i*(SPLIT_NUM_OF_PKT)+k]                          =
(&kr_split1[split_index])->range((k+1)*SPLIT_WIDTH_PER_VALUE-1, k*SPLIT_WIDTH_PER_VALUE);
				kr_split2_bram[i*(SPLIT_NUM_OF_PKT)+k]                          =
(&kr_split2[split_index])->range((k+1)*SPLIT_WIDTH_PER_VALUE-1, k*SPLIT_WIDTH_PER_VALUE);
			}
			split_index++;
		}


		// PREPARATION OF BRAM BUFFERS TO STORE PARAMETERS TRANSFERRED THROUGH WIDE PORTS
		float kr_thres_bram[THRES_NUM_OF_VALUES];
		#pragma HLS array_partition variable=kr_thres_bram complete dim=0
		#pragma HLS resource variable=kr_thres_bram core=RAM_2P_LUTRAM

		// USE OF WIDE PORT WITH LOOP UNROLLING/PIPELINE
		loop_kr_node_thres_pkt:for(int i=0; i<THRES_NUM_OF_PKT; i++)
		{
			#pragma HLS loop_tripcount min=500 max=500
			#pragma HLS pipeline II=1

			loop_thres_data_per_pkt:for(int k=0; k<THRES_DATA_PER_PKT; k++)
			{
				kr_thres_bram[i*(THRES_NUM_OF_PKT)+k]                           =
(&kr_node_thres[thres_index])->range((k+1)*THRES_WIDTH_PER_VALUE-1, k*THRES_WIDTH_PER_VALUE);
			}
			thres_index++;
		}



		loop_num_trees: for(int j = 0; j < kr_tg_len; ++j)
		{
			// LOOP UNROLLING
			#pragma HLS loop_tripcount max=500 min=500
			#pragma HLS unroll factor=4

			int maxTests = 4;
			int n = 0;
```

```
                        num_trees_section:{
                                loop_max_tests: for (k = 0; k < maxTests; ++k)
                                {
                                        // LOOP UNROLLING
                                        #pragma HLS latency max=2 min=2
                                        #pragma HLS loop_tripcount max=4 min=4
                                        #pragma HLS unroll factor=4

                                        indx = numCasc*500*maxN+j*maxN+n;

