H2020 FRAMEWORK PROGRAMME
ICT-01-2014: Smart Cyber-Physical Systems

PROJECT NUMBER: 645496

Agile, eXtensible, fast I/O Module for the cyber-physical era

D3.2 – Report on Proof of Concepts

Due date of deliverable: 31st January 2017
Actual Submission: 7th February 2017 (agreed extended date)

Start date of the project: 1st February 2015 Duration: 36 months

Lead contractor for the deliverable: UNISI

Revision: See file name in document footer.

Project co-founded by the European Commission within the HORIZON FRAMEWORK PROGRAMME (2020)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Date	Author	Organization	Change History
0.1	01.01.2017	Antonio Rizzo	UNISI	Initial version
0.2	03.01.2017	David Oro	HERTA	Added SVS section
0.3	05.01.2017	Nicola Bettin	VIMAR	Added SHL section
1.0	31.01.2017	Antonio Rizzo	UNISI	Improved sections 1-8
1.5	02.02.2017	Xavier Martorell	BSC	Revisions
1.5	02.02.2017	Stefano Garzarella	EVI	Revisions
2.0	03.02.2017	Antonio Rizzo	UNISI	Final version

Release Approval

Name	Role	Date
Antonio Rizzo	WP Leader	03.02.2017
Roberto Giorgi	Project Coordinator for formal deliverable	04.02.2017

The following list of authors will be updated to reflect the list of contributors to the document.

Antonio Rizzo, Francesco Montefoschi, Sara Ermini
Department of Social, Political and Cognitive Sciences
University of Siena – UNISI – AXIOM

David Oro
RD department
Herta Security – HERTA – AXIOM

Nicola Bettin
RD department
VIMAR S.p.A. – VIMAR – AXIOM

Printed in Siena, Italy, Europe.
Part number: *Please refer to the File name in the document footer*.

TABLE OF CONTENTS

GLOSSARY

AEP – AXIOM evaluation platform
ARM – Instruction set architecture developed by ARM Holdings Ltd.
ASIC – Application-specific integrated circuit
ATLAS – Automatically tuned lineal algebra software
BLAS – Basic linear algebra subprograms
CNN – Convolutional neural network
FC – Fully-connected layer
FFT – Fast Fourier transform
FIFO – First in first out
FP32 – 32-bit floating point number
GCC – GNU compiler collection
GLFW – Open-source multiplatform library for OpenGL, OpenGL ES and Vulkan
GMM – Gaussian mixture model
GUI – Graphical user interface
HCI – Human Computer Interaction
HDL – Hardware description language
HLS – High-level synthesis
IDCT – Inverse discrete cosine transform
IDE – Integrated development environment
INT32 – 32-bit integer number
IP – Intellectual property or internet protocol (depending on the context)
IRM – Iris recognition module
LBP – Local binary pattern
LibAv – Open-source libraries derived from the FFmpeg project to handle multimedia data
LRN – Local response normalization
Mali – A GPU microarchitecture developed by ARM Holdings Ltd.
Mercurium – OmpSs compiler
Nanos++ – OmpSs runtime
NDA – Non-disclosure agreement
NEON – SIMD extensions for the ARM instruction set
NIC – Network Interface Controller
OpenCV – Open source computer vision library
OpenGL ES – Reduced specification of the OpenGL standard that targets embedded devices
PCM – Pulse-code audio modulation
PL – Programmable logic
PReLU – Parametric rectified linear unit
Qt – Cross-platform application framework developed by The Qt Company
RGB – Red-Green-Blue color space format
ROC – Receive operating characteristic

ROI – Region of interest
RTL – Register transfer language
RTSP – Real-time streaming protocol
SGEMM – Single-precision floating-point general matrix multiply
SHL/SLH – Smart home living scenario developed for the AXIOM project
SIMD – Single instruction, multiple data
SMP – Symmetric multiprocessing
SMT – Simultaneous multithreading
SoC – System on chip
SPro – Open source speech signal-processing toolkit
SSE2 – Streaming SIMD Extensions 2
STD – Standard deviation
SVM – Support vector machine
SVS – Smart surveillance scenario for the AXIOM board
VAD – Voice activity detection
VHDL – VHSIC hardware description language
VPM – Video processing module
YUV – Luminance blue–luminance red–luminance color format

Executive summary

This deliverable reports the application development and porting for Smart Video Surveillance and the Smart Home Living scenarios.

The SVS case study implements a real-time face analysis framework that processes video feeds for surveillance applications. This framework is the core component of a software solution that is aimed to both increase security and gather demographic statistics in highly-crowded areas such as train stations, airports and shopping malls. In order to solve this challenging problem, an initial application framework prototype was developed for the target SoC included in the AXIOM board. The initial analysis and design space exploration of such face analysis kernels shows that the OmpSs programming model greatly increases the programmer's productivity while scaling performance with minimal efforts. Additionally, the developed prototype also served as a baseline for testing and training from scratch the required convolutional neural networks required for estimating the gender and age of detected faces.

The SHL case study implements a solution to enhance the security level of dwellings, and to increase the natural interaction for end-users and their homes. The SHL solution was implemented looking forward to take advantage of the heterogeneous resources of the AXIOM system through the OmpSs programming model. A set of benchmarks were analyzed and enriched by the OmpSs directives in order to exploit the underlying SMP resources. The exploration has shown the possibility of easy parallelizing the SHL application by relying on OmpSs directives. The obtained results, knowledge gained on OmpSs programming model, and the visualization/profiling tools have created a solid base to continue the improvement of the SHL algorithms. Future work will take advantage of FPGA resources and scalability across nodes built from interconnecting several AXIOM boards.

Designing an IoT product requires a different approach to user experience. Moreover, in AXIOM the challenges addressed by the two case studies rest on machine learning solutions. We must acknowledge that today there are no authoring tools for rapid prototyping Apps or Services based on Interactive Machine Learning. In the design of our prototyping environment, we found that the best opportunities were offered by App Inventor, an open source Web IDE. We extended App Inventor adding machine learning capabilities to facilitate the production of interactive prototypes of the future applications and services based on the challenges addressed in both case studies.

1 Introduction

1.1 Document structure

This deliverable contains a report on the work about the proof of concept and the porting of the SVS and SHL scenarios. The work is reported and organized as follows:

- Section 2 describes the porting activity for the SVS scenario;
- Section 3 describes the porting activity for the SHL scenario;
- Section 4 describes the prototyping of envisioned applications and services;
- Section 5 describes the data collection activity used for machine learning training.

1.2 Tasks involved in this deliverable

This deliverable is the result of the work carried out during the following task:

- **Task 3.2**: Proof of Concept and Porting of SHL and SVS Case Studies

  Selection, envisioning and refinement of Scenarios to be put into scene by prototypes of AXIOM architecture in the domain of Smart Living Home and Smart Video Surveillance.

  Porting of the Smart Living Home Application and Smart Video Surveillance to the OmpSs Programming Model.

  Partner UNISI (Interaction Design group) will envision new scenarios for the using the AXIOM CPS platform. UNISI will take care of the "Role Prototyping" of the App/Service, while addressing the two challenges of services/system integration and appealing user experience. UNISI will define the Interaction Design pattern in the design of the application on the AXIOM CPS. UNISI will carry out the Conception and Definition of the user experience in adopting into scene the new enabling CPSs.

  Partner VIMAR will design and develop algorithms for real-time data management used in the Home Automation application. Partner VIMAR will develop a modular, cost and power effective software architecture including a reference version and the porting of such reference version to the OmpSs programming model (the hardware part will be developed in WP6).

  Partner HERTA will design and develop algorithms for real-time face recognition used in the smart surveillance application. Partner HERTA will optimize the fine-grained parallel algorithm for FPGA accelerator and will develop and port their application to AXIOM architecture using OmpSs.

  Partner BSC will give support to VIMAR and HERTA for the porting of the applications to OmpSs.

2 Smart Surveillance Scenario

The Smart Surveillance Scenario (SVS) use case is modelled for serving as a proof of concept for an embedded version of a facial analysis product for marketing applications. This scenario leverages the AXIOM platform for performing low-power H.264 video decoding, face detection, and for conducting gender and age estimation in real-time on each person's face appearing on a given video frame. Potential future end-user implementations of such technology include automated video surveillance for preventing terrorism and security threats, and marketing applications for the retail sector among many others. Those scenarios extensively rely on analyzing facial features on crowded locations to avoid discarding frames when they are broadcasted from live surveillance IP cameras. This performance requirement can be met by using a combination of traditional computer vision techniques and convolutional neural networks to guarantee high-accuracy.

In order to avoid architecting IP blocks for implementing those kernels in an RTL hardware description language, the OmpSs programming model and #pragma annotations are extensively used to port sequential versions of the algorithms directly coded in C/C++ language. Complex steps such as caching, overlapping memory transfers with computations to hide latencies, and thread scheduling rely on the rest of the system architecture, e.g., on the BSC's Nanos++ runtime. On the other hand, the automatic generation of HDL code of the hardware IP blocks from annotated C/C++ code is managed by BSC's Mercurium compiler infrastructure by seamlessly interacting with Xilinx's Vivado HLS tools to generate the proprietary bitstream for the target FPGA microarchitecture.

In deliverable D3.1, it was disclosed that the SVS implemented in WP3 would rely mainly on four high-level kernels that constituted a good representation for emerging video processing and machine learning workloads. These kernels are summarized again in Table 1 enclosed below.

Table 1. *Selected workloads for the SVS*

Kernel Name	Description
`H264_video_decoding`	H.264 codec decoder working at slice level
`LBP_cascade_evaluation`	Face detection based on LBP patterns
`CNN_inference`	Convolutional neural network inference engine
`YUV_to_RGB`	Color space conversion for displaying frames

The selected kernels are in fact subsequently split into several low-level calls to other kernels that implement partial sub steps of the abovementioned processes. For instance, `H264_video_decoding` kernel internally include entropy decoding, inverse quantization, IDCT, deblocking filtering, intra-prediction and motion compensation. Similarly, the evaluation of the cascade classifier of LBP features `LBP_cascade_evaluation` performs several resizing and filtering operations to build a synthetic image pyramid, while `CNN_inference` depends on the inner architecture of the neural network model. In this latter case, the most resource-intensive parts are convolutions and the evaluation of fully-connected layers as it has been pointed out in prior research works [1]. Finally, the color space conversion kernel `YUV_to_RGB`) is the most naïve one, and certainly does not constitute a challenging workload when compared to the others.

Source code of the selected kernels is included in the Appendix as a reference. Due to the complexity and length of the `H264_video_decoding` kernel, its corresponding code listings were omitted in this deliverable.

Since the first prototype of the AXIOM board features the Xilinx Zynq UltraScale+ ZU9EG SoC, H.264 video decoding cannot be performed on the fixed-function logic decoder since this hardware IP block is only available on the ARM Mali video processor included in the Xilinx Zynq UltraScale+ EV series. As such, for the moment it was decided not to implement video decoding on the FPGA using a third-party IP block with the help of SECO as it was originally planned. The main reason behind this decision was to free up as much programmable logic (PL) resources as possible for the AXIOM NIC block, and for offloading time-critical face detection and CNN inference computations.

2.1 Software architecture

Between months m17 through m19, an initial working prototype of the SVS application was coded with the aim of integrating the execution flow of all the involved kernels for performing facial analysis using input H.264 videos in an easy manner. Since the standalone application kernels were meant to be profiled and optimized with the toolsets and runtime environments provided by BSC, it was decided to develop the monolithic application in standard ANSI C language. This decision greatly simplified the debugging process and eased the interaction with the Mercurium compiler used for compiling the annotated kernels. As such, the prototype application interfaced with minimal third-party libraries mainly for display (OpenGL ES), and for video demuxing and decoding (LibAV [2]).

Figure 1 depicts the high-level architecture of the different software components involved for profiling and testing the selected kernels.

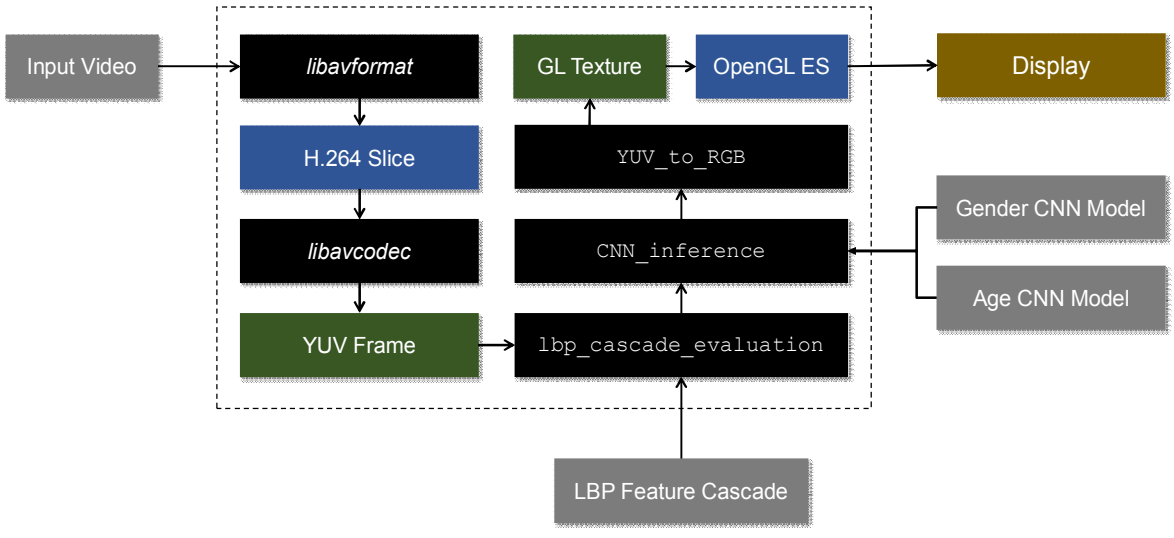

Figure 1. *High-level architecture of the SVS prototype application.*

Input video footage (either a file or a live RTSP stream) is parsed and demuxed by means of the `libavformat` library, which is manually set to only retrieve a video stream tagged as `AV_CODEC_ID_H264`. At this point, H.264 slices are decoded on software on CPU cores by relying on the multithreaded implementation of the `libavcodec` library. It should be noted that at a future stage of

the AXIOM project, this video software decoding layer could be replaced by an IP block if enough PL resources are freed up.

The output of the decoder is usually a YUV4:2:0 raw video frame, in which chrominance components are discarded as the selected facial analysis algorithms require only luminance. Therefore, an output buffer of `uint8_t` luminance components obtained from the decoded frame is used as an input of the `LBP_cascade_evaluation` and `CNN_inference` kernels. Finally, the results of such steps (e.g. bounding of located faces and estimation of demographic attributes) are annotated over the RGB color version of the decoded frame obtained after calling to the `YUV_to_RGB` kernel. After such color conversion takes place, the final resulting buffer is mapped into a texture to display the final results through the OpenGL ES pipeline.

Since the development and debugging of the prototype application was conducted on a PC platform in which the GPU driver used for display only has OpenGL support on Linux environments, the GLFW version 3.2.1 library was also selected for ensuring portability. This fact enabled OpenGL ES support for the ARM Mali-400 GPU included on the Xilinx Zynq UltraScale+ SoC, and the typical OpenGL stack available on PC platforms.

2.2 Face classifier cascade model

The binary face classifier model used for locating faces on the SVS application prototype was trained by HERTA using proprietary internal tools and databases, and will not be publicly released at the end of the AXIOM project. This model consists of a cascade of features trained using a customized version of the classic Adaboost algorithm [3] (see Figure 2). As previously agreed, the selected features are based on the well-studied local binary patterns (LBP) since they provide a good trade-off between accuracy and speed for low-power embedded devices. Even though state of the art results in accuracy for face detection are obtained using very deep CNNs, it is still unfeasible to adopt them in low-power embedded devices that target video frames involving dozens of simultaneous faces at HD resolutions and beyond. Real-time applications using deep CNNs for object detection at HD or 4K resolutions, currently require a full-custom ASIC specifically designed for inference, a high-end discrete GPU or a high-density FPGA. All three implementations are expensive, and the latter two usually dissipate more than 100 Watts at 16 nm [4] [5].

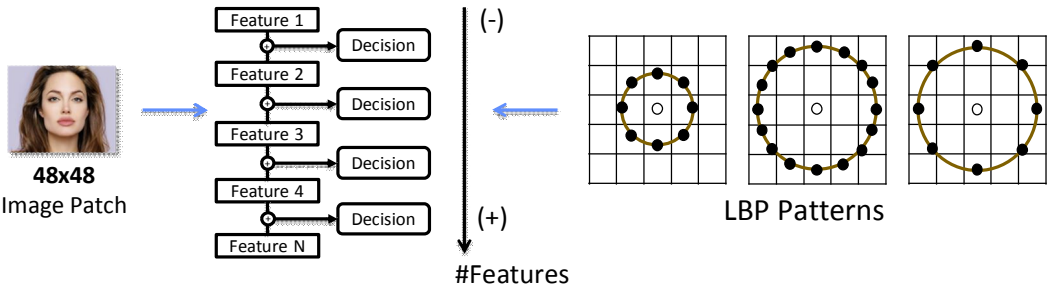

Figure 2. *Evaluation of the face classifier cascade*

Additionally, LBP features do not require to perform complex operations, and work with 8-bit integer arithmetic, which are more power-efficient and simpler to map to the underlying FPGA architecture during synthesis than other features requiring floating point arithmetic.

As a result of this, the obtained cascade has over a thousand LBP features, works with faces as small as 48x48 pixels, and when evaluated accumulates single-precision floating point scores resulting from extracted features until the target threshold is met or a given feature early rejected. Therefore, an exhaustive sliding-window approach is effectively used for analyzing the input image with the purpose of quickly discarding regions not containing faces. Even though less accurate, this approach clearly has benefits when compared to other face detection algorithms involving CNN architectures such as the widely used VGG-16 [6], GoogleNet [7], ResNet [8] or variations of the Faster R-CNN [9] framework.

Further work will be carried out with our model during FPGA implementation to avoid using floating point arithmetic for scores. A common approach is to replace these values with binarization and fixed-point arithmetic, similarly as it is currently being done for neural network pruning targeting inference on low-power embedded devices. These strategies will be explored between months m26 and m36.

Finally, the obtained trained cascade model and the evaluation source code used was kept confidential, and distributed between the interested AXIOM partners under NDA in order to discuss the potential optimization strategies to be followed.

2.3 Convolutional neural network models

After face coordinates and dimensions are located on a given video frame, the SVS prototype application conducts the required underlying analysis of the facial features for marketing and surveillance. This analysis is implemented by means of several CNNs that are meant to be executed in parallel on the programmable logic using task-based parallelism.

During months m13 and m20 the construction and modelling of such networks took place. So far, the AXIOM project has succeeded in training two highly-accurate deep CNNs for face gender classification and age estimation. The architecture of such networks was designed from scratch, and thus it is not based on the generic deep learning models already found in popular frameworks such as Caffe [10], Torch [11] or TensorFlow [12]. Figure 3 depicts the different layers used in the gender classifier, and age regressor. The binary classifier for gender estimation is implemented using a 10-layer neural network while the age regressor consists of 11 layers. The input of both networks is a given detected face (i.e. image region) that must be downscaled to map the expected input size of the first convolutional layer. At this point, each network use different types of layers by relying on convolutions, parametric rectified linear units (PReLU), standard deviation (STD) or max-pooling, local response normalization (LRN), fully-connected (FC), maxout units and softmax functions. Finally, the output of the gender classifier network is an integer (0,1) encoding whether the input image is male (0) or female (1). Similarly, the regressor outputs the age estimation normalized in the interval (0.0-1.0), and encoded using a 32-bit floating-point number.

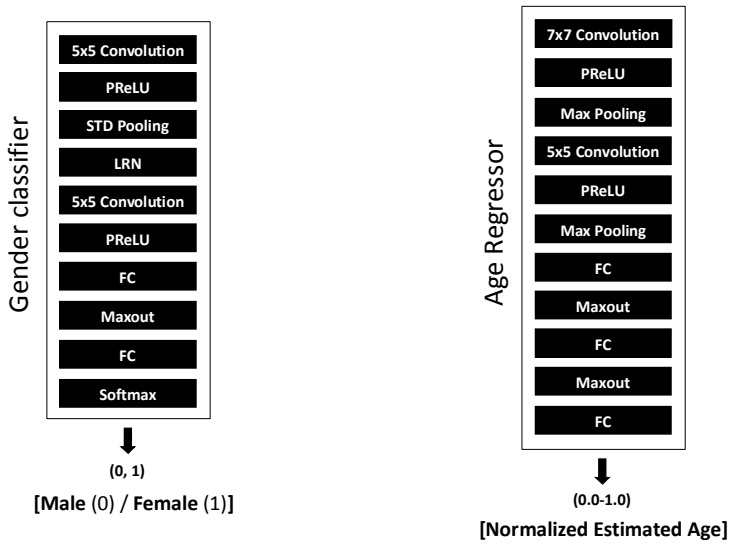

Figure 3. *Architecture of the facial analysis CNNs trained for the AXIOM project*

The purpose of designing and training such networks from scratch was useful for HERTA for gaining highly valuable knowledge about this very complex fine-tuning process. Another important reason was to succeed producing models with lower computational footprint than other alternatives typically requiring a power-hungry high-end discrete GPU for real-time CNN inference.

The obtained ROC curves benchmarking the accuracy of such networks on several research databases are going to be kept confidential, as they were trained using proprietary HERTA face databases in combination with several data augmentation techniques, and will be later integrated on a future commercial product. The size of the database used for training the networks was tens of thousands of pictures before applying data augmentation, which by itself increased several orders of magnitude the number of pictures. Other model parameters such as dimensions of matrices, the strides used for convolutional layers, and number of channels are also not going to be disclosed on publicly available deliverables. Therefore, another approach was used to achieve our project goals as described below.

In order to guarantee compliance with the mandate of open access research data sets described in the guidelines of the Horizon 2020 program, HERTA partnered with UNISI to construct a face database for retraining the CNN models earlier described. This face database will be publicly released on the AXIOM website at the end of the project. It is expected that this approach will be a win-win for both all the partners involved in the AXIOM project and the whole academic research community:

- Firstly, HERTA can add the pictures collected by UNISI to its internal proprietary face database to improve the quality of the models used in its products.
- Secondly, AXIOM partners will have access to working CNN models trained exclusively with the database collected by UNISI, which will be enough for evaluating optimization strategies for speeding-up CNN inference on the PL of the AXIOM board. This also guarantees reproducible research results, as there are no IP restrictions with such database and models.
- Finally, the research community will also have the possibility of downloading the AXIOM face database using a permissive license. As such, the research community can also use the pictures to increase the size of their data sets, and thus improve the overall quality of deep learning models for face analytics.