                                        if (kr_split1[indx]<0)
                                        {
                                                break; // premature leaf
                                        }
                                        bool  left  =  kr_intensities_bram[kr_split1_bram[indx]]-
kr_intensities_bram[kr_split2_bram[indx]]> kr_thres_bram[indx];
                                        n = left ? 2 * n + 1 : 2 * n + 2;
                                }
                                indx = numCasc*500*maxN+j*maxN+n;
                                loop_num_LM: for (k=0;k<2*LM;k++)
                                {
                                        // LOOP UNROLLING AND PIPELINE
                                        //#pragma HLS latency max=2 min=2
                                        #pragma HLS loop_tripcount min=136 max=136
                                        #pragma HLS UNROLL factor=8
                                        #pragma HLS pipeline II=1
                                        kr_sr_tg[k]     +=     kr_node_mean[indx*(2*LM)+k]     *
learningRates;
                                }
                        }
                }
        }
}
```

The full source code of the Xilinx Vitis project that implements both the HW kernels and the SW application can be downloaded from the following link: https://git.esda-lab.gr/npetrellis/cpsosawaredsm-vitis

4.1.3.1. HW Kernels developed in the DSM application implemented in FPGA

The parameters of the ERT face alignment model are described in D4.8 in detail. The ones that affect the implementation of the HW kernels are listed in Table 1. From the description of the ERT parameters in this table it is clear that a trade-off has to be made between accuracy and speed. However, the convergence to a higher accuracy is much slower after these parameters reach a certain limit and in this case, it is not worth extending excessively the latency without a useful result.

Table 1: ERT parameters customized for the DSM application

Parameter Name	Default value	Experimentation in the range	Description
Cascade stages (Tcs)	10	9-12	Fewer cascade stages result in faster shape estimation but with lower accuracy. However error floor prevents accuracy improvement if excessive cascade stages are added

Trees (Ntr)	500	400-600	Fewer trees increase speed but reduce accuracy
Tree depth (Td)	5	4-6	Number of tree nodes is 2Td-1. Shorter tree depth (and consequently tree nodes) results in faster operation but with lower accuracy
Reference pixels (Nc)	600	400-800	This is the number of pixels that the sparse image consists of. Fewer reference pixels are expected to increase speed with accuracy penalty

4.1.3.2. Datasets

The effect of the parameter values in Table 1 also depend on the image dataset used. The iBug or Helen dataset that were used to train the models shipped with the DEST package contain general face images. However, training the model with these images was not optimal for driver face alignment since people displayed in the iBug and Helen datasets are not in the driver's seat position. Moreover, the lighting exposure of these photographs do not resemble with driving in nighttime conditions.

The developed DSM module has been tested initially with videos that did not display a driver. These videos were simple ones captured by the office web camera either in daytime or nighttime with good light exposure. However, the conditions in these videos did not resemble the driving environment in a real vehicle especially during nighttime.

There aren't many public datasets for driver drowsiness detection. A rich dataset with drivers yawning can be found in the YAWDD dataset (https://ieee-dataport.org/open-access/yawdd-yawning-detection-dataset). The video specs in this dataset are the following: video 640x480 24-bit true color (RGB) 30 fps AVI without audio. This dataset consists of several photographs and videos. The videos are split in two main categories:

1. Videos captured from camera placed on the dash of the car
2. Videos captured from a camera mounted on the internal mirror.

In the first group the camera points directly to the face of the driver while in the second group the camera points the face of the driver from an angle. Taking into consideration that the shape alignment followed is a 2D approach it is expected that videos taken from the dash of the car will achieve a higher accuracy.

The dash dataset consists of 16 videos with male drivers and 13 with female drivers. The mirror dataset consists of 47 male drivers and 43 female drivers[3]. Most of the videos last between 60 and 90 seconds. For each driver in the mirror dataset, 3-4 video types are available:

1. wearing or not wearing glasses
2. yawning

[3] Age, race and ethnicity of the drivers are not considered

3. talking.

The driver had three yawnings in the corresponding videos.

Figure 4: *The same female driver in the YawDD mirror dataset with shortsighted glasses (left) and sunglasses (right)*

Figure 5: *Various male drivers from the YawDD dash dataset that has been used in our experiments*

Although the YawDD dataset is quite extensive it has the following drawbacks:

- All the videos are captured in daytime, thus the darker environmental conditions needed to test a DSM module in nighttime drive cannot be tested
- The cars are stationary and the drivers pretend to drive so the conditions are not fully realistic
- Only yawning can be tested, the drivers do not close their eyes in a sleepy way, nor do they pretend to get a microsleep (a sleep for a few seconds) and the distraction can be partially tested (e.g., from the videos where the drivers talk).

For these reasons, although the YawDD dash videos have been used in our tests, UoP also developed its own dataset that is appropriate for testing DSM modules in nighttime conditions. Initially, UoP used 7 videos of 20-30 seconds duration with the same driver but in nighttime and

real drive in a highway as seen in Figure 6. The driver is yawning, periodically closes his eyes in a sleepy way and pretends to get a few microsleeps.

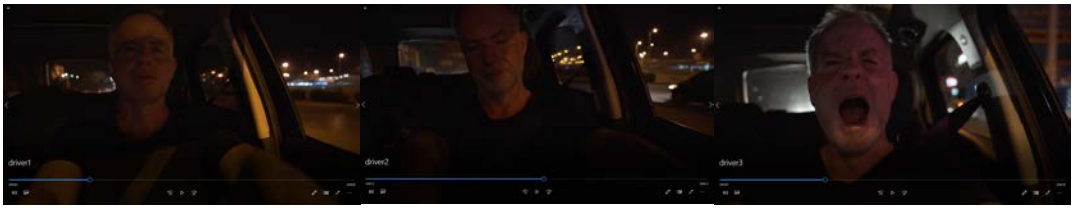

Figure 6: *Indicative frames from the 7 videos used initially to experiment with nighttime driver drowsiness detection*

A more ambitious dataset (NYSEM-Nighttime, Yawning, SleepyEye, Microsleep) for testing DSM modules in the nighttime with real driving conditions is also under development by UoP. It will consist of videos with more than 20 male and female drivers. For each driver 6 videos lasting 20-30 second will be available and in each video the driver will yawn 3 times. Moreover, for each driver 6 videos will also be made with the driver pretending to have microsleeps (this is a feature that cannot be found in any public dataset already available). For safety reasons these 6 videos have been captured stationary with the driver pretending to drive as is the case with YawDD videos. All of these videos are captured in the nighttime.

By the end of the project NYSEM dataset will be used in a complementary way with YawDD to test the DSM developed by UoP as well as other DSM modules developed as smartphone apps or with GPUs from Catalink and ISI as will be described in detail in WP6. Moreover, NYSEM will also be used in the future to extract frames that will be used to train new ERT face alignment models specialized for nighttime drowsiness detection.

4.1.3.3. ERT models developed

Various models have been trained with the same Helen general purpose dataset. Table 2 shows the combinations tested and the training error exposed by the DEST training application that has also been ported to Ubuntu. A different application is used to evaluate a model using a test set of 300 photos, different from the Helen dataset. The details of these models are discussed in D4.8.

Models M15 and M16 have been defined as combinations of various parameter values for the best accuracy and the highest speed, respectively. Since we did not have a dataset available with driver images and especially in nighttime lighting conditions, we defined three models called Dark0.3, Dark0.4 and Dark0.5 that have been trained from the Helen images again but after artificially darkening them during the training process.

The ERT parameters with the default values were used but a dynamic range adaptation has been applied in grayscale to compress the pixel values to the 30%, 40% or 50% of their original range in the models Dark0.3, Dark0.4 and Dark0.5, respectively. For example, if the pixel intensity is initially between 0 and 100, this intensity is linearly adapted to shorten the range between 0 and 30 in the Dark0.3 model. It is obvious that Dark0.3 has been trained with darker images while

Dark0.5 is closer to the M0 model. The test error is worse than the other models. However, in a real car environment and nighttime drive it is expected that the behavior of the Dark0.* models may be better than several other models listed in Table 2.

Table 2: ERT models used, based on different ERT parameters

Model No.	Cascade stages	Trees	Tree Depth	Ref. Pixels	Splits	λ	lf	shapes	Training Error	TestErr Average	Test Err Worst
M0 (default)	10	500	5	600	20	0.1	0.15	20	0.016	0.03887	0.74937
M1	10	500	5	400	20	0.1	0.15	20	0.017	0.03949	0.73315
M2	10	500	5	600	20	0.1	0.08	20	0.021	0.03893	0.71978
M3	10	500	5	600	20	0.3	0.15	20	0.016	0.03937	0.70967
M4	10	500	5	600	16	0.1	0.15	20	0.017	0.03845	0.70771
M5	10	500	5	600	20	0.1	0.15	10	0.014	0.03955	0.73077
M6	12	500	5	600	20	0.1	0.15	20	0.015	0.03834	0.75377
M7	10	500	5	600	20	0.1	0.20	20	0.014	0.03928	0.73658
M8	10	500	5	800	20	0.1	0.15	20	0.016	0.03805	0.69967
M9	10	500	4	600	20	0.1	0.15	20	0.022	0.03948	0.69410
M10	10	500	6	600	20	0.1	0.15	20	0.011	0.26439	1.43381
M11	10	400	5	600	20	0.1	0.15	20	0.018	0.03896	0.73354
M12	10	600	5	600	20	0.1	0.15	20	0.015	0.03806	0.73958
M13	10	500	5	600	24	0.1	0.15	20	0.016	0.03867	0.73369
M14	10	500	5	600	20	0.1	0.15	30	0.017	0.03863	0.71161
M15-Accurate	12	600	5	800	20	0.1	0.15	20	0.013	0.03790	0.70010
M16-Fast	9	400	4	400	16	0.1	0.15	10	0.023	0.04165	0.72510
Dark0.3	10	500	5	600	20	0.1	0.15	20	0.016	0.09033	0.91768
Dark0.4	10	500	5	600	20	0.1	0.15	20	0.015	0.05882	0.75510
Dark0.5	10	500	5	600	20	0.1	0.15	20	0.015	0.04886	0.74717

4.1.3.4. Accuracy of the models

Exhaustive experiments are described in D4.8 concerning the accuracy of the models listed in Table 3 under different environmental conditions. Table 3 lists the conclusions about which models are more appropriate for certain combinations of driver gender/lighting conditions/mount position of the camera. The F1-score has been used to sort the model accuracy since this metric is a combination of precision and sensitivity. The male and female drivers in daytime conditions have been evaluated with the YawDD dataset (camera mounted on the mirror or the dash). The nighttime is evaluated for the present, only with the 7 videos that we have developed in UoP. More exhaustive tests will be performed when the NYSEM dataset will be available.

Table 3: Top-3 models with the highest accuracy in yawning measurement.

Condition	Model
Male-Daytime-Dash	M16, M8, M15
Male-Daytime-Mirror	Dark0.3, M8, Dark0.5
Female-Daytime-Dash	M12, M15, M16
Female-Daytime-Mirror	Dark0.3, M8, Dark0.5
Male-Nighttime (from Table 5)	M8, M16, M15

4.1.3.5. Speed, power consumption and resources of the models tested in Vitis-ZCU102

Among the various models that have been trained in the Ubuntu environment and were listed in Table 1, we focused on the models listed in Table 3 since they achieved the highest accuracy.

Table 4: ERT parameter dependence of the developed hardware kernels

ERT Parameter	Affects Old predict_kernel() implementation	Affects New predict_tree() implementation	Notes
Cascade stages (Tcs)	Yes	No	In the predict_kernel() implementation the cascade loop was implemented inside the HW kernel. In the new predict_tree(), it is implemented in the SW level outside the hardware kernel.
Trees (Ntr)	Yes	Yes	The HW kernel latency is proportional to the number of trees examined in each cascade stage
Tree depth (Td)	Yes	Yes	The regressor tree nodes is 2Td but only Td nodes are visited. The HW kernel latency is proportional to Td
Reference pixels (Nc)	Yes	Yes	The larger the number of reference pixels, the longer time is needed to transfer their intensities to the kernel. However, the number of reference pixels examined depends on the regressor trees and their depth instead of the Nc. This is due to the fact that the pair of reference pixel intensities is determined in each regressor tree node that is visited.
Splits	No	No	
λ	No	No	
lf	No	No	
Shapes	No	No	

The variations in the ERT parameter values between the models listed in Table 1 affect the implementations of the old hardware kernels predict_kernel() and the new one: predict_tree(). Table 4 shows the dependencies between these kernels and the ERT parameters. Based on these tables it can be deducted that the models M0, M2-M5, M7, Dark0.3, Dark0.4, Dark0.5 may all use the same HW kernel with 10 cascade stages, 500 regressor trees / cascade stage, tree depth equal to 5 and 600 reference pixels. The HW kernels that support the rest of the models can be customized and require different resources and power, while the show different latency. The resources required by the models that achieved the highest accuracy in Table 3 are shown in Table 5. As expected, the lower resources are required by the hardware kernel for M16 and the higher resource are required by M15.

Table 5: FPGA resources needed by the HW kernels of the models referenced in Table 3

Model/ Resources	M0, M4, Dark0.3, Dark0.4, Dark0.5	M8	M12	M15	M16
LUT	90371 (33%)	90631 (33%)	86639 (32%)	94241 (34%)	69976 (26%)
LUTRAM	21860 (15%)	22020 (15%)	19040 (13%)	25212 (18%)	7802 (5%)
FF	123228 (22%)	123268 (22%)	119299 (22%)	125173 (23%)	111104 (20%)
BRAM	284 (31%)	284 (31%)	284 (31%)	284 (31%)	275 (30%)
DSP	88 (3%)	88 (3%)	90 (4%)	90 (4%)	91 (4%)
BUFG	15 (4%)	15 (4%)	11 (3%)	15 (4%)	7 (2%)
MMCM	1 (25%)	1 (25%)	1 (25%)	1 (25%)	1 (25%)

The estimated power consumption of the models referenced in Table 3 is analyzed in Table 6 which shows the dynamic power consumption and in Table 7. Since we are interested in the power consumption of the kernels that are implemented in the PL system we get the total power dissipation, static and dynamic, by adding all the power estimations for the PL part and the results are listed in Table 8. As we can see the power consumption ranges between 2.665W (M16) and 2.905W (M15).

Regarding the latency of each HW kernel implementation and its effect on the overall processing time needed by a single frame, the results are listed in Table 9. In the first row of Table 9 the latency of a single frame is shown, as it is measured from the software side with appropriate print messages, as seen in Figure 7. In the second row the HW kernel latency is listed as it is profiled using the XRT real time monitoring facilities that generate a runtime csv file (profile_summary.csv). Since it was not possible to migrate data pixel intensities using wide port and local BRAM in the same way we implemented migrate of the other HW kernel arguments, its latency was measured separately as shown in the last row of Table 9.

Table 6: Dynamic power consumption of the HW kernels of the models referenced in Table 3.

Model/ Power	M0, M4, Dark0.3, Dark0.4, Dark0.5	M8	M12	M15	M16
Clocks	0.496W (10%)	0.498W (10%)	0.515W (11%)	0.515W (10%)	0.461W (10%)
Signals	0.488W (10%)	0.510W (11%)	0.540W (11%)	0.561W (11%)	0.454W (10%)
Logic	0.369W (8%)	0.374W (8%)	0.399W (8%)	0.401W (8%)	0.350W (7%)
BRAM	0.585W (12%)	0.584W (12%)	0.586W (12%)	0.604W (12%)	0.576W (12%)
DSP	0.088W (2%)	0.089W (2%)	0.092W (2%)	0.088W (2%)	0.089W (2%)

MMCM	0.097W (2%)	0.097W (2%)	0.097W (2%)	0.097W (2%)	0.097W (2%)
PS	2.659W (56%)	2.659W (55%)	2.659W (54%)	2.659W (55%)	2.659W (57%)

Table 7: Static power consumption of the HW kernels of the models referenced in Table 3

Model/ Power	M0, M4, Dark0.3, Dark0.4, Dark0.5	M8	M12	M15	M16
PL	0.638W (86%)	0.638W (86%)	0.639W (86%)	0.639W (86%)	0.638W (86%)
PS	0.100W (14%)	0.100W (14%)	0.100W (14%)	0.100W (14%)	0.100W (14%)

Table 8: Total power dissipation of the PL part