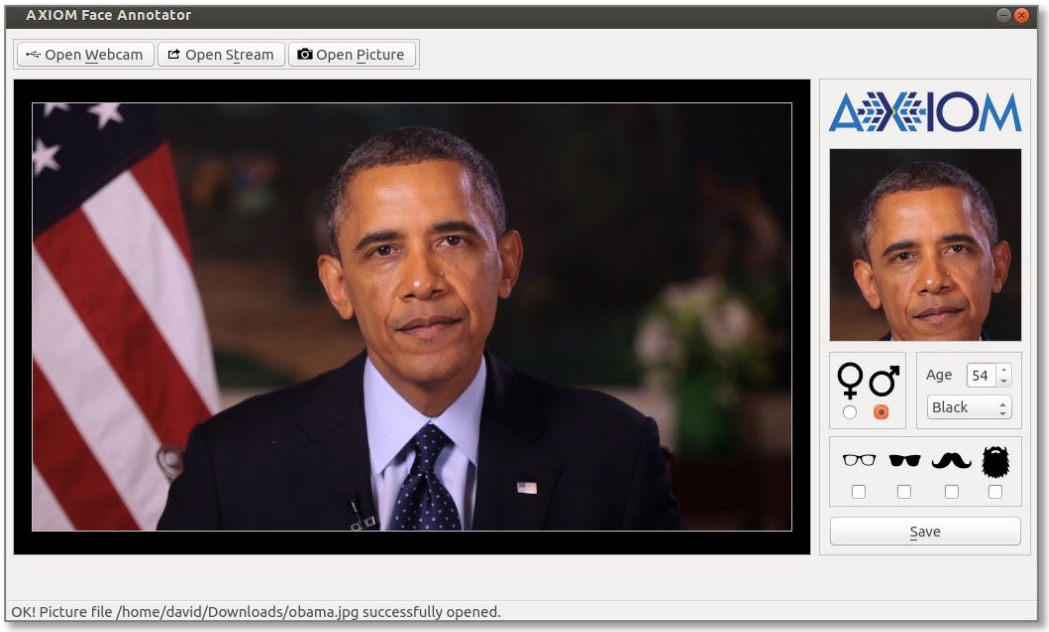

Figure 4. *User interface of the AXIOM face annotator application*

Once this decision was agreed with UNISI, during month M14 a face annotator tool (shown in Figure 4) was developed by HERTA to increase the productivity of the process of capturing and labelling pictures for the facial database. The open-source application is available at the project's GIT server, and was successfully tested working with USB webcams, network-based surveillance cameras, and pre-recorded video files.

The AXIOM face annotator tool was developed in C++ by relying on the Qt5 cross-platform application framework. The idea of this side development was also to label attributes such as gender, age, ethnicity and other features (i.e. glasses, sunglasses, beard and moustache) in the pictures collected by UNISI. Future improvements could also include the possibility of adding fiducial landmark points to further improve the accuracy of the CNN models.

2.4 Status of porting to the AXIOM platform using OmpSs

In accordance with the architectural decisions described in subsection 2.1, the selected kernels were isolated in separated `.c` and `.cpp` files and directories. The full source code of the LBP-based face classifier, CNN inference engine, and color space conversion is available on the AXIOM GIT server.

The development of the prototype was conducted on an x86-64 workstation with the aim of transparently porting the CPU host code later to the ARMv7 and ARMv8 architectures available on Xilinx SoC platforms by means of the backend of the C/C++ compiler used for generating the application binary. As such, the host CPU code containing the main software loops, file I/O management, user interface management, and final rendering/screen compositing was compiled using the stable branch 5.4.x of GCC. These parts of the SVS prototype application were obviously not related to FPGA PL offloading, and therefore did not target any performance optimizations. Device code targeting the dataflow engine to be synthesized on PL was annotated following the advice and recommendations of BSC's researchers. Typically, the code of the selected algorithms (see the Appendix) were implemented using nested loops, and therefore parallelized using `#pragma omp taskloop` and `#pragma omp for` directives.

The source code of a lightweight work-in-progress SVS application (`demographics`) is also available on the AXIOM project GIT server. When compiled, it works with any input video file container format supported by LibAv or URL address of a remote RTSP surveillance camera. The application has been successfully compiled and tested on both an x86-64 Linux workstation and on a Xilinx ZC706 board.

In order to simplify the parallelization process to the maximum extent and minimize errors, it was decided to target first during the second year SMP architectures (i.e. x86-64, ARMv7 and ARMv8) with OmpSs annotations (i.e. `#pragma omp target device smp`) to ensure the correctness of the parallelized kernel code. It is expected that once the interaction between OmpSs@FPGA and Nanos++ runtime for the management of FPGA memory data transfers is fully completed (i.e. data-copy clauses) during the third year, RTL code targeting the FPGA PL will be automatically generated. As it was already discussed in deliverable D4.1, this stage of the design flow will be implemented by calling the Xilinx Vivado HLS compiler simply by replacing the `#pragma omp target device smp` annotation with the `fpga` target. Finally, the bitstream generation of kernels will be implemented using scripts developed for automatically calling Xilinx's synthesis tools available in the Vivado suite.

Scalability among several AXIOM boards will be achieved during third year by means of the Nanos++ runtime and OmpSs@Cluster, as it has already been demonstrated for a matrix multiply example in deliverable D4.2. As it is well-known, the matrix multiply operation is also a requirement for implementing convolutions and fully-connected layers when inferencing CNNs.

Regarding the main binary generation, the isolated kernel source code `.c` and `.cpp` files were compiled with OmpSs/Mercurium version `2.0.0 713c99c` using the `--ompss` and `--instrument` command line options for generating the required object files, linked against Nanos++ version `0.10.3`, and with the remaining sequential modules plus the required third-party libraries. This final linking step was performed also using BSC's Mercurium compiler.

2.5 Profiling and parallel scalability

Initially, the selected kernels were studied in an isolated manner with minimal test applications using the same input files (e.g. same input pictures for both LBP face classifier / YUV2RGB kernels, and same matrices for CNN inference). The validation and correctness of the parallelized kernels were guaranteed by matching the obtained results against expected output files, which were previously computed using the sequential CPU version of the same algorithms (cf. D7.1 Section 6).

Parallel scalability on the SMP target platforms were studied by launching the test applications and subsequently varying the `NX_SMP_WORKERS` environment variable between executions, effectively iterating multiple runs ranging from 1 to the maximum number of cores available in the underlying platform (i.e., 12 logical cores in the case of the Intel Core i7 desktop platform featuring hardware-enabled SMT, 2 physical cores in the case of the Xilinx ZynQ-7000, and 4 physical cores on the Xilinx Zynq UltraScale+ ZU9EG). Finally, hand-coded SIMD vectorization on CPUs was discarded as a CPU baseline for the LBP face classifier and YUV2RGB kernels. Even though these kernels offer opportunities for hand-crafted SIMD parallelization, this was not an objective pursued by the AXIOM project as the parallelization efforts are managed by the OmpSs programming model and runtime.

2.5.1 Convolutional neural network inference

CNN inference was a special case, when compared to the other kernels. Even though it is possible to implement parallel naïve CPU versions of the layers involved in both gender and age estimation CNNs

with nested loops, this approach is not computationally efficient as it involves low-level matrix multiply operations. Matrix multiply on CPUs is nowadays a well-studied problem, and it is usually implemented with BLAS libraries optimized by hand at the assembly level with SIMD instructions. As such, we selected the open-source ATLAS library [13], and relied on the SGEMM operation as the low-level CPU backend, even though at the high level the code was annotated with OmpSs directives. In the future, we plan to use also as a CPU baseline the GEMM-lowp [14] library recently released by Google, as it includes support for 8-bit matrix multiply operations (used in quantized/pruned CNNs), and it is optimized for ARMv7 and ARMv8 platforms at hand using assembly NEON instructions. This latter library is a more suitable baseline for comparing the CPU versions of the implementation of CNNs layers running on the ARM cores available on the FPGA against the computations offloaded to the PL.

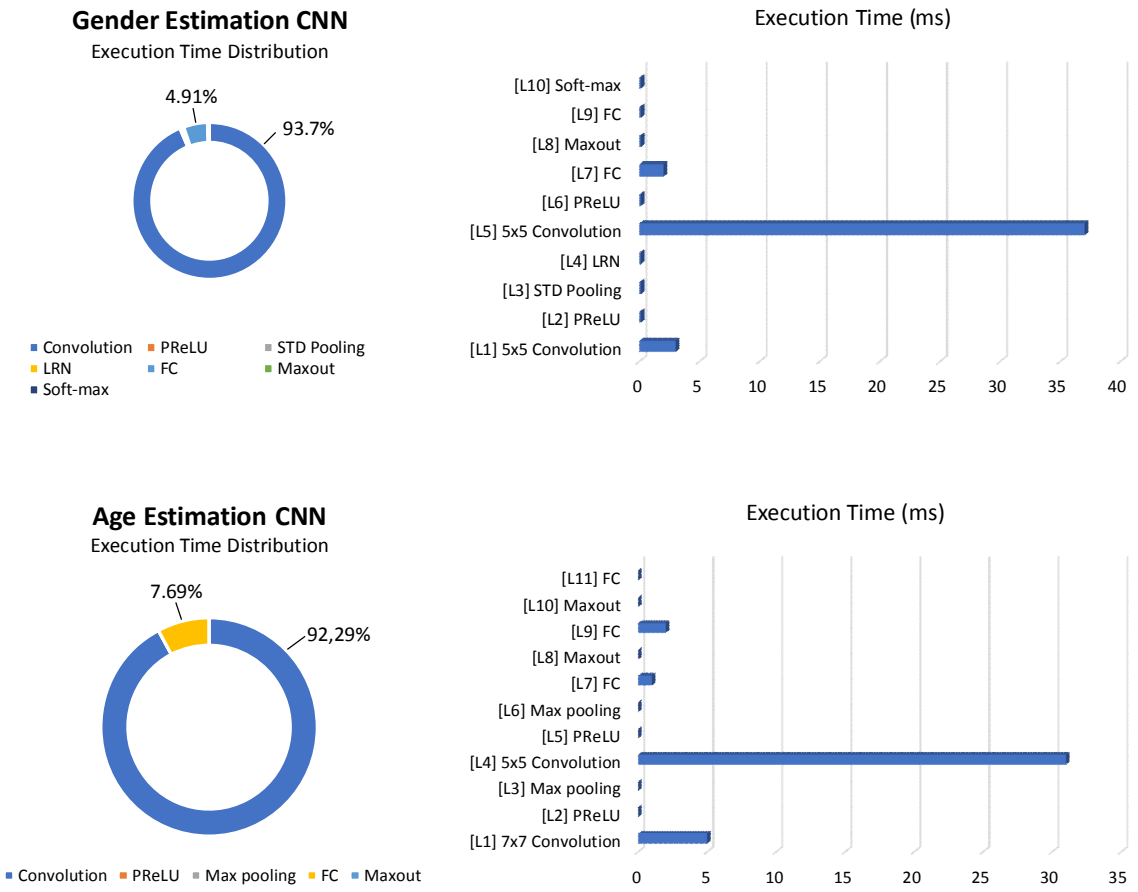

Figure 5. *Execution time distribution of the different layers of the trained CNNs*

Due to the importance of CNN inference for the SVS application, the different neural network layers were profiled for both trained CNN models to determine the most time-consuming layers (see execution times in Figure 5). These benchmarks were conducted by compiling the CNN inference engine to target the x86-64 architecture, and executed on an Intel Core i7-4770 3.4 GHz microprocessor using as an input a single face. The BLAS library used for these tests was the ATLAS library version 3.10.2, which as opposed to the recently-released Intel MKL-DNN [15] library, only has support for old SSE2 SIMD vector extensions. Nevertheless, this fact was not a major drawback for an initial code hotspot analysis.

The obtained performance results included below depict the latency in milliseconds per layer, and also the inference time distribution across layers for each CNN model.

As the obtained results show, it is only worth to offload to the PL of the FPGA convolutions and fully-connected layer computations. These results are in line with inference time distributions reported in earlier works found in the literature for other network architectures such as VGG-16 [1]. As it is also known, those layers are internally implemented using linear algebra by means of the abovementioned SGEMM operation available in the BLAS specification. The source code of the convolve operation used in the CNN inference engine is included as a reference in the Appendix. The implementation annotated with OmpSs directives, and an in-depth analysis of a naïve matrix multiply kernel working on the PL of the Zynq-7000 SoC was conducted by BSC researchers. Kernel source code, and preliminary results for this study are detailed in sections 2.4.1 and 4.1.2 of deliverable 4.2.

Unfortunately, more sophisticated alternatives for speeding up convolutions on the AXIOM board require a low-level access to the DSP48E2 slices found on the UltraScale+ architecture. These optimizations were proposed by Xilinx during 2016 in the whitepaper *Deep Learning with INT8 Optimization on Xilinx Devices* [15], and basically rely on packing two INT8 operations in the multiply-accumulate unit available in the DSP48E2 slice. However, this optimal solution requires to reschedule HERTA's porting efforts from software development and analysis to retraining and pruning the CNNs models. The challenge is to retrain the CNNs using low-precision 8-bit integers rather than single-precision floating point arithmetic while keeping a similar ROC accuracy.