Model/ Power	M0, M4, Dark0.3, Dark0.4, Dark0.5	M8	M12	M15	M16
PL	2.761W	2.79W	2.868W	2.905W	2.665W

Table 9: Total power dissipation of the PL part

Model/Latencies	M0 model	M4 Model	M8 Model	M12 Model	M15 Model	M16 Model	Dark0.3 Model	Dark0.4 Model	Dark0.5 Model	Notes
Frame processing time	33.166 ms	34.301 ms	34.658 ms	38.629 ms	47.756 ms	29.314 ms	33.816 ms	37.717 ms	33.371 ms	1st Frame
HW kernel Predict_Trees (XRT profile_summary.csv)	2.33 ms	2.42 ms	2.37 ms	2.86 ms	2.90 ms	2.09 ms	2.26 ms	2.26 ms	2.40 ms	Worst Case (1st frame)
Migrate pixel intensities (XRT profile_summary.csv)	0.076 ms	0.149 ms	0.160 ms	0.151 ms	0.153 ms	0.145 ms	0.149 ms	0.149 ms	0.114 ms	Worst Case (1st Frame)
Maximum supported frame rate	30 fps	29 fps	28 fps	25 fps	20 fps	34 fps	29 fps	26 fps	29 fps	

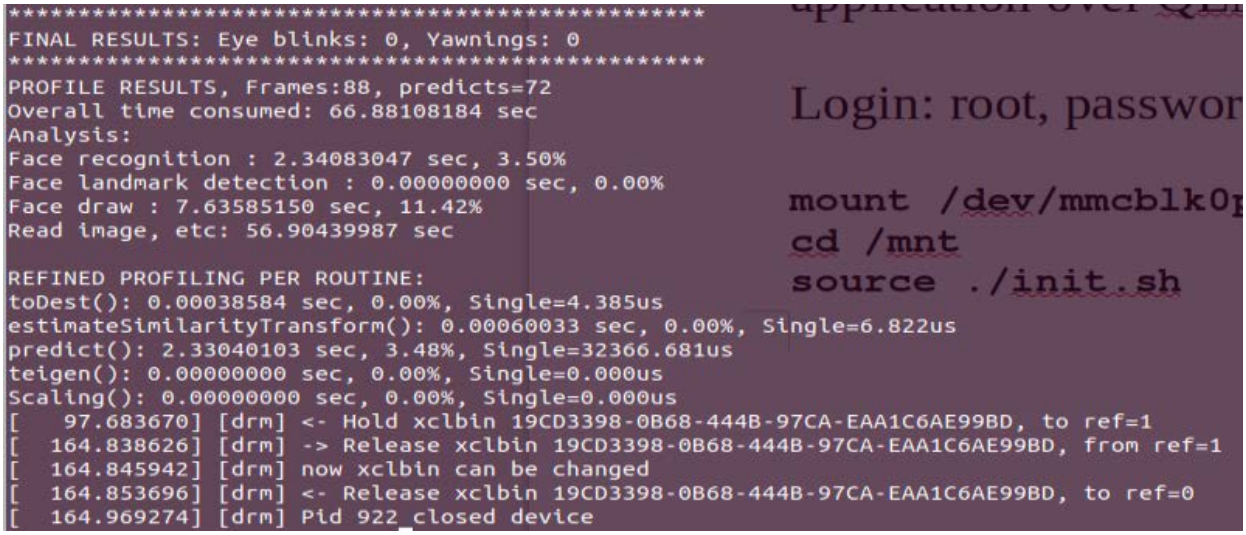

Figure 7: *Estimation of the single frame processing latency (predict()-single) using appropriate messages printed in the application software level.*

4.2. Security pillar cryptography runnable

4.2.1. Description

As algorithmic complexity is gradually increasing, hardware designers and implementers need advanced tools to assist them in designing complex algorithm hardware implementations. HLS tools, as discussed above, aim to automate the hardware implementation process by generating the actual RTL implementation from user-provided C, C++, System-C, or OpenCL input code along with a series of configuration directives (i.e., pragmas).

However, the performance of the generated design in terms of computational speed and required resources is related to the appropriate setup of the HLS pragmas. The inference of required pragmas to produce the right interface for each code function arguments and to pipeline loops and functions within the input code is not a straightforward process since familiarity with the algorithmic flow, the target hardware/Programmable Logic resources (e.g., FPGA LUTs, BRAMS, DSPs) as well as state-of-the art hardware design and optimization techniques (e.g., loop unrolling, pipelining and parallel processing etc.) are needed.

In the security and cryptography research domain, one of the latest trends, given the evolution of quantum computing, is the research and development of postquantum cryptography algorithms, i.e., algorithms that will withstand the quantum computer crypto-analytic capacity. Most of such algorithms rely on Lattice-based computing, i.e., Lattice-based cryptography (LBC).

However, as many cryptography solutions, LBC has considerable computational complexity since it relies on matrix/lattice operations over finite field polynomial representations. In order to implement such algorithms in hardware (or hardware programmable logic/FPGAs) hardware engineers need to fully understand the algorithm's inner workings. The complexity of such a task

indicates that the HLS tool design path can be a good alternative to traditional RTL design. Several researchers have shown that such an approach does have merit and can lead to good results so long as the right pragmas are used.

In the latest round of National Institute of Standards and Technology (NIST) contest relative to cryptography several Postquantum Cryptography (PQC) algorithms are proposed that could be robust against attacks from Quantum computers in the future. Candidates are using the Number Theoretic Transform (NTT) multiplication, in order to decrease the multiplication complexity between two polynomials from O(N) to O(n*log(n)). Multiplication is the main bottleneck of these algorithms, that's why optimization and further development of the existing proposed implementations is so crucial.

Our contribution relies into optimizing the memory access scheme of NTT and Inverse-NTT (INNT), in order to have multiple parallel reads/writes. This happens by minimizing the memory-dependency between NTT/INTT processing elements and increase parallel memory-access of these elements by forcing the HLS tool to bypass the dependence, as long as our memory scheme allows it.

4.2.1.1. NTT/INTT basic algorithm

In this section we review the NTT which corresponds to a Fast Fourier Transform (FFT) where the roots of unity are taken from a finite ring instead of the complex numbers. The n-point FFT with $n = 2^k$ is an efficient method to evaluate a polynomial

$$a(x) = \sum_{j=0}^{n-1} a_j * x^j \ e \ Z[x]$$

in the n-th roots of unity w_n^i for $i = 0, \dots, n-1$, where w_n denotes a primitive n-th root of unity. More precisely, on input we do have the coefficients $a\_0, .., a\_(n-1)$ and w_n, the FFT computes:

$$FFT(a_j, w_n) = [a_j * (w_n^0), a_j * (w_n^1), .., a_j * (w_n^{n-1})]$$

in O(n*logn) time. Due to the orthogonality relations between the n-th roots of unity, we can compute the inverse-FFT simply as the NTT replaces the complex roots of unity by roots of unity in a finite ring Z_q (twiddle factors). Since we require elements of order n, q is chosen to be a prime with $q = 1 \ mod \ 2n$.

The basic algorithm which relies on the Cooley-Tukey (CT) approach can be seen below in Algorithm 1.

Algorithm 1 In-place forward radix-2 Cooley-Tukey NTT

Input a polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$,
Input n-th primitive root of unity $\omega_n \in \mathbb{Z}_q$
Input $n = 2^l$
Output $\hat{a}$ = NTT(a)

$\hat{a} \leftarrow$ bit-reverse(a)
for $(i = 1 : i < l : i++)$ **do**
  $m \leftarrow 2^{l-i}$
  $\omega_m \leftarrow \omega_n^{n/m}$
  for $(j = 0 : j < n : j++)$ **do**
    $\omega \leftarrow 1$
    for $(k = 0 : k < m/2 : m++)$ **do**
      $t1 \leftarrow \omega \cdot \hat{a}[k + j + m/2] \bmod q$
      $t2 \leftarrow \hat{a}[k + j]$
      $\hat{a}[k + j] \leftarrow t1 + t2 \bmod q$
      $\hat{a}[k + j + m/2] \leftarrow t1 - t2 \bmod q$
      $\omega \leftarrow \omega \cdot \omega_m \bmod q$
    end for
  end for
end for
return $\hat{a}(x)$

The input of the algorithm is a polynomial $a(x)$ in the time domain and the output is the polynomial $\hat{a}(x)$ in the spectral domain.

The same applies to the INTT algorithm, which is based on CT approach too. The algorithm takes as input the $\hat{a}(x)$ from the spectral domain and outputs the inverse NTT polynomial a(x) to the time domain. The INTT algorithm can be seen below on Algorithm 2.

Algorithm 2 In-place forward radix-2 Cooley-Tukey INTT

Input $\hat{a}$ polynomial $\hat{a}(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
Input n-th primitive root of unity $\omega_n \in \mathbb{Z}_q$
Input $n = 2^l$
Output a = NTT($\hat{a}$)

1: $a \leftarrow \hat{a}$
2: **for** $(i = l : i >= 1 : i - -)$ **do**
3: $\quad m \leftarrow 2^{l-i}$
4: $\quad \omega_m \leftarrow \omega_n^{n/m}$
5: $\quad$ **for** $(j = 0 : j < n : j + +)$ **do**
6: $\quad\quad \omega \leftarrow 1/n$
7: $\quad\quad$ **for** $(k = 0 : k < m/2 : m + +)$ **do**
8: $\quad\quad\quad t1 \leftarrow a[k + j]$
9: $\quad\quad\quad t2 \leftarrow a[k + j + m/2]$
10: $\quad\quad\quad a[k + j] \leftarrow t1 + t2 \bmod q$
11: $\quad\quad\quad a[k + j + m/2] \leftarrow \omega \cdot [t1 - t2] \bmod q$
12: $\quad\quad\quad \omega \leftarrow \omega \cdot \omega_m \bmod q$
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: **end for**
16: **return** $a(x)$

Both NTT and INTT immediately leads to a fast multiplication algorithm in the ring, given two polynomials a, b. As such, we can easily compute their product $c = a * b$, by computing the following:

$$c = NTT_{w\_n}^{-1}\left(NTT_{w_n}(a) * NTT_{w_n}(b)\right),$$

where * denotes point-wise multiplication.

4.2.2. Deployed HLS pragmas

In order to bypass the dependencies of the inner loop and optimize the algorithm in terms of area, by reusability, time using pipeline and parallel execution, there is a need to introduce some directives for the compiler, called pragmas, which indicate to the compiler what optimization needs to be applied.

Our deployment is an Iterative n-point NTT Algorithm architecture that uses the Cooley-Tukey butterfly approach, which consists of $\log_2(n)$ stages, where Processing Elements (PE) need to be executed on each stage. The input to the algorithm consists of the array of n coefficients â, the array of n pre-computed twiddle factors (based on n-th root of unity ω_n) and B which is the number of parallel executed PEs on each stage. The coefficients â are updated on each stage, based on each PE's execution, as can be seen in lines 22–24 of Algorithm 3. Relative to the

modular reduction of line 22, we are applying the Montgomery reduction algorithm. The HLS-friendly CT algorithm can be seen in Algorithm 3 and our implementation used B equal to 2.

Algorithm 3 HLS-friendly algorithm

Input a polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
Input $n = 2^l$, number of Parallel PEs B
Input $\omega[N]$, where $\omega[i] = \omega^i$
Output $\hat{a}$ = NTT(a)

1: $\hat{a} \leftarrow$ bit-reverse(a)
2: STAGE LOOP:
3: **for** $(i = 0 : i < l : i++)$ **do**
4: BUTTERFLY LOOP:
5: **for** $(s = 0 : s < N/2 : s = s + b)$ **do**
6: IDX_LOOP:
7: **for** $(b = 0 : b < B : b++)$ **do**
8: $j[b] \leftarrow (s + b) \gg (l - 1 - i)$
9: $k[b] \leftarrow (s + b)\&((u \gg i) - 1)$
10: $i_e[b] \leftarrow j[b] \cdot (1 \ll (l - u)) + k[b]$
11: $i_o[b] \leftarrow i_e[b] + (1 \ll (l - i - 1))$
12: $i_w[b] \leftarrow (1 \ll i) + ((s + b) \gg (l - i - 1))$
13: **end for**
14: MEM_READ_LOOP:
15: **for** $(b = 0 : b < B : b++)$ **do**
16: $U[b] \leftarrow \hat{a}[i_e[b]]$
17: $V[b] \leftarrow \hat{a}[i_o[b]]$
18: $W[b] \leftarrow \omega[i_w[b]]$
19: **end for**
20: OP_LOOP:
21: **for** $(b = 0 : b < B : b++)$ **do**
22: $t \leftarrow W[b] \cdot V[b] \bmod q$
23: $E[b] \leftarrow U[b] - t$
24: $O[b] \leftarrow U[b] + t$
25: **end for**
26: MEM_WRITE_LOOP:
27: **for** $(b = 0 : b < B : b++)$ **do**
28: $\hat{a}[i_e[b]] \leftarrow E[b]$
29: $\hat{a}[i_o[b]] \leftarrow O[b]$
30: **end for**
31: **end for**
32: **end for**
33: **return** $\hat{a}(x)$

As a starting point of our design and analysis, we adopted the code-architecture of a literature implementation where memory-indexing on each PE execution is not dependent of the two outer

loops, as in Algorithm 1, and parallel PE calculation is applied. This will be the base architecture of our proposed design. The first inner loop, i.e., the PE-execution loop, of Algorithm 1 is divided into four parts. Each PE calculates the corresponding memory-access indices (IDX LOOP), then the â and ω values are loaded from memory (MEM READ LOOP), the CT butterfly operations and Montgomery reduction (OP LOOP) are applied and results are written back to memory (MEM WRITE LOOP). This helps the synthesizer to find a better solution, as the code is more structured. The restructured code can be seen in Algorithm 3 above. The HLS pragmas that were used in the HLS tools can be seen in Table 10 below.

Table 10. Initial Version HLS Pragmas

Code	HLS-Pragma
j, k, i_e, i_o, i_w, w	array_partition complete
U, V, W, E, O	array_partition complete
$\hat{a}$	array_partition block factor=8
BUTTERFLY_LOOP	pipeline
IDX/READ/OP/WRITE_LOOP	unroll

After synthesis, based on reports produced by the HLS synthesizer, we realized that the HLS tool using the Algorithm 3 code could not resolve the memory-dependency between MEM READ LOOP and MEM WRITE LOOP, which is a write-after-read dependency. This highlights the necessity for a code modification that will result in a full independence between the two mentioned loops.

In order to resolve the memory-access dependency between read and write operations, we adopted a technique which is used in Fast Fourier Transform (FFT), and adapted it in our NTT domain. This path required two Dual Port RAMs and was applied for B = 2, as each PE required two parallel memory-accesses concurrently.

The proposed optimized restructured code using the above technique is presented in Algorithm 4 below. There are several differences between Algorithm 4 and Algorithm 3. The â variable corresponding to memory units in the FPGA implementation, is now split into two variables one for odd-parity and one for even. This will lead the HLS tool to produce two different memories for â that can be accessed through two dual port RAMs

Algorithm 4 Optimized Memory-Access HLS-friendly algorithm for 2 RAMs and $B = 2$

Input a polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
Input $n = 2^l$, number of Parallel PEs $B = 2$
Input $\omega[N]$, where $\omega[i] = \omega^i$
Output $\hat{a}$ = NTT(a)

$\hat{a} \leftarrow$ bit-reverse(a)
STAGE LOOP:
for $(i = 0 : i < l : i++)$ **do**
  BUTTERFLY LOOP:
  for $(s = 0 : s < N/2 : s = s + b)$ **do**
    IDX_LOOP:
    for $(b = 0 : b < B : b++)$ **do**
      $i_e[b] \leftarrow$ rotateLeft$(((s + b) \ll 1) + 0, i)$
      $i_o[b] \leftarrow$ rotateLeft$(((s + b) \ll 1) + 1, i)$
      $i_w[b] \leftarrow (1 \ll i) + ((s + b) \gg (l - i - 1))$
      $parity[b] \leftarrow calc\_parity(i_e[b])$
    end for
    MEM_READ_LOOP:
    for $(b = 0 : b < B : b++)$ **do**
      $U[b] \leftarrow \hat{a}[i_e[b] \gg 1][not(parity[b])]$
      $V[b] \leftarrow \hat{a}[i_o[b] \gg 1][parity[b]]$
      $W[b] \leftarrow \omega[i_w[b]]$
    end for
    OP_LOOP:
    for $(b = 0 : b < B : b++)$ **do**
      $t \leftarrow W[b] \cdot V[b] \bmod q$
      $E[b] \leftarrow U[b] - t$
      $O[b] \leftarrow U[b] + t$
    end for
    MEM_WRITE_LOOP:
    for $(b = 0 : b < B : b++)$ **do**
      $\hat{a}[i_e[b] \gg 1][not(parity)] \leftarrow E[b]$
      $\hat{a}[i_o[b] \gg 1][parity] \leftarrow O[b]$
    end for
  end for
end for
return $\hat{a}(x)$

In Algorithm 4, the synthesizer achieved a 20% decrease in latency and drastically reduced resources usage, as seen in Table 11.

Table 11. Results and comparison for n=256, 23-bit Montgomery Modulo Reduction

Version	Latency (Clock Cycles)	LUT	FF	BRAM	DSP	B
OpenCL	4382	36785	30834	9	0	2
Alg3	3987	4596	3825	39	14	2
Alg4	3218	2661	1735	7	12	2
HLS	5123	979	-	6	6	1
HLS	3075	12565	-	48	32	8

Using only this technique however, it can be observed that the dependency between the MEM READ LOOP and MEM WRITE LOOP is not fully resolved by the HLS tool. To solve the issue, the DEPENDENCE pragma has to be used. The DEPENDENCE pragma is used to provide additional information to the HLS synthesizer in order to overcome loop-carry dependencies and allow loops to be pipelined, or pipelined with lower intervals, for a specific variable. There are two options available, intra and inter which specifies whether the dependence is within the same loop iteration or in different loop iteration, respectively. Also, there are three directions available, Read-After-Write (RAW), Write-After-Read (WAR) and Write-After-Write (WAW).

Using our proposed memory-access scheme from FFT, it can be assured that there are no loop-carry dependencies or same index between inner-loops, so the DEPENDENCE pragma with inter false can be freely applied to Algorithm 4, which we call Algorithm 4$*$ to distinguish the usage of different pragmas. More specifically, this pragma is applied to â with inter option inside the BUTTERFLY LOOP, between line 4 and 5, to solve this dependency. As a result, a 50% decrease in latency was achieved compared to the original Algorithm 4 with a negligible increase in resources.

Table 12. Optimized Memory-Access HLS Pragmas

Code	**HLS-Pragma**
j, k, i_e, i_o, i_w, w	array_partition complete
U, V, W, E, O	array_partition complete
$\hat{a}$	array_partition block factor=8
BUTTERFLY_LOOP	pipeline
IDX/READ/OP/WRITE_LOOP	unroll
BUTTERFLY_LOOP: variable $\hat{a}$	dependence inter false

Our proposed optimized algorithm 4* has as an addition the false data-dependency pragma which was applied on variable â, as can be seen on Table 12. Leading to further reduction of area and latency of the design, displayed on Table 13. The same HLS architecture could be applied for INTT with minor differences, based on what we already discussed

Table 13. Results and comparison between Algorithm 4 and Algorithm 4*

Version	Latency (Clock Cycles)	LUT	FF	BRAM	DSP	B
Alg4	3218	2661	1735	7	12	2
Alg4*	1685	2590	1885	7	12	2

4.2.3. Demonstration, usage example and validation

Our demo utilizes the NTT and Inverse-NTT modules to create an NTT-multiplier. Codes can be found in https://gitlab.com/elkady.alexander/ntt/-/blob/main/codes/vitis_files/ntt_multiplyer/kernels/myNTT_multiplyer.cpp .
The top level function of our kernel can be seen bellow:

```
void NTT_multiplyer(unsigned int* a){
	unsigned int localRegs_a[N] ; unsigned int localRegs_b[N] ; unsigned int localRegs_c[N] ;
	#pragma HLS array_partition variable=localRegs_a type=block factor=2
	#pragma HLS array_partition variable=localRegs_b type=block factor=2
	#pragma HLS array_partition variable=localRegs_c type=block factor=2

	//Write from input to memory
	ReadInput_a_b: for (int i = 0; i<N; i++){
	#pragma HLS unroll factor=2
		localRegs_a[i] = a[i];
		localRegs_b[i] = a[i+N];
//		printf("%d: %d * %d\n",i,localRegs_a[i],localRegs_b[i]);
	}
	//Execute NTT for a and b vectors
	NTTA:NTT_256_hls(localRegs_a);
	NTTB:NTT_256_hls(localRegs_b);
	//dot multiplication
	dotMultiplication: for(int i=0;i<N;i++){
		#pragma HLS unroll factor=2
		localRegs_c[i] = montgomery_reduce((long unsigned int)localRegs_a[i]*localRegs_b[i]);
	}
//	for(int i=0; i<N; i++)
//		printf("%d: %d = %d * %d\n",i,localRegs_c[i],localRegs_a[i],localRegs_b[i]);
	//Execute Inverse-NTT for output c
	InvNTT:InverseNTT_256_hls(localRegs_c);
	//Write to output from memory
	WriteOutput: for (int i = 0; i<N; i++){
	#pragma HLS unroll factor=2
		a[i] = localRegs_c[i];
	}
}
```

Our interface accepts as input the coefficients $a(N)$ and $b(N)$ in a single array a, which are then inserted in our local RAMs $localRegs\_a$ and $localRegs\_b$, being dual-port 2-block RAMs as indicated by the pragmas right below their initialization. Then we run two NTT-functions labeled NTTA and NTTB for the inputs a and b respectively, the outputs are saved in the same input-memory of the functions. We continue with the dotMultiplication, which multiplies the outputs of the NTT-functions and applies Montgomery-reduction. The output of the dotMultiplication is saved in a separate RAM $localRegs\_c$ for convenience (we could re-use on of the two RAMs we

already initialized). At the end we write the output of the Inverse-NTT, which is saved in $localRegs\_c$, in our input single array a.

Figure 8 depicts the performance and result estimations from synthesizing the module and passing it to the HLS synthesizer:

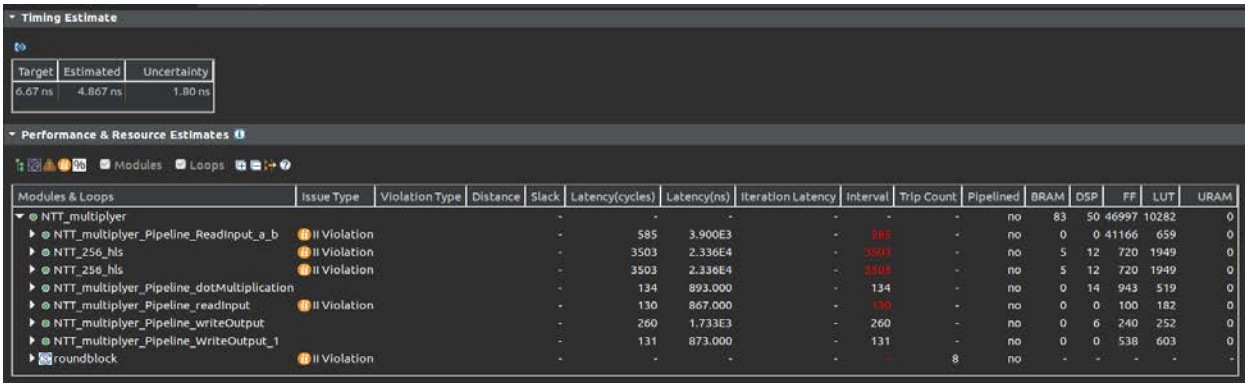

Figure 8: NTT Multiplication performance and result estimations

Figure 8 provides the area-utilization and latency of each module discussed above. From the HLS log it can be concluded that there are many optimizations that could be done in order to bypass the II Violation and decrease the latency of some modules. It could be mentioned here that *roundblock* corresponds to the InverseNTT, a label inside the Inverse-NTT function.

As a case study, we adopted the NTT computation of Dilithium Digital Signature scheme that uses a q value of 23 bits and n =256. The proposed approach can be applied to other NTT versions with different parameters. On each PE execution we need to calculate the corresponding input memory-indices for both arrays â and ω_n, then access the corresponding values of the arrays, execute PE calculations based on these values and finally update the coefficient array â with the previously calculated values.

As such, this multiplier is compared with the NTT-multiplier inside the Dilithium-Project based on the test_mul.c function (https://github.com/pq-crystals/dilithium/blob/master/ref/test/test_mul.c) .

The verification code can be found in the git-project https://gitlab.com/elkady.alexander/ntt/-/blob/main/codes/vitis_files/ntt_multiplyer/src/test/test_myNTT_multiplyer.cpp.

- At the start of the test we initialize our device with the initialize_device function.
- On lines 96 we have the outer loop of testing and on line 98 the inner loop based on number of polynomials $polyNum$.
- On lines 99-104 we produce some randoms polynomials and we insert them in the corresponding memories $noCL$ and $withCL$ corresponding to C-run and device HLS-run respectively.

- On lines 110-127 we run the multiplication based on our CPU C-code and we save the execution time.
- On lines 132-158 we run the multiplication based on our device HLS-run and we save the execution time.
- At the end, on lines 165-174 we compare the results of CPU-run and device-run in order to validate our design.
- On lines 176-185 we print the corresponding execution-times.

As a first step, utilizing Vitis we ran the test in a Software-Emulation mode and we debugged our code based on QEMU and XRT utilities. Then after the verification process of Software-Emulation, we deployed it to the Xilinx zcu104 device, and we got the following results on our terminal (based on SSH):

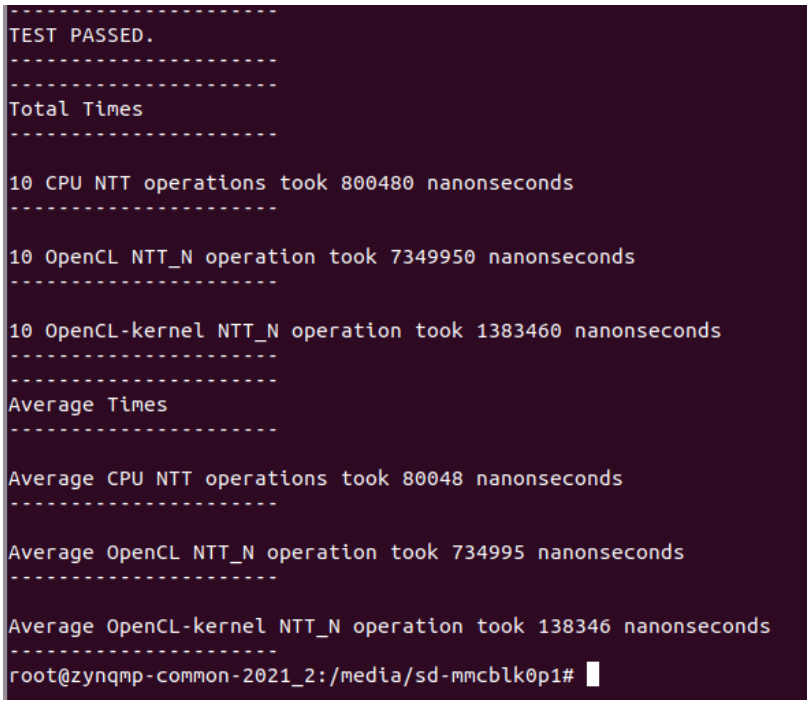

Figure 9: NTT Multiplication performance and result estimations

Our setup specification include a Cortex-M CPU running at 1.3GHz and an FPGA running at 150MHz. Table 14 shows the final results.

Table 14. NTT multiplication Results

	CPU-run	Zcu104-run I/O	Zcu104-kernel
Run-time in ns	**80048**	**596649**	**134346**
Clock Cycles	**61575384615**	**3977660**	**895640**

As we can see the clock-operation of our design and the corresponding latency is negligible in comparison with the CPU times. The big difference in frequency of the two devices, leads the CPU run time to be less than our device. We need to mention here that FPGAs are used for base-designs which are further implemented in ASICs, where frequencies are much higher from 150Mhz.

5. SW Runnables

5.1. DSM application

The functionality of the DSM application implemented in FPGA by UoP has been described in Section 4.1 and Figure 2. Part of the functionality of the predict() routines described in Figure 3 is implemented in HW as described in Sections 4.1.2 and 4.1.3.

5.1.1. SW Runnables of the DSM application implemented in FPGA

More specifically the functionality of the Regressor::predict() and Tree::predict() routines shown in Figure 3 were implemented by the HW kernel called predict_tree(). The rest of the functionality of the DSM application has been implemented in SW. The HW kernel predict_tree() is initiated within the top level Tracker::predict() routine of Figure 3. A full software implementation of the predict_tree() routine has also been released serving as a reference to compare the speed of the developed alternative versions of predict_tree() according to the ERT model employed.

5.1.2. OpenCL attributes in the DSM application implemented in FPGA

The HW kernels described in section 4.1.3 are called within the Tracker::predict() top level function in each cascade stage iteration. In the Xilinx Vitis environment calling a HW kernel requires some specific steps implemented with OpenCL commands and these steps are listed in Table 15.

Table 15: 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)

Steps 1-4 can be performed during initialization. Thus they can be executed within the predict_prepare() routine of Figure 2. Loading and enqueuing arguments (steps 5-6) that do not change between successive kernel calls can also be performed during initialization. The rest of the arguments are prepared and enqueued from the Tracker::predict() routine along with the rest of the steps described in Table 15. The OpenCL commands that implement the steps of this table are the following:

Predict_prepare():

```
char xclbinFilename[32];
strcpy(xclbinFilename,"binary_container_1.xclbin");

std::vector<cl::Device> devices;
cl::Device device;
std::vector<cl::Platform> platforms;
bool found_device = false;
cl::Platform::get(&platforms);
for(size_t i = 0; (i < platforms.size() ) & (found_device == false) ;i++)
{
        cl::Platform platform = platforms[i];
        std::string platformName = platform.getInfo<CL_PLATFORM_NAME>();
        if ( platformName == "Xilinx")
        {
                devices.clear();
                platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
                if (devices.size())
                {
                        device = devices[0];
                        found_device = true;
                        break;
                }
        }
}
if (found_device == false)
{
        std::cout << "[PCHRI] ERROR: Unable to find Target Device {Inside predict_prepare()}" <<
device.getInfo<CL_DEVICE_NAME>() << std::endl;
        return;
}
// Creating Context and Command Queue for selected device
data.context = cl::Context(device);
data.q= cl::CommandQueue(data.context, device, CL_QUEUE_PROFILING_ENABLE);
// Load xclbin
std::cout << "[PCHRI] MSG: Loading '" << xclbinFilename << " {Inside predict_prepare()}'\n";
std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
bin_file.seekg (0, bin_file.end);
unsigned nb = bin_file.tellg();
bin_file.seekg (0, bin_file.beg);
char *buf = new char [nb];
bin_file.read(buf, nb);
// Creating Program from Binary File
cl::Program::Binaries bins;
bins.push_back({buf,nb});
devices.resize(1);
cl::Program program(data.context, devices, bins);
// This call will get the kernel object from program. A kernel is an
// OpenCL function that is executed on the FPGA.
data.krnl_trees=cl::Kernel(program,"predict_trees");