When these pruned CNNs are ready, further potential low-level optimization proposals may be discussed with BSC for future integration in the OmpSs@FPGA framework.

2.5.2 LBP face detector

The selected face detector kernel relies on boosted ensembles and a well-known image descriptor (i.e. LBP). The core algorithm of the evaluation of the classifier cascade basically consists of two high-level nested loops (see the pseudocode included in the Appendix). These loops are in charge of computing the boosted ensembles by relying on a sliding window, which is responsible for scanning a synthetic image pyramid generated from the input frame. This image pyramid is necessary for enabling the detection of faces of arbitrary size using a fixed-sized sliding window. On top of these loops, there is an additional one responsible for executing the face detector kernel for the multiple scales constituting the image pyramid.

Internally, the sliding window must evaluate all LBP features until a given threshold is violated (see Figure 6). As such, this algorithm yields a highly irregular control flow due to the fact that the iteration count of the inner loop depends on the characteristics of the input image. Therefore, if an input image does not contain any faces, the first LBP features of the cascade will soon violate thresholds, and the remaining loop iterations will not be completed thus dramatically speeding up the kernel. On the other hand, if the amount of faces appearing on a given image represents a high percentage of the total area of the input frame, the little opportunities available for early rejection of image regions will increase the execution time of the face detection kernel.

This unbalanced behaviour is common for any object detection cascade classifier based on the sliding window approach. As the sample depicted in Figure 6 illustrates, the processing latency in clock cycles of each image patch analyzed by the sliding window (right) is highly correlated with image regions containing faces (left).

Figure 6. *Execution time of a cascade of features (right) for an input image (left)*

From a coarse-grain parallelization perspective, a challenging scenario is considered as the baseline input for the experiments. As such, kernel parallelization efforts were conducted using as an input the same 1080p picture containing 25 faces. This ensures that a high percentage of image regions will reach the latest LBP features rather than being discarded too early on the first stages of the cascade.

The selected picture is available on the private AXIOM GIT server and is named `testImage.raw` for ensure reproducibility of the obtained results.

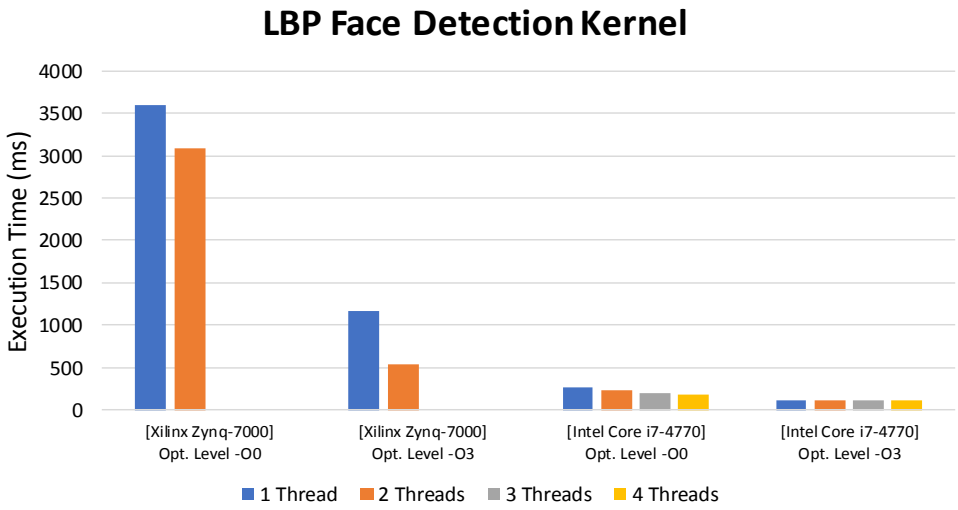

Figure 7. *Parallel scalability of the LBP face detection kernel annotated with OmpSs (please note that Zynq 7000 is a 32-bit platform while Intel Core i7 is a 64-bit platform)*

For the initial design space exploration and porting on SMP platforms, it was decided to create a task for each one of the LBP kernel calls required for evaluating the `NUM_SCALES` images constituting the synthetic input image pyramid. As the code included in the Appendix shows, it is only necessary to add a single `#pragma omp task` annotation preceding the header of the `LBPCascadeEvaluation` function implementing such kernel. When the execution of the kernel concludes, a `#pragma omp taskwait` directive is also necessary to ensure synchronization and the correctness of parallel execution. Even though that further low-level optimizations are possible, this simple parallelization strategy succeeded in obtaining a 1.16x speed up on the dual ARM A9 CPU cores of the Zynq-7000 platform

when the number of worker threads was increased from one to two. Similarly, a 1.44x speed up was observed when increasing the number of worker threads from one to four on the Intel Core i7-3400.

It should be noted that these results were obtained when the LBP kernel was compiled without enabling aggressive compiler optimizations. In order to determine the potential effect of such optimizations, the kernel was recompiled again using the `-mfpu=neon` option in the case of the ARMv7 target, and optimization level `-O3` on both x86-64 and ARMv7 (32-bit) targets. These optimizations significantly reduced the execution time on the ARM A9 cores (see Figure 7), and yielded a 3.06x speed up when compared to the unoptimized `-O0` baseline OmpSs kernel code running a single worker thread. When the worker thread count was increased to two, it yielded a 6.59x speed up against the `-O0` version, and a 2.15x against the `-O3` optimized version, respectively. The speed ups observed in the `-O3` optimized versions of the executed on the Intel Core i7-3400 platform were almost negligible. This latter effect was probably related to the aggressive prefetching, branch prediction, and out-of-order execution capabilities available on the x86-64 cores when compared to the simpler in-order ARM A9 cores. It has also be stressed that the typical power consumption of the Zynq-7000 is about 5W while the Core-i7 typically uses a 100W (more precise measurements are also in progress).

Experiments enabling more than two worker threads on the dual ARM A9 cores were not possible, as the main application crashed and the bug could not be successfully tracked. However, it is expected that further increases of the worker thread count beyond two would not bring any particular benefits on the dual core platform when both cores were running at 100% of usage while executing the kernel.

2.5.3 Color conversion

Another important step involved in the SVS prototype application is video display to end-users. The main kernel involved in this task is the one related to color space conversion (`YUV_to_RGB`). Two different versions of this kernel have been implemented (i.e. using floating point (FP32) and integer (INT32) operations). The main idea behind this decision was to determine if the type of operations substantially impacted on performance. As it is shown in the Appendix, the color space conversion kernel was parallelized using the OmpSs programming model simply by adding a single annotation (`#pragma omp taskloop grainsize(16) private(i,j,y,u,v,r,g,b)`) on both FP32 and INT32 implementations.

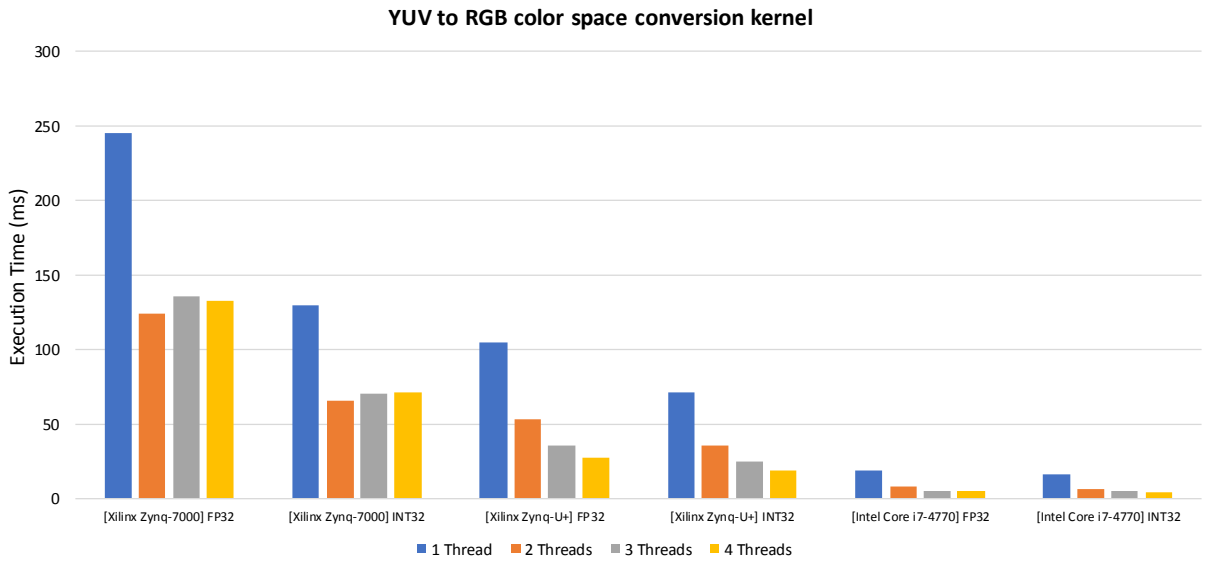

Figure 8. *Parallel scalability of the color space conversion kernel annotated with OmpSs*

The initial results of such parallelization on CPU cores are shown on Figure 8. As it was initially expected, the execution time scales with the number of cores available on the underlying platform (Zynq-7000, Zynq UltraScale+, and Intel Core i7-4770) as the number of worker threads are increased. In the case of the Zynq-7000, since the SoC features only two ARM A9 cores, creating more than two worker threads even degrades performance substantially. On both quad-core platforms (UltraScale+ and Intel Core i7-4770) execution time is further reduced when the worker thread count equals four. It should be noted that the highest speed-ups (2x) are obtained when the number of worker threads is set to two, and then performance improvements marginally diminish as the worked thread count is further increased. Again this comparison is done only taking into account the pure performance and not the global power consumption which is about 5W in the case of Zynq- Ultrascale+ while it is about 100W in the Intel Core i7.

On the other hand, since the final color picture quality is quite similar if the low-level kernel implementation is switched from FP32 to INT32 operations, extra reductions in the execution time are possible simply by relying on integer operations. This improvement in performance related to INT32 operations was quantified, and on average provided an additional 15% benefit.

3 Smart Home/Living Scenario

The SHL case study implements a solution to enhance the security level and comfort of homes. The developed solution consists of a system that can analyze multimedia streams broadcasted from specific points inside and outside of smart homes. This system receives multimedia streams broadcasted from devices attached to the network. Then demuxes and decodes audio and video streams, and analyzes raw data using machine learning algorithms to gather precious information. The information extracted from multimedia streams are subsequently processed to define the feedback that will be finally reported to the end-users living in the smart home environment.

The main goal of the SHL scenario developed in the context of the AXIOM project is to achieve a high level of home automation, and to allow a natural interaction between end-users and their homes. To achieve this goal, it is required not to interrupt the end-user action flow while analyzing data, and to generate the feedback very rapidly. This challenge represents a strict timing constraint in the envisioned architecture of the SHL scenario.

In order to take advantage of the heterogeneity and the cluster architecture developed for the AXIOM project, the SHL software application extensively relies on OmpSs directives. Several OmpSs@SMP solutions have been explored with the aim of defining both the tasks that can be concurrently executed and the granularity required to satisfy the performance targets of the SHL application. The results of this analysis of the multi-core system will be used for the actual development of the FPGA and the cluster to exploit the resources of the AXIOM board. In this way, OmpSs@FPGA directives will be used to efficiently synthesize the most time-consuming sections of the selected kernels on FPGA PL resources. After kernels are properly implemented in PL, OmpSs@Cluster will be used to split and parallelize the execution of the application in different nodes of the cluster. This architecture is designed to meet real-time constraints, and to minimize hardware resources while keeping a low power consumption target.

3.1 Software architecture

The SHL software application was designed and developed during months m12 and m22. It required to analyze several open-source audio and video processing frameworks, libraries, toolkits and algorithms. As such, the SHL software was developed with high-modularity in mind in order to simplify the testing, profiling and optimization phases. The main blocks that characterize the developed SHL application are divided into on-line blocks and off-line blocks. On-line blocks process data at run-time, so they need to meet real timing constraints. On the other hand, off-line blocks generate the required models used in machine learning algorithms, and do not suffer from time constraints.

- **ON-LINE BLOCKS**
  - **Input block:** It gathers multimedia data from the network, demuxes and decodes the audio and video data. The input block also manages the FIFOs required for storing data. This block is based on the GStreamer open-source framework.
  - **Trigger block:** It recognizes the event enabling the start of the identification phase.
  - **Speaker identification block:** It processes the audio track recorded. Then it extracts features from the input audio, and compares them with probabilistic models for speaker identification (i.e. to grant/deny people's access into their home premises).
  - **Iris recognition block:** It processes all input frames, and locates the position of eyes inside images. If the eyes were found, it then performs the extraction/generation of the iris code of located eyes, and later compares it with the iris code of enrolled users.

  - **Fusion block:** It performs the biometric fusion of both the iris recognition and speaker identification processes. It also defines the feedback action.
- **OFF-LINE BLOCKS**
  - **Speaker training block**: It trains the target speaker models and organizes the underlying file system.
  - **Iris training block**: It trains the iris code of the target and organizes the file system.

As it has been pointed out, the on-line blocks characterizing the SHL application are mainly the speaker identification block and the iris recognition block. These two blocks are presented in the next section.

3.1.1 Speaker identification block

The speaker identification block was developed during months m12-m16. It is based on ALIZÉ [16], an open-source platform for speaker recognition developed in C++, and on SPRO [17], an open-source speech signal-processing toolkit developed in C. Figure 9 shows the architecture of the speaker identification block. This block processes a chunk of audio captured from the input block. Then it decides if inside the audio stream there is a human voice signal, and finally extracts the voice sample. If the voice signal has a good quality, biometric features are extracted and then matched against the models of previously enrolled subjects. The result of these computations is used to identify whether the speaker is authorized or not to perform the operation requested (i.e. basically, to deny/grant access to the home).

The main steps of the speaker identification block are:

- Extracting a set of features from input audio data.
- Identifying and normalizing the features that describe human voice.
- Matching the extracted features by relying on probabilistic models (Gaussian Mixture Model GMM) against a database of pre-enrolled persons. This list of persons must be able to access their home premises using biometric authentication. The probabilistic models of these individuals are generated with the help of the off-line training block.
- Normalizing the results of the compare/matching operation using two different methods. The output of the methods is used for determining if the processed audio matches an enrolled person.

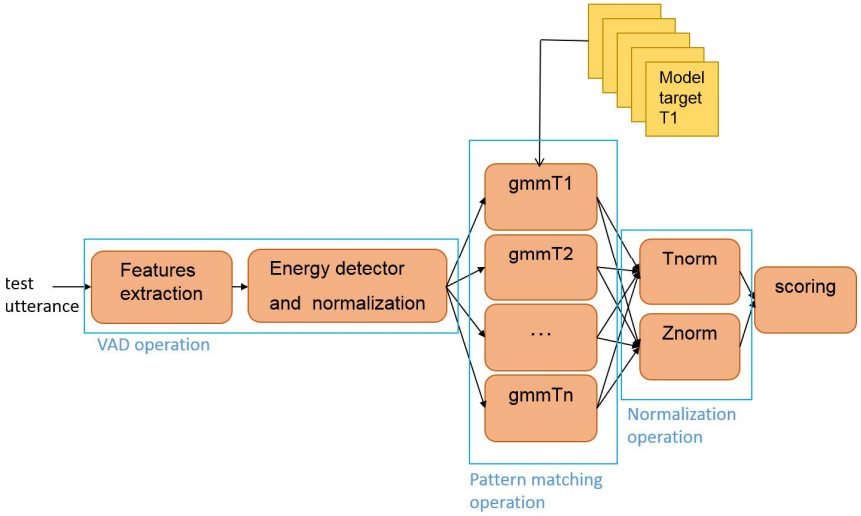

Figure 9. *Schema of the speaker identification blocks.*

3.1.2 Iris recognition block

During months m17-m20 the iris recognition block was designed and implemented. It is based on OSIRIS [18], an open-source iris recognition software and relies on OpenCV [19] , also an open-source computer vision library. Figure 10 shows the architecture of the iris recognition block. From a high-level perspective, this block is subdivided into two additional low-level blocks:

- A Video Processing Module (VPM)
- An Iris Recognition Module (IRM)

The VPM processes all the frames gathered from the input block. This module identifies and extracts the coordinate of the Regions Of Interest (ROIs), which is the minimal box that incorporates an eye inside the frame (i.e. Eye Region Detection block, and Eye Region Extraction block). ROIs are processed to extract the metrics required for determining the image quality (i.e. Image Quality Assessment block). Then metric values are tested against a threshold range in order to determine if the image quality meets the minimum standards. Basically, this process checks if the regions enclosing eyes were captured using correct light conditions and sharpness using previously fine-tuned thresholds. Finally, ROIs are sent to the IRM. However, if they violate image quality thresholds, they are discarded (i.e. Image Selection block).

The IRM processes the input ROI to generate the iris code of the previously found eyes. This module is also used in the iris training block (off-line block) when computing the iris code for the access control of subjects into their home. Inside the IRM, the inner and the output boundaries of iris are detected (i.e. Segmentation block). Thereafter, the iris annulus is transformed into a size-invariant strip, following Daugman's rubber-sheet method. This new image is then filtered to extract iris features, and finally, an *iris code* is generated as a selection of the complete set of features. To conclude the pipeline, the matching phase consists in finding the distance from the processed iris codes to the codes saved in the enrolled subjects database. The final match decision is based on meeting a minimum threshold distance.

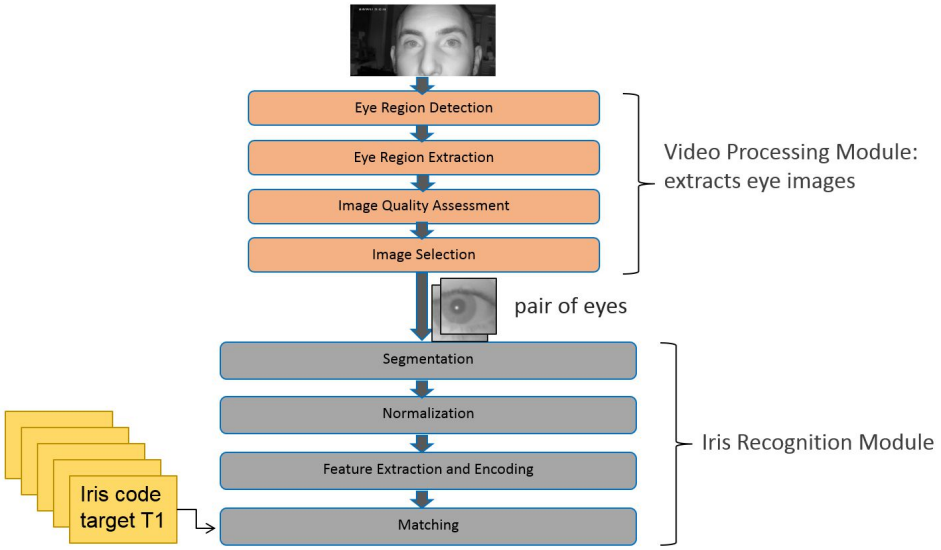

Figure 10. *Schema of the iris recognition blocks*

3.1.3 Application workflow

The final integration of the multiple abovementioned blocks, and the application workflow code was developed and extensively tested during months M20-M22. The obtained SHL software processes the flow of the input audio and video streams in order to grant/deny access by performing matches against the database of pre-enrolled subjects (Figure 11).

The recognition phase starts after the identification of a trigger event. After this event is captured, the application starts to analyze in sequential mode multimedia streams. The audio track is processed only after having recorded a voice sample of a pre-defined length. This sample was set to 3 seconds in the proof of concept tests. On the other hand, video processing starts as soon as a frame is broadcasted. Figure 12 shows an example featuring the input data processed over time by the SHL application.

However, this simplified representation does not show the latency required to process different video frames, as this depends on the characteristics of the frame currently analyzed. The number of frames required to recognize the iris of a given person is closely related to the degree of cooperation of the user with the video capture system. If the end-user is not cooperative, it could become very difficult to capture pictures with the adequate quality and angles for enclosing eye regions.

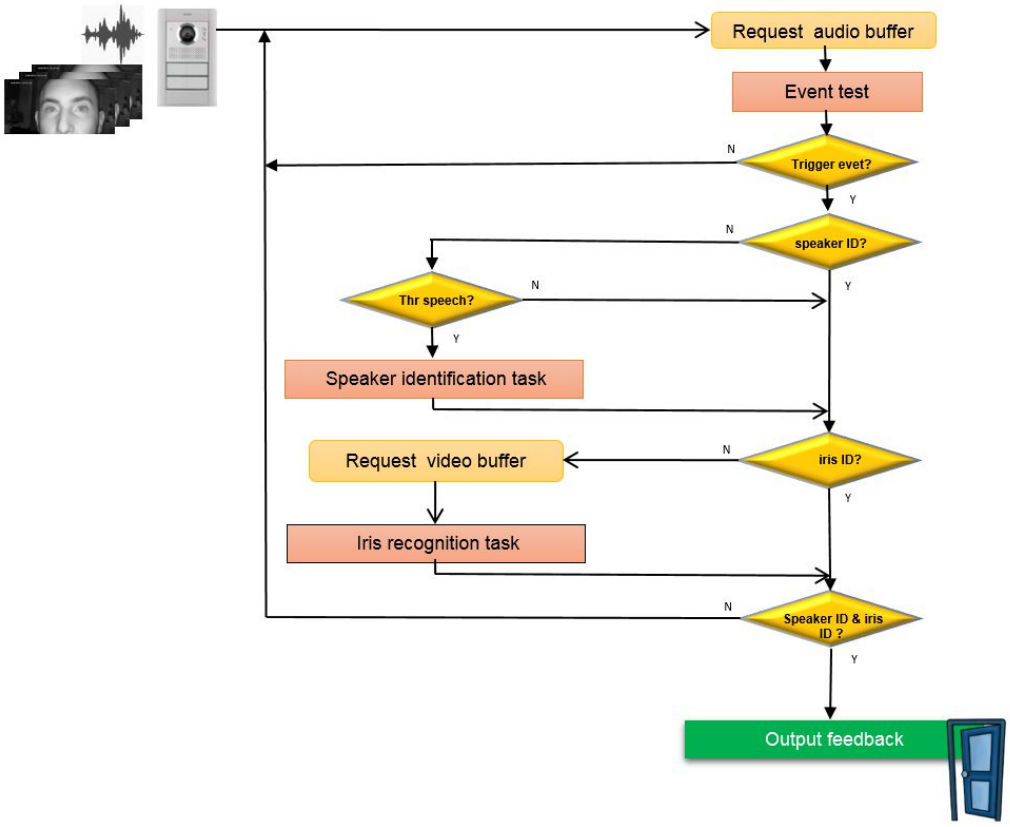

Figure 11. *The SHL application workflow*

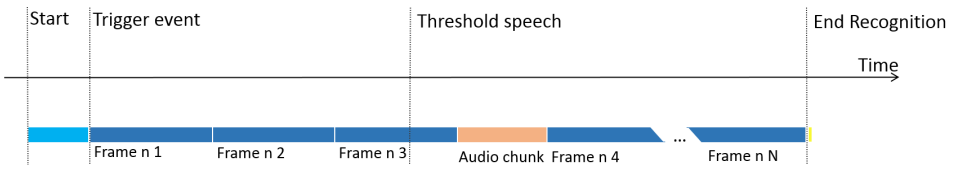

Figure 12. *Sequential approach for the SHL application*

3.2 Performance target goal and initial baseline performance

The SHL application checks the user's identity before enabling his/her command request to the smart home system. In order to enable a natural interaction, this operation must be as fast as possible so it cannot degrade or interrupt the smoothness of the user action flow.