//…….
data.btr_sr_tg=cl::Buffer(data.context, CL_MEM_READ_WRITE, sr_tg_sib);
data.btr_casc=cl::Buffer(data.context, CL_MEM_READ_WRITE, casc_sib);
```

```
data.btr_learningRates=cl::Buffer(data.context, CL_MEM_READ_ONLY, learningRates_sib);
data.btr_Global_node_split1=cl::Buffer(data.context,CL_MEM_READ_ONLY, Global_node_sib);
data.btr_Global_node_split2=cl::Buffer(data.context,CL_MEM_READ_ONLY, Global_node_sib);
data.btr_intensities=cl::Buffer(data.context, CL_MEM_READ_ONLY, intensities_sib);
data.btr_Global_node_thres=cl::Buffer(data.context,CL_MEM_READ_ONLY, Global_node_thres_sib);
data.btr_Global_node_mean=cl::Buffer(data.context,CL_MEM_READ_ONLY, Global_node_mean_sib);

// Set kernel arguments
data.krnl_trees.setArg(data.narg++,data.btr_sr_tg);
data.krnl_trees.setArg(data.narg++,data.tr_tg_len);
data.krnl_trees.setArg(data.narg++,data.btr_casc);
data.krnl_trees.setArg(data.narg++,data.btr_Global_node_split1);
data.krnl_trees.setArg(data.narg++,data.btr_Global_node_split2);
data.krnl_trees.setArg(data.narg++,data.btr_intensities);
data.krnl_trees.setArg(data.narg++,data.btr_Global_node_thres);
data.krnl_trees.setArg(data.narg++,data.btr_Global_node_mean);
//data.krnl_trees.setArg(data.narg++,data.btr_learningRates);

data.ptr_sr_tg = (float *) data.q.enqueueMapBuffer (data.btr_sr_tg, CL_TRUE , CL_MAP_WRITE , 0,
sr_tg_sib);
data.ptr_casc = (int*) data.q.enqueueMapBuffer (data.btr_Global_node_mean , CL_TRUE , CL_MAP_WRITE
, 0, casc_sib);
data.ptr_Global_node_split1 = (short *) data.q.enqueueMapBuffer (data.btr_Global_node_split1 ,
CL_TRUE , CL_MAP_WRITE , 0, Global_node_sib);
data.ptr_Global_node_split2 = (short *) data.q.enqueueMapBuffer (data.btr_Global_node_split2 ,
CL_TRUE , CL_MAP_WRITE , 0, Global_node_sib);
data.ptr_intensities = (float *) data.q.enqueueMapBuffer (data.btr_intensities , CL_TRUE ,
CL_MAP_WRITE , 0, intensities_sib);
data.ptr_Global_node_thres = (float *) data.q.enqueueMapBuffer (data.btr_Global_node_thres ,
CL_TRUE , CL_MAP_WRITE , 0, Global_node_thres_sib);
data.ptr_Global_node_mean = (float *) data.q.enqueueMapBuffer (data.btr_Global_node_mean , CL_TRUE
, CL_MAP_WRITE , 0, Global_node_mean_sib);

data.ptr_Global_node_split1= data.tr_Global_node_split1;
data.ptr_Global_node_split2= data.tr_Global_node_split2;
data.ptr_Global_node_thres= data.tr_Global_node_thres;
data.ptr_Global_node_mean= data.tr_Global_node_mean;
data.ptr_casc= data.tr_casc;

data.q.enqueueMigrateMemObjects(
{
        data.btr_Global_node_split1,
        data.btr_Global_node_split2,
        data.btr_Global_node_thres,
        data.btr_Global_node_mean
},0
```

Predict_kernel():

```
data.q.enqueueMigrateMemObjects(
{
        data.btr_intensities,
        data.btr_casc
},0
data.q.enqueueTask(data.krnl_trees);
data.q.enqueueMigrateMemObjects({data.btr_sr_tg},CL_MIGRATE_MEM_OBJECT_HOST);
data.q.finish();
```

The implementation of the all software version of the DSM application i.e., the one where *predict_tree()* is implemented in software and runs on an A53 core of ZynqMP FPGA does not include any OpenCL commands since it is called *by predict_kernel* as an ordinary software routine.

5.1.3. SW implementation of the predict_tree() kernels in the DSM application

The software implementation of the *predict_tree()* routine lead to an implementation of the *predict_kernel()* routine with an average latency of 42ms. This latency served as a reference to compare the acceleration achieved with the HW implementations of the *predict_tree()* routine. The latency of the *predict_kernel()* routine is the time needed to process a single frame thus, the inverse of this latency is the frames/sec that can be processed. Please take into consideration that the latency of 42ms has been achieved when the fastest clock is used with the A53 ARM core (1.3GHz). Reducing the speed of the core to the minimum clock frequency of 670MHz will double the latency. This case would be useful if the dynamic power consumption of the ARM core has to be reduced to the half.

5.2. NTT/INTT application

The functionality of the NTT/INTT multiplication implemented in FPGA by ISI has been described in Section 4.2. However, there are edge devices that do not include an FPGA but may include a GPU or even have a CPU that can support parallelization. To support these devices, ISI has implemented NTT/INTT in OpenCL explicitly for GPGPUs and CPUs (with OpenCL support).

5.2.1. SW Runnables of the NTT/INTT application.

The basic NTT and INTT algorithms which relies on the Cooley-Tukey (CT) approach can be seen in Algorithm 1 and Algorithm 2 in Section 4.2.

A GPU can contain a large number of cores, ranging from hundred to thousands, allowing tasks to be executed in parallel. GPUs are used for a multiple different applications, other than graphic computations, such as machine learning and cryptography. GPUs also have on-chip memory, which is small but fast, and off-chip memory, which is large but slow. It is up to the developer to allocate the memory to be used as well as provide the functional computations to be performed on the input. There are two major frameworks for heterogeneous parallel computing, NVIDIA's Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL).

Both NTT and Inverse NTT algorithms using CT-FFT are implemented using three for-loops taking as input a polynomial in the form of a vector v with n values, each n_i value being the coefficient of the polynomial. The outer for-loop counts the rounds of the algorithm. The number of rounds is equal to $\log_2(\mathrm{n})$. In each round the vector v is split into equal size blocks. While the number of rounds is equal for NTT and Inverse NTT, the computations and the length of each block is slightly different.

In NTT, each round the input vector is spilt into $2^{round-1}$ blocks as can be seen in Figure 10. Each block has a length equal to $\frac{n}{2^{round-1}}$ and is logically split into two halves. The two internal for-loops, as described in Algorithm 1 – lines 5-14, execute the butterfly operations for each round in each block by performing computations using one coefficient from the first half of the block with the corresponding coefficient in the second half of the block, $n[i]$, $n[i + \frac{blockLength}{2}]$. The

corresponding coefficients required for calculating the results for the next round for the second half of the block follows a similar pattern, $n[i]$, $n[i - \frac{blockLength}{2}]$.

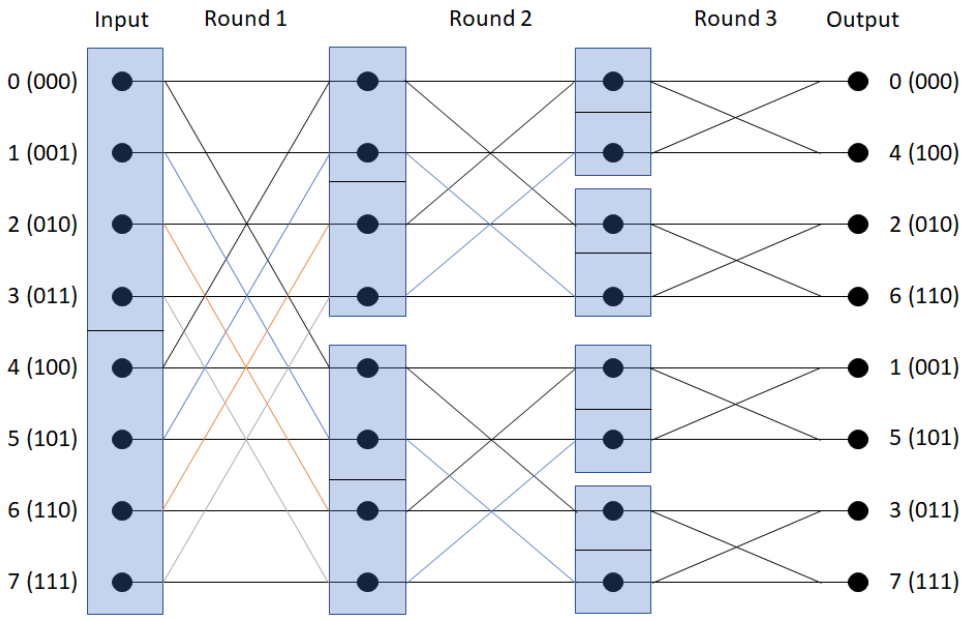

Figure 10: NTT computations without initial bit reversal

It should be noted that Figure 10 does not detail the computation performed with the coefficients as defined by the algorithm, rather it shows which coefficients are required to be processed with each other in order to output the required input for the next round of computations. It must also be noted that Algorithm 1 first does a bit reversal on line 1 on input vector and changes the order of the coefficients of the polynomial in order to perform the operations and calculate the output coefficients. While the example showed in Figure 10 does not perform the bit reversal, however the same operations are performed and the only difference is that the output is bit reversed. Our implementation of NTT implements Figure 10.

We can safely parallelize the computations of each round, which correspond to the two internal for-loops, as the computations that are occurring with the corresponding coefficients $n[i]$, $n[i + \frac{blockLength}{2}]$ in each block are independent of the other computations in all the other blocks of the vector v in the same round and are done in parallel and not sequential and don't depend on results from other rounds.

In Inverse NTT, the logic is the same, but the splitting of the blocks is backwards. In each round the vector is split into $2^{\log_2(n) - round - 1}$ blocks, which is the reverse of NTT. Each block has a length equal to $\frac{n}{2^{\log_2(n) - round - 1}}$. Identical to NTT, each block is logically split into two halves. The two internal for-loops execute the butterfly operations for each round in each block by performing computations using one coefficient of the first half of the block with the

corresponding coefficient in the second half of the block, $n[i]$, $n[i + \frac{blockLength}{2}]$. E.g. For a vector v with $n = 256$, for the first round, there are 128 blocks and the $n[0]$ coefficient is computed with the $n[1]$ coefficient.

As with NTT, Inverse NTT can also have the two internal for-loops safely parallelized as, again, the computations that are occurring with the corresponding coefficients $n[i]$, $n[i + \frac{blockLength}{2}]$ in each block are independent of the other computations in all the other blocks of the vector v in the same round and are done in parallel.

Finally, there was also another avenue to increase the efficiency of parallel platforms by batching computations as much as possible. Algorithms, such as discussed in I, may require the transformation of multiple polynomials in sequence. If there are enough cores available in the platform, we can batch the input with a number of equal sized polynomials as one input vector.

In order to implement batching, it was crucial to distinguish each index of the different polynomial coefficients so that we can identify the block, as well as which half it belongs to. This was simply implemented by discovering the local index of each coefficient of each polynomial based on the index location of the input vector, the global index, by performing a modulo operation of the global index with the size of the polynomials.

Once the local index is computed, it is easy to calculate the local block by performing the same operations as discussed previously. This same process is applied both for the NTT and the Inverse NTT algorithm. The output of the batching process is one vector containing the results of each polynomial in the same sequence as the input.

5.2.2. OpenCL attributes in the NTT/INTT application

The OpenCL kernels described in section 5.2.1 are called from a host program running on the CPU. For the host program to call an OpenCL kernel requires some specific steps implemented with OpenCL commands and these steps are listed in Table 16.

Table 16: Steps describing how an OpenCL kernel is loaded.

1	Detect device, platform and create Command Queue for the context of this device
2	Define, compile and load the OpenCL file with the kernels
3	Create a Program from the device, context
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

We have build a number of helper functions:

- readKernelFile: Reads a .cl file and returns the file as buffer pointer to be compiled
- buildOpenCLProgram: Compiles the kernel file read by readKernelFile
- createKernel: Creates a specific kernel based on its name from the built OpenCL program.