The amount of time required for the end-user identification is strongly correlated to the degree of co-operation the end-user has with the SHL system. Similarly, the quality of multimedia data recorded also has an impact on the recognition latency. Therefore, the time required to process an audio sample and a given frame is affected by these unpredictable issues. Table 2 presents the initial baseline performance results. The input workloads used for the experiments were the following:

- An audio sample of 3 seconds (PCM format) with 16-bit depth and sampling rate of 16 KHz
- A video stream in Full-HD resolution (1920x1080) with ROIs of 260x260 pixels

The first three lines of Table 2 are related to the three main steps of the speaker identification process:

- Feature extraction (with the algorithm presented in next section)
- Voice activity detection and feature normalization
- Pattern matching recognition of the features with the target models (GMMs)

The last three lines of Table 2 are related to the processing time of an image with three different types of complexity:

- In the first case, the image does not incorporate any ROIs, and the frame is discarded after the VPM process. The process time includes VPM processing time latencies but not IRM processing time latencies.
- In the second case, the image incorporates a single ROI. The processing time includes VPM and IRM processing time latencies.
- In the third case, the image incorporates two ROIs, so processing time includes VPM and two IRM processing time latencies.

All the measures presented in Table 2 were obtained after averaging the latencies of 10 executions of the final SHL application. All benchmarks were performed on the AXIOM Evaluation Platform (AEP) (cf. D7.1, D7.2) equipped with the Xilinx-Zynq7000 SoC. The application was compiled with `g++` (version 4.8.2), and enabling the `-O2` compilation flag.

Table 2. Execution time of the SHL scenario on the AEP.

Operation	Execution time
Feature extraction in a 3-seconds audio sample	0.2 sec
Energy detection and normalization	0.15 sec
Compute speaker pattern matching (with 9 GMMs)	0.67 sec
Process a frame without ROIs	0.27 sec
Process a frame with 1 ROI	3.7 sec
Process a frame with 2 ROIs	7 sec

On the original AXIOM proposal, it was mentioned that the video decoding process must be implemented using a third-party IP core on the FPGA reconfigurable logic. As such, the implementation of the interconnection of the video decoder with chip memory and ARM cores requires a close collaboration between SECO and VIMAR partners.

The proposal was written with the hypothesis that it was feasible to decompress H.264 video streams on the Xilinx Zynq-7000 SoC. As it is already known, this SoC does not include an on-die hardware video decoder. Additionally, H.264 software video decoding on the ARM A9 cores yields a very low performance, and it is thus unable to sustain the required frame rates for real-time applications. This fact has been independently confirmed by both VIMAR and HERTA partners with initial tests conducted on the ARM-A9 cores of the Zynq-7000 platform.

However, the recently released Xilinx Zynq UltraScale+ SoC was finally selected for the first AXIOM prototype board. This change in the original board specifications had an impact on the initial hypothesis and project's planning efforts. The Xilinx Zynq UltraScale+ SoC includes a quad-core ARM Cortex-A53 clocked up to 1.5 GHz, and its EV family includes an on-die H.264 / HEVC (H.265) high-performance video decoding engine. For this reason, VIMAR and SECO decided to cancel the planned purchase of a third-party H.264 IP video decoder block, and the corresponding development of the logic within the FPGA to manage it. The final decision was also motivated by considering this latter approach a very uncompetitive solution for the market. The first prototype of the AXIOM board is based on the EG device family, which do not integrate the video decoder. The EV family was still not in production on silicon at good yields, and thus was not available for the first board prototype. For these reasons, the initial prototypes of both SHL and SVS applications running on the AXIOM board will perform software video decoding on CPUs.

3.3 Optimization process using the OmpSs programming model

On deliverable D3.1, it was hypothesized that the SHL application implemented in WP3 would rely mainly on three high-level kernels. These kernels are summarized again in Table 3. The first two kernels are related to audio processing, and the latest one to video processing. All these benchmarks are related to machine learning workloads.

Table 3. *Proposed kernels representing the SHL scenario workloads.*

Kernel Name	Description
`Voice_Activity_Detector`	Algorithm to label speech frames
`Speaker_Recognition`	Pattern matching for speaker identification
`Iris_Recognition`	Pattern matching for iris recognition

At the time when the deliverable D3.1 was written, the information on these algorithms was limited. The initial exploration and the development of the SHL solution showed that the iris recognition task required a higher workload than the speaker recognition task. More efforts to speeding up video processing while reducing them on the optimization of the audio processing while be carried out in the next task T3.3 while testing for user experience.

The benchmarks that were studied are included below:

- The Voice activity detection (VAD) task, in which the focus is feature extraction algorithms, (widely used in the speech processing application).
- The Anisotropic smoothing task, which is a very time-consuming process when performing the segmentation steps of the iris recognition blocks. Additionally, this task is also used to filter the input image before finding the inner and output boundaries of the iris ring.
- The Iris recognition task, which is the main process conducted in the Iris recognition module. This task includes a part of the ROIs selection and all the processing of these.

3.4 Experimental setup

The results presented in next sections were obtained by compiling the SHL modules with the GCC/g++ compiler version 4.8.2, Mercurium compiler version 2.0.0, and Nanox++ run-time library version 0.12a. The optimization level used by the compilers was set to `-O2` for all benchmarks. OmpSs directives were added to the code to exploit the underlying SMP resources of the AEP.

These results are basically the average of 10 executions on the AEP. Additionally, they show the speed up between the original code, and sequential/parallel annotated code compiled and managed by OmpSs.

The code featuring OmpSs annotations was executed on the AEP by setting up SMP resources with the `NX_SMP_WORKERS` variable. The exploration was done with 1 or 2 worker threads, as these are the number of SMP resources available in the Xilinx- Zynq7000 SoC (ARM Cortex-A9). The number of worker threads were not set to more than 2 to avoid overload and performance degradation.

For a more detailed analysis, traces were recorded using the Extrae library version 3.4.1, which enables code instrumentation. Additionally, the traces were visualized with BSC's Paraver tool. These tools were used to analyze parallel code and several screenshots are shown during the text to prove its use-fulness.

3.5 Feature extraction module

The feature extraction in speaker identification and in speech recognition consists in transforming the speech signal into a set of feature vectors. The aim of this transformation is to obtain a new representation that is more compact, less redundant, and more suitable for a statistical modelling and a calculation of a distance or any other kind of score. The features representation (or speech parameterizations) used in the speaker identification block is a cepstral representation of the speech. Figure 13 shows a modular representation of the algorithms used in the SHL application. The code of this module was extracted using the SPro toolkit. SPro is an open-source speech signal-processing toolkit which provides runtime commands implementing the standard feature extraction algorithms for speech and speaker recognition applications and a C library to implement new algorithms [17].

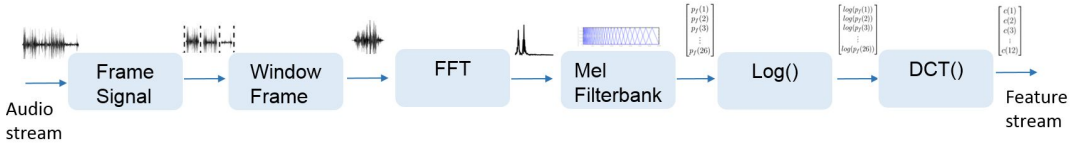

Figure 13. *Pipeline architecture of the Cepstrum analysis algorithm.*

The SPro toolkit code related to the cepstrum analysis is included in the `sfbcep` function. Such function is written to implement a sequential code and it does not take advantage of the heterogeneity and the multicore architectures of the AXIOM system.

The exploration done on this module has defined which tasks of the algorithm can run concurrently and introduce the OmpSs directives to take advantage of the multicore architecture and the FPGA resources.

The exploration results have revealed the possibility to process windows of 20 ms of input audio data in parallel.

To process the input data in parallel, the SPro `sfbcep` code has been slightly modified. The two main changes are:

1) Convert a set of global variables into local variables;
2) Modify the architecture of the application to extract in parallel the input audio windows from the FIFO input and to write in parallel the output feature vectors on to the FIFO output. The FIFO input and the output are used from the original code to read/write data from/to files.

Figure 14 shows SPro `sfbcep`'s architecture with the input and output FIFOs represented with "`is`" and "`os`" in the figure. Figure 15 shows the new architectures that enable parallel access to data: the input and the output data are saved on two vectors (`buffArray[]` and `cArray[]`). The green lines in Figure 15 indicates the tasks that can be created with the OmpSs directives, obtaining parallel processing. The pseudocode of the feature extraction function with the OmpSs annotation is presented in the Appendix.

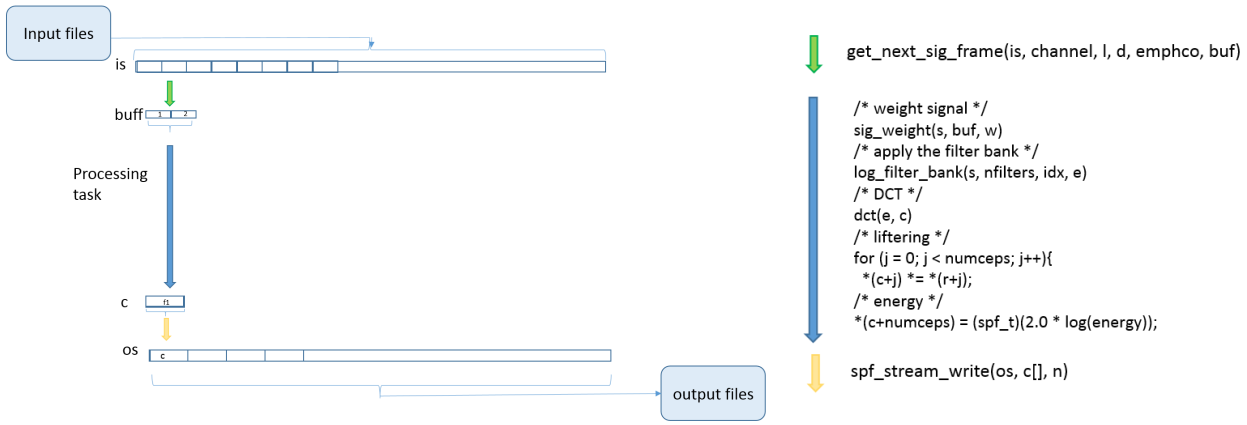

Figure 14. *The original sfbcep sequential program.*

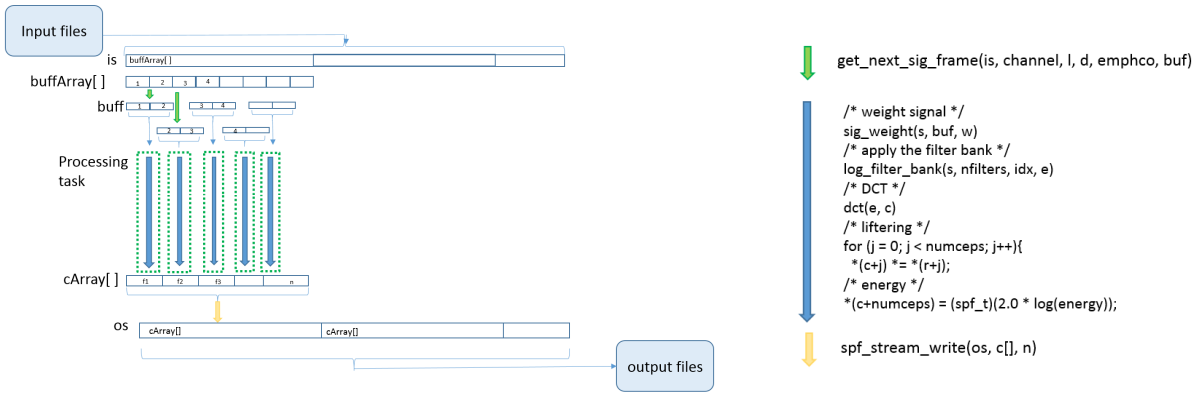

Figure 15. *The proposed sfbcep parallel program.*

3.5.1 Trace of the task execution and obtained performance results

Figure 16 shows the trace of the feature extraction module processed with two worker threads. The two horizontal rows of the graph are the threads running the OmpSs tasks. The different colors mean different thread states along the execution time of the application. The red blocks show the main task, therefore the execution time consumed for the instructions outside the OmpSs pragma on the pseudocode.

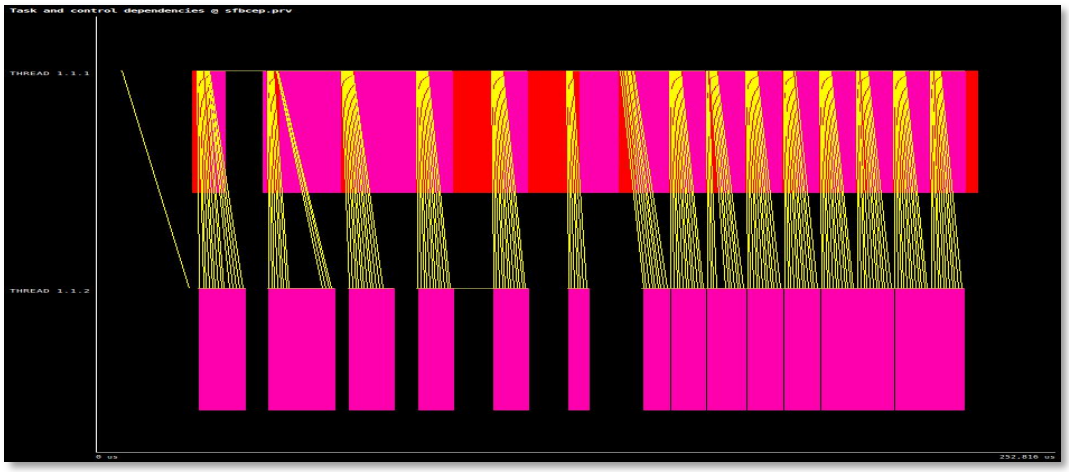

Figure 16. ***Paraver trace of the sfbcep program using 2 threads of the AEP.***

The magenta blocks show the OmpSs tasks, the execution time consumed on the instructions inside the OmpSs pragma, and on which worker threads the tasks are processed. The yellow lines show the scheduling operation of the tasks. By the yellow lines is possible to see where the `buffArray[]` vectors are created and scheduled and where the `taskwait` pragma are set.

Table 4 shows the execution time of the feature extraction module on the AEP. The input data processed are three audio samples with different duration. The audio format is uncompressed PCM audio with bit depth of 16bits and sample rate of 16 KHz.

Table 4. *Execution times of the feature extraction task on the AEP.*

Input audio [sec]	Execution time without OmpSs pragma [sec]	Number of OmpSs tasks	1 worker thread		2 worker thread	
			Execution time with OmpSs@SMP [sec]	speedup	Execution time with OmpSs@SMP [sec]	speedup
3	0.18	299	0.19	**0.9x**	0.12	**1.5x**
30	1.8	2999	1.9	**0.9x**	1.01	**1.7x**
60	3.6	5999	3.8	**0.9x**	2	**1.8x**

Speedup smaller than 1x (i.e., a slowdown) appears in all the cases on which the application enriched with the OmpSs directives uses only 1 worker thread. This expected result is attributed to the inability to exploit parallelism with 1 worker only and the overhead is due to the creation and synchronization of tasks that is not present in the sequential execution. Speedup up to 1.8x is gained when parallelism is exploited with 2 worker threads. In this scenario, tasks are processed concurrently on the two SMP resources.

A further increase of the speedup is expected on the final AXIOM board, in which four ARM cores are present, and parallelism of tasks have further opportunities for exploiting the increase in the core count. Task offloading to the FPGA PL using OmpSs@FPGA could also reduce the execution time considerably.

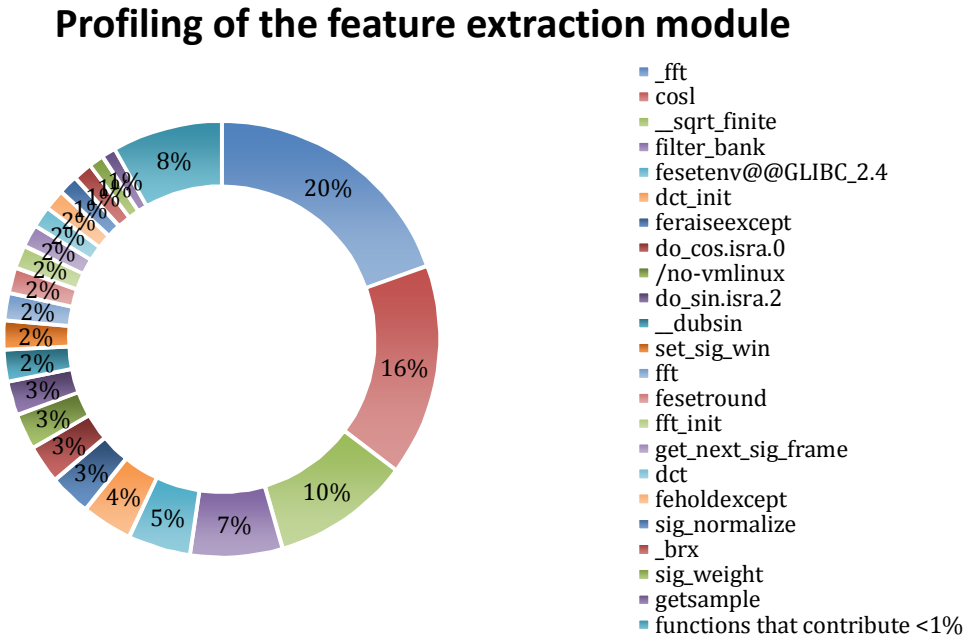

Figure 17. *Profiling of the feature extraction module. The profile was done on the AEP using the operf and the opreport tools.*

Profiling information of the feature extraction module depicted in Figure 17 shows that the Fast Fourier Transform (FFT) is a very time-consuming part of the application. Moreover, the pipeline structure of this algorithm pushes our interest to try to offload the algorithm as much as possible to the FPGA PL resources. This mapping operation to FPGA resources requires to define data formats and internal operations in a suitable manner to enable a correct and efficient synthesis of the algorithms. This exploration is carried out in collaboration with BSC's researchers and leverages OmpSs@FPGA and Xilinx Vivado HLS for automating such task.

The feature extraction module was also used to explore the possibility to parallelize with OmpSs annotations the GStreamer open-source project, which is a popular framework to handle multimedia streams. During last year, VIMAR in close collaboration with BSC explored the possibility to speed up a GStreamer plug-in using both OmpSs@SMP and OmpSs@FPGA targets. This study proved the feasibility of using the Nanox++ runtime system to manage threads created within the GStreamer framework using OmpSs. These results were presented at the GStreamer Conference 2016 [20] in Berlin together with the AXIOM project and the OmpSs programming model.

3.6 Anisotropic smoothing module

The Anisotropic Smoothing module is an image denoising technique that is aimed to preserve the edges of images while smoothing regions of uniform intensity. This type of filtering is usually used as a preprocessing stage of segmentation algorithms. As such, the Anisotropic Smoothing task is a very time-consuming workload required for the segmentation steps of iris recognition blocks. This task aims at filtering the ROIs recognition in frames in order to retrieve the iris contours. ROIs are filtered several times to retrieve precise contours and coarse contours that improve the accuracy of the normalization circles. Figure 18 shows an example of this filter.

Figure 18. *Example of execution of the anisotropic smoothing module. The original sample image is shown on the left side; the processed image is shown on the right side.*
The sample was processed with the anisotropic smoothing task with 100 internal iterations.

The original code of the anisotropic smoothing is based on the OSIRIS framework [18], which is developed in C++ and available in the AXIOM project GIT server (cf. D7.2). A minimal C++ program based on OpenCV framework [19] was developed with the purpose of obtaining a toolset for testing the module in several experiments while exploring level of parallelism in algorithms. The source code of

the module and the toolset were shared with AXIOM partners to guarantee the reproducibility of experiments.

The initial analysis of these algorithms showed the possibility of defining several tasks during image processing. Basically, these algorithms consist of a loop with *m* iterations in which all image pixels are processed in two steps. In the first step, only even pixels are processed, while in the second phase odd pixels are processed. Pixel values modified in the two steps are defined by reading the neighbouring pixels inside a 3x3 kernel, as shown in Figure 19. Image pixel are processed using two insert loops that read and write pixels by rows and by columns.

The underlying structure of the algorithm enables defining several tasks for processing pixels implementing different levels of granularity. For instance, granularity of tasks could be 3x3 pixels at kernel level, at row level or at image chunks level. An important constraint is that all even pixels must be processed before odd pixels and vice versa.

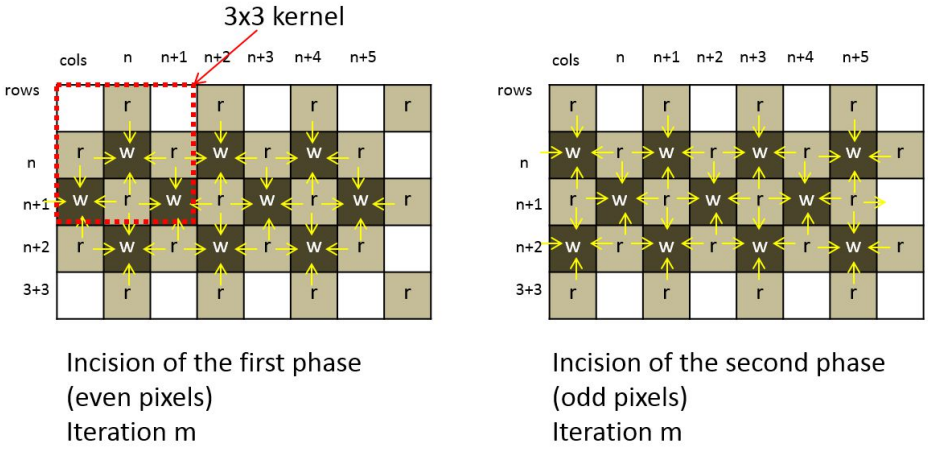

Figure 19. *Section of the image during the processing of even and odd pixels.*

We explored several strategies when adding OmpSs annotations in the source code:

- Create a task for each 3x3 kernel of pixels. If we do not consider the boundary exceptions, this strategy creates a number of tasks equal to the number of pixels for each iteration;
- Create a task for each row of the image. If we do not consider the boundary exceptions, this strategy relies on a number of tasks equal to the double of the number of rows of the image for each iteration, one task for odd pixels of a row and one for even pixels of the row;
- Create a task for processing half image pixels. This strategy creates 4 tasks for each iteration, two tasks for odd pixels and two for even pixels;

The design space exploration done for this module showed that the first proposed strategy yielded a number of tasks with low computational footprint. Moreover, performance using the SMP resources in the AEP was dramatically degraded when compared to the compilation without OmpSs directives.

The second strategy gave better results when pixels of a row were above of a given threshold. In fact, if the number of pixels processed are too low, the overhead introduced by task creation decreases performance. For this reason, the third strategy is the best one in cases in which the number of pixels in a

given row of an image are too few. The best solution strongly depends on the size of the image to be processed. The final code will take into account these results in order to maximize performance.

The pseudocode of the anisotropic smoothing function parallelized with OmpSs annotations is shown in the Appendix.

3.6.1 Trace of the task execution and obtained performance results

Figure 20 shows a chunk of the trace execution of the anisotropic smoothing function with two different granularities. The upper graph and the lower graph show two different tests. Horizontal axes of the two graphs are fixed to the same timescale. Therefore, the two graphs show the same time duration. In the trace of the upper part, granularity of tasks is fixed to the row level, whereas in the lower part granularity is fixed to half image level. Horizontal rows of the graph correspond to threads running OmpSs tasks. On the other hand, different colors mean different thread states along the execution time of the application.