```
char* readKernelFile(char* filename, size_t* program_size) {
    FILE* program_handle;
    int j = fopen_s(&program_handle, filename, "rb");
    if (program_handle == NULL)
    {
        return NULL;
    }
    fseek(program_handle, 0, SEEK_END);
    *program_size = ftell(program_handle);
    rewind(program_handle);
    char* program_buffer = (char*)malloc(*program_size + 1);
    if (program_buffer == NULL) {
        return NULL;
    }
    program_buffer[*program_size] = '\0';
    fread(program_buffer, sizeof(char), *program_size, program_handle);
    fclose(program_handle);
    return program_buffer;
}
int buildOpenCLProgram(unsigned int gpuID, char* program_buffer, size_t program_size, cl_program*
program, cl_context* context, cl_device_id* device) {
        //Create initial info
        cl_platform_id* platforms;
        cl_platform_id platform;
        cl_uint num_platforms;
        cl_int i, err;
        char* ext_data;
        size_t ext_size, log_size;

        char* program_log;

        //Get platforms
        err = clGetPlatformIDs(1, NULL, &num_platforms);
        if (err < 0) {
                perror("Couldn't find any platforms.");
                return err;
        }
        platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) * num_platforms);
        clGetPlatformIDs(num_platforms, platforms, NULL);
        for (i = 0; i < num_platforms; i++) {
                err = (cl_int) clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, 0, NULL,
&ext_size);
                if (err < 0) {
                        perror("Couldn't read extension data.");
                        return err;
                }
                ext_data = (char*)malloc(ext_size);
                clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, ext_size, ext_data, NULL);
                if (gpuID == i) {
                        printf("Selected platform %d vendor: %s\n", i, ext_data);
                }
                else {
                        printf("Platform %d vendor: %s\n", i, ext_data);
                }
                free(ext_data);
        }
        err = (cl_int) clGetDeviceIDs(platforms[gpuID], CL_DEVICE_TYPE_GPU, 1, device, NULL);
        if (err < 0) {
                perror("Couldn't get the first platform.");
```

```
                return err;
        }
        *context = clCreateContext(NULL, 1, device, NULL, NULL, &err);

        //Create program
        *program = clCreateProgramWithSource(*context, 1, (const char**)&program_buffer,
&program_size, &err);
        free(program_buffer);
        const char options[] = "-cl-mad-enable -cl-std=CL1.2";

        //Build program
        err = clBuildProgram(*program, 1, device, options, NULL, NULL);
        //If build fails, find reason
        if (err < 0) {
                clGetProgramBuildInfo(program, *device, CL_PROGRAM_BUILD_LOG, 0, NULL,
&log_size);
                program_log = (char*)calloc(log_size + 1, sizeof(char));
                clGetProgramBuildInfo(program, *device, CL_PROGRAM_BUILD_LOG, log_size + 1,
program_log, NULL);
                printf("%s\n", program_log);
                free(program_log);
                return err;
        }
        return 0;


}

int createKernel(cl_program* program, char* kernelName, cl_kernel* kernel){
        cl_int err;
        //Create kernel
        *kernel = clCreateKernel(*program, kernelName, &err);

        return err;
}

int createCommandQueue(cl_context* context, cl_device_id* device, cl_command_queue* queue)
{
        cl_int err;

        *queue = clCreateCommandQueue(*context, *device, CL_QUEUE_PROFILING_ENABLE, &err);

        return err;
}
```

The OpenCL commands that implement the steps of Table 16 are the following:

```
    char filename[30] = "NTTCL.cl";
    size_t program_size;
    char* program_buffer = readKernelFile(filename, &program_size);
    err = buildOpenCLProgram(GPU, program_buffer, program_size, &program, &context, &device);
    char kernelName[30] = "NTT_N";
    err = createKernel(&program, kernelName, &NTTkernel);
    char invKernelName[30] = "InverseNTT_N";
    err = createKernel(&program, invKernelName, &InvNTTkernel);
    char PWMKernelName[30] = "PointWiseMontgomery";
    err = createKernel(&program, PWMKernelName, &PointWiseMont);
    err = createCommandQueue(&context, &device, &queue);

        polyvecNum noCL, withCL;

        for (i = 0; i < polyNum; ++i) {
            poly_uniform(&noCL.vec[i], seed, nonce++);
            memcpy(&withCL.vec[i], &noCL.vec[i], sizeof(uint32_t) * N);
        }
        NTTCL_polyvecNum(&withCL);
```

```
        InvNTTCL_polyvecNum(&withCL);
```

The following functions NTTCL_polyvecNum, InvNTTCL_polyvecNum and PointWiseCL_polyvecNum are the function that create call and return the results for the NTT, INTT and Pointwise montgomerry OpenCL kernels.

```
cl_ulong NTTCL_polyvecNum(polyvecNum* a)
{
        extern cl_context context;
        extern cl_kernel NTTkernel;
        extern cl_command_queue queue;

        cl_int err;

        cl_event prof_event;
        cl_ulong cl_tstart, cl_tend;

        cl_mem in_buff, out_buff, temp_buff;
        size_t global_size = N * polyNum;

        in_buff = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, a, &err);
        out_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
sizeof(cl_uint) * global_size, NULL, &err);

        uint32_t num = N;

        err = clSetKernelArg(NTTkernel, 0, sizeof(num), &num);
        err = clSetKernelArg(NTTkernel, 1, sizeof(cl_mem), &in_buff);
        //      err = clSetKernelArg(NTTkernel, 2, sizeof(cl_uint) * global_size, NULL);
        err = clSetKernelArg(NTTkernel, 2, sizeof(cl_mem), &out_buff);

        size_t work_units_per_kernel = units_kernel;
        if (global_size < units_kernel)
                work_units_per_kernel = global_size;

        err = clEnqueueNDRangeKernel(queue, NTTkernel, 1, NULL, &global_size,
&work_units_per_kernel, 0, NULL, &prof_event);
        err = clEnqueueReadBuffer(queue, out_buff, CL_TRUE, 0, sizeof(cl_uint) * global_size, a,
0, NULL, NULL);

        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START, sizeof(cl_tstart),
&cl_tstart, NULL);
        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END, sizeof(cl_tend),
&cl_tend, NULL);

        err = clReleaseMemObject(in_buff);
        err = clReleaseMemObject(out_buff);
        return cl_tend - cl_tstart;
}

cl_ulong InvNTTCL_polyvecNum(polyvecNum* a) {
        extern cl_context context;
        extern cl_kernel InvNTTkernel;
        extern cl_command_queue queue;

        cl_int err;

        cl_event prof_event;
        cl_ulong cl_tstart, cl_tend;

        cl_mem in_buff, out_buff, temp_buff, temp_buff2;
        size_t global_size = N * polyNum;

        in_buff = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, a, &err);
```

```
        temp_buff = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, a, &err);
        temp_buff2 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, a, &err);
        out_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
sizeof(cl_uint) * global_size, NULL, &err);

        uint32_t num = N;

        err = clSetKernelArg(InvNTTkernel, 0, sizeof(num), &num);
        err = clSetKernelArg(InvNTTkernel, 1, sizeof(cl_mem), &in_buff);
        //err = clSetKernelArg(InvNTTkernel, 3, sizeof(cl_uint) * global_size, NULL); //This is a
local buffer. No need to it set up
        err = clSetKernelArg(InvNTTkernel, 2, sizeof(cl_mem), &temp_buff); //This is now a global
buffer. No need to it set up
        err = clSetKernelArg(InvNTTkernel, 3, sizeof(cl_mem), &temp_buff2); //This is a local
buffer. No need to it set up
        err = clSetKernelArg(InvNTTkernel, 4, sizeof(cl_mem), &out_buff);

        size_t work_units_per_kernel = units_kernel;
        if (global_size < units_kernel)
                work_units_per_kernel = global_size;

        err = clEnqueueNDRangeKernel(queue, InvNTTkernel, 1, NULL, &global_size,
&work_units_per_kernel, 0, NULL, &prof_event);
        err = clEnqueueReadBuffer(queue, out_buff, CL_TRUE, 0, sizeof(cl_uint) * global_size, a,
0, NULL, NULL);
        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START, sizeof(cl_tstart),
&cl_tstart, NULL);
        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END, sizeof(cl_tend),
&cl_tend, NULL);
        err = clReleaseMemObject(in_buff);
        err = clReleaseMemObject(temp_buff);
        err = clReleaseMemObject(temp_buff2);
        err = clReleaseMemObject(out_buff);
        return cl_tend - cl_tstart;
}

cl_ulong PointWiseCL_polyvecNum(polyvecNum* result, polyvecNum* a, polyvecNum* b) {
        extern cl_context context;
        extern cl_kernel PointWiseMont;
        extern cl_command_queue queue;

        cl_int err;

        cl_event prof_event;
        cl_ulong cl_tstart, cl_tend;

        cl_mem in_buff, in_buff2, out_buff;
        size_t global_size = N * polyNum;

        in_buff = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, a, &err);
        in_buff2 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(cl_uint) * global_size, b, &err);
        out_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
sizeof(cl_uint) * global_size, NULL, &err);


        err = clSetKernelArg(PointWiseMont, 0, sizeof(cl_mem), &in_buff);
        err = clSetKernelArg(PointWiseMont, 1, sizeof(cl_mem), &in_buff2);
        err = clSetKernelArg(PointWiseMont, 2, sizeof(cl_mem), &out_buff);

        size_t work_units_per_kernel = units_kernel;
        err = clEnqueueNDRangeKernel(queue, PointWiseMont, 1, NULL, &global_size,
&work_units_per_kernel, 0, NULL, &prof_event);
        err = clEnqueueReadBuffer(queue, out_buff, CL_TRUE, 0, sizeof(cl_uint) * global_size,
result, 0, NULL, NULL);
```

```
        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START, sizeof(cl_tstart),
&cl_tstart, NULL);
        err = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END, sizeof(cl_tend),
&cl_tend, NULL);

        err = clReleaseMemObject(in_buff);
        err = clReleaseMemObject(out_buff);
        return cl_tend - cl_tstart;
}
```

The following are the OpenCL kernel files for NTT, INTT and PointwiseMontgomery:

```
__kernel void NTT_N(const unsigned int coN, __global unsigned int* data, __global unsigned int*
outData)
{
        size_t globalId = get_global_id(0);
        //We may have more than one polynomial. In this case, we need to find the local block id
(in our 256 block length). Thus localId = globalId % 256;
        size_t localId = globalId % coN;
        //The number of rounds is the log2(coN)
        unsigned int rounds = (unsigned int)log2((float)coN);

        //Loop: The round number we're in. Starts from 1 and ends with 8.
        unsigned int loop;
        //StartingZeta: The Zeta starting point for the round.
        unsigned int sz;
        //OffsetZeta: The offset zeta for the inner loop of the round
        unsigned int ofz;
        //Block number: The block in the round
        unsigned int block;
        //Len: the length of each block in the round.
        unsigned int len;
        //CurrentZeta: The zeta value
        unsigned int zeta;
        int localTemp;
        //localData[globalId] = data[globalId];
        barrier(CLK_GLOBAL_MEM_FENCE);
        for (loop = 1; loop <= rounds; ++loop) {
                //Each loop has a number of blocks.
                //Starting at 256, the first loop has two blocks of 128 ints.
                //Next loop has 4 blocks of 64 bits. Each round has twice more blocks than the
previous.
                //As such the length of each block is 256/(2^loop). The total number of ints
divided by the 2^loop (1->2, 2->4, 3-->8, 4-->16, etc..)
                len = coN >> loop;
                //Calculate Starting Zeta of the round.
                //First round is 1, second round is 2, third round is 4, fourth round is 8,...
                //Therefore Starting Zeta is 2^(loop-1)
                sz = 1 << (loop - 1);
                //Calculate Offset Zeta for the block of the round.
                //For the first round both blocks have the same zeta. sz+0
                //For the second round the first two blocks have sz+0, the second two blocks
have sz+1
                //For the third round the first two blocks have sz+0, the second two blocks have
sz+1, the next two blocks have sz+2 and the last two blocks have sz+3
                //We need to find the position of the int based on the location. Round Down of
div globalId/(len*2) will give us the block number
                ofz = sz + localId / (len * 2);
                //Now that we have the zeta offset we can get the zeta.
                zeta = zetas[ofz % 256];
                //We need to find the block id. Therefore the block number is:
                block = localId / len;
                //Calculate Temp value. If we're at the first half of the block we need to
calculate the temp with offset of len:
                if (block % 2 == 0) {
                        localTemp = montgomery_reduce((unsigned long)zeta * data[globalId +
len]);
```

```
                }
                //If we're at the second half of the block we don''t have to use the offset
                else {
                        localTemp = montgomery_reduce((unsigned long)zeta * data[globalId]);
                }
                //Block here for everyone to catch up
                barrier(CLK_GLOBAL_MEM_FENCE);
                if (block % 2 != 0) {
                        data[globalId] = data[globalId - len] + 2 * Q - localTemp;
                }
                //Block here for everyone to catch up
                barrier(CLK_GLOBAL_MEM_FENCE);
                if (block % 2 == 0) {
                        data[globalId] = data[globalId] + localTemp;
                }
                //Block here for everyone to catch up
                barrier(CLK_GLOBAL_MEM_FENCE);
        }
        outData[globalId] = data[globalId];
        //outData[globalId] = rounds;
}

__kernel void InverseNTT_N(const unsigned int coN, __global unsigned int* data, __global unsigned
int* localData, __global unsigned int* localTemp, __global unsigned int* outData)
{
        size_t globalId = get_global_id(0);
        //We may have more than one polynomial. In this case, we need to find the local block id
(in our 256 block length). Thus localId = globalId % 256;
        size_t localId = globalId % coN;
        //The number of rounds is the log2(coN)
        unsigned int rounds = (unsigned int)log2((float)coN);

        //Loop: The round number we're in. Starts from 1 and ends with 8.
        unsigned int loop;
        //StartingZeta: The Zeta starting point for the round.
        unsigned int sz;
        //OffsetZeta: The offset zeta for the inner loop of the round
        unsigned int ofz;
        //Block number: The block in the round
        unsigned int block;
        //Len: the length of each block in the round.
        unsigned int len;
        //CurrentZeta: The zeta value
        unsigned int zeta;
        localData[globalId] = data[globalId];
        barrier(CLK_GLOBAL_MEM_FENCE);

        for (loop = rounds; loop > 0; --loop) {
                //Each loop has a number of blocks.
                //Starting at 256, the first loop has two blocks of 128 ints.
                //Next loop has 4 blocks of 64 bits. Each round has twice more blocks than the
previous.
                //As such the length of each block is 256/(2^loop). The total number of ints
divided by the 2^loop (1->2, 2->4, 3-->8, 4-->16, etc..)
                len = coN >> loop;
                //Calculate Starting Zeta of the round.
                //First round is 0, second round is 1, third round is 3, fourth round is 8,...
                //Therefore Starting Zeta is 2^(loop-1)
                //sz = (1 << (8 - loop));
                sz = coN - (1 << (loop));
                //Calculate Offset Zeta for the block of the round.
                //For the first round both blocks have the same zeta. sz+0
                //For the second round the first two blocks have sz+0, the second two blocks
have sz+1
                //For the third round the first two blocks have sz+0, the second two blocks have
sz+1, the next two blocks have sz+2 and the last two blocks have sz+3
                //We need to find the position of the int based on the location. Round Down of
div globalId/(len*2) will give us the block number
                ofz = sz + localId / (len * 2);
```

```
                //Now that we have the zeta offset we can get the zeta.
                zeta = zetas_inv[ofz % 256];
                //We need to find the block id. Therefore the block number is:
                block = localId / len;
                if (block % 2 == 0) {
                        localTemp[globalId] = localData[globalId];
                        localData[globalId] = localData[globalId] + localData[globalId + len];
                        barrier(CLK_GLOBAL_MEM_FENCE);
                }
                barrier(CLK_GLOBAL_MEM_FENCE);
                //Block here for everyone to catch up
                if (block % 2 == 1) {
                        localData[globalId] = montgomery_reduce((unsigned long)zeta *
(localTemp[globalId - len] + 256 * Q - localData[globalId]));
                        barrier(CLK_GLOBAL_MEM_FENCE);
                }
        }
        //Block here for everyone to catch up
        outData[globalId] = montgomery_reduce((unsigned long)f * localData[globalId]);
}

__kernel void PointWiseMontgomery(__global unsigned int* a, __global unsigned int* b, __global
unsigned int* outData)
{
        //Get ID
        size_t globalId = get_global_id(0);
        //Calculate pointwise multiplication
        outData[globalId] = montgomery_reduce((unsigned long)a[globalId] * b[globalId]);
}
```

5.2.3. SW implementation of the NTT, INTT

As a testing environment, we implemented our work on an Intel i7-7700 CPU with 3.6 Ghz frequency hosting an NVIDIA GTX 1050 Ti GPU with 786 cores running Windows 10. The selection criteria of this GPU was the utilization of an average performance commercial device. Running our NTT/INTT implementation for the Dilithium with 256 number of coefficients and a batch size of 4, the CPU requires 6,4 seconds, while the OpenCL code requires 10 seconds. The main bottleneck that creates this disparity is the memory transfer between the CPU and GPU.

With a larger number of coefficients and different batch size, our final results showed that there is a potential speed up against the CPU, with a maximum of speed up around 7.5x times depending on the number of input coefficients. We found that for an average number of 1664 coefficients the GPU kernel execution time is on par with the CPU and after that there is significant gain on using the GPU.

6. Conclusions

In the above deliverable the work done in T5.1 of CPSoSaware WP5 has been presented. The work is focused on showcasing how complex runnable can be produced in hardware and in software using HLS tools and/or purely OpenCL compilers (for hardware and software respectively). The work of D5.1 is the last part of a series of tasks that aim to model, design, optimize and eventually implement efficiently CPS and CPSoS applications that are in line with the project activities.