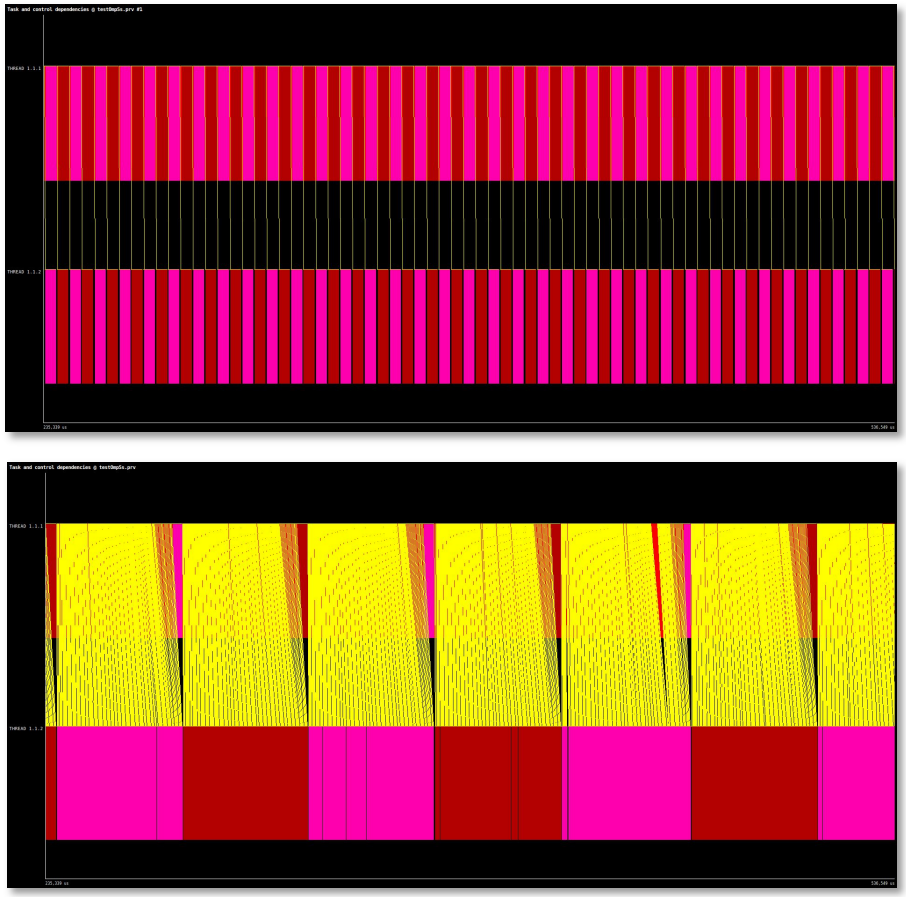

Figure 20. *Paraver trace of the anisotropic smoothing task with two different granularities with the same timescale in horizontal axes. The upper graph shows execution of the task with the granularity fixed to row level; the lower graph shows the execution of the task with granularity of half image level.*

Table 5 shows the execution time of the anisotropic smoothing module on the AEP. The input data sets for the experiments were two images: ROIs of 260x260 pixels obtained from the IRM, and a Full HD frame. The number of iterations *m* was fixed to 100. All internal calculations were performed using 32-

bit floating point precision. Figure 21 shows the obtained speedups with several number of pixels processed within tasks.

Table 5. *Execution time of the anisotropic smoothing task on the AEP.*

Input frame	Execution time without OmpSs pragma [sec]	Number of OmpSs tasks	Granularity	Number of pixels for tasks	1 worker thread		2 worker thread	
					Execution time with OmpSs@SMP [sec]	speedup	Execution time with OmpSs@SMP [sec]	speedup
260x260	1.479	51600	half image	258	2.401	0.6x	1.343	1.1x
260x260	1.479	400	row	33540	1.519	0.9x	0.833	1.8x
1920x1080	47.192	215600	row	1918	52.546	0.8x	26.702	1.8x
1920x1080	47.192	400	half image	1035720	47.324	0.9x	33.142	1.4x

Also in this module, a further increase of performance is expected to be achieved on the final AXIOM board, in which four ARM cores are present, and the parallel execution of tasks could efficiently exploit these additional resources.

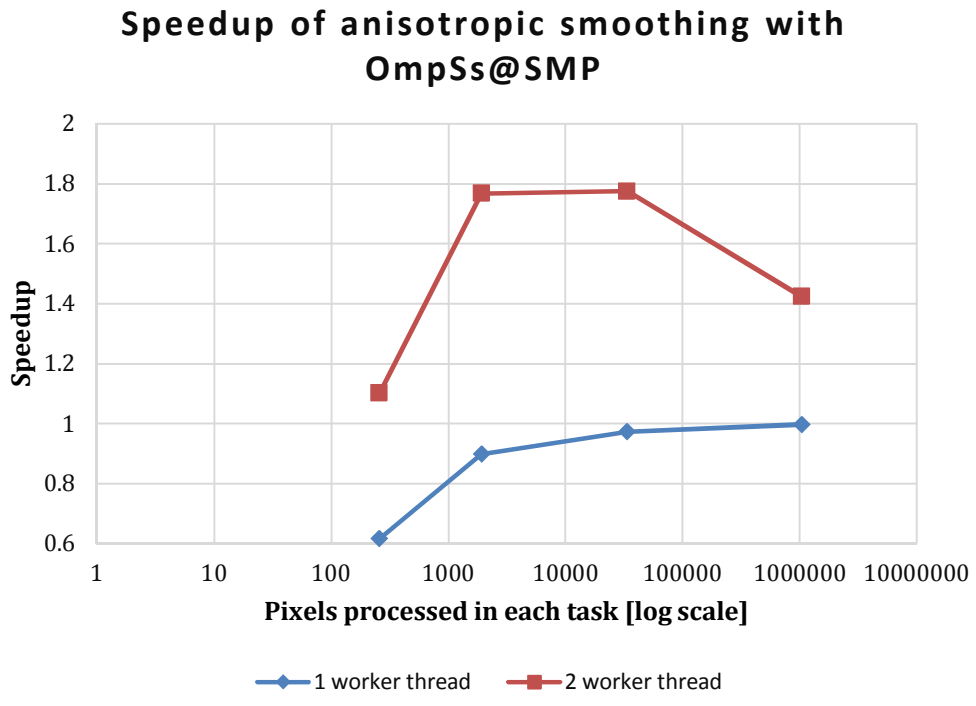

Figure 21. *Speedup of the anisotropic smoothing task on the AEP with 1 and 2 worker threads. Results are shown with several task granularities, fixed by the number of pixels processed on each task.*

3.7 Iris recognition module

The Iris recognition module is responsible for analyzing ROIs selected in the VPM of iris recognition blocks. As such, this module incorporates several processing steps presented in Section 3.1.2 related to the Iris recognition block. The original code of this module comes from the OSIRIS toolkit, and it is available on the AXIOM project GIT server. The module uses a large number of functions included in the OpenCV library [19]. Also, the OSIRIS code was modified and customized in order to be included it in the SHL application.

The analysis done on the SHL application shows that the data of each ROI extracted from the VPM module is independent, and can be processed using different tasks. The OmpSs programming model can transform the sequential execution of the video frames into tasks that can be scheduled by the Nanox++ runtime system by leveraging unused resources of the underlying architecture. Since these tasks feature a high CPU consumption, they are good candidates for both OmpSs@SMP and OmpSs@Cluster, and benefit from coarse-grained parallelism.

Figure 22 shows the three main operations involved in frame processing: location and extraction of the ROIs (which are depicted using yellow blocks), and the processing of the two ROIs (operations indicated with the green blocks). This figure also shows both sequential and parallel solutions. The pseudocode of the iris recognition task with OmpSs annotations is included in the Appendix.

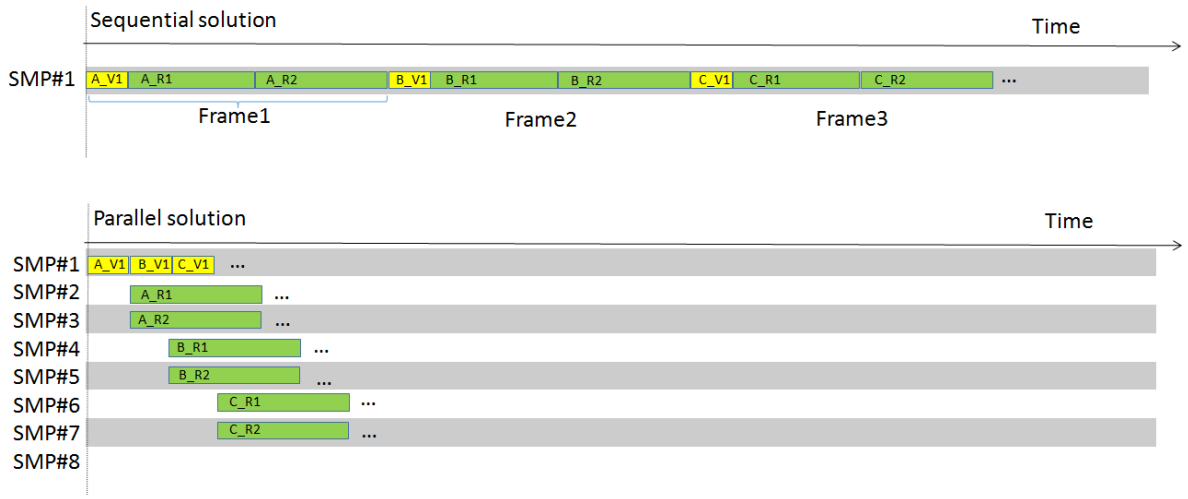

Figure 22. *Execution diagram of the Iris recognition task in the sequential and parallel solutions.*

3.7.1 Trace of the task execution and obtained performance results

The execution trace of this module is shown in Figure 23. Red boxes represent operations to locate and extract ROIs, the dark red and magenta boxes represent tasks used to process the ROIs that are scheduled into unused SMP resources.

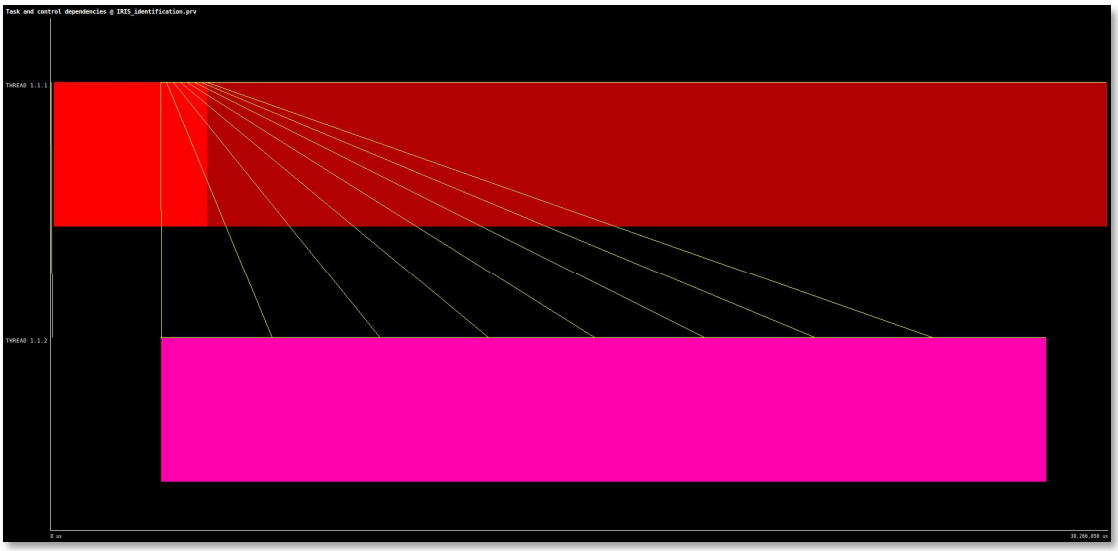

Figure 23. *Paraver trace of the iris recognition task using 2 threads of the AXIOM Evaluation Platform.*

Table 6 shows the execution time of the iris recognition module on the AEP. The input data used for the experiments were two videos: one with 8 frames, and another one with 98 frames. Both videos were encoded at Full HD resolution, and contained two ROIs with size of 260x260 pixels.

Table 6. *Execution time of the iris recognition task on the AEP.*

Number of frame	Execution time without OmpSs pragma [sec]	Number of OmpSs tasks	1 worker thread		2 worker thread	
			Execution time with OmpSs@SMP [sec]	speedup	Execution time with OmpSs@SMP [sec]	speedup
8	52.3	16	52.3	1x	38	1.37x
98	596.5	196	590	1x	421	1.41x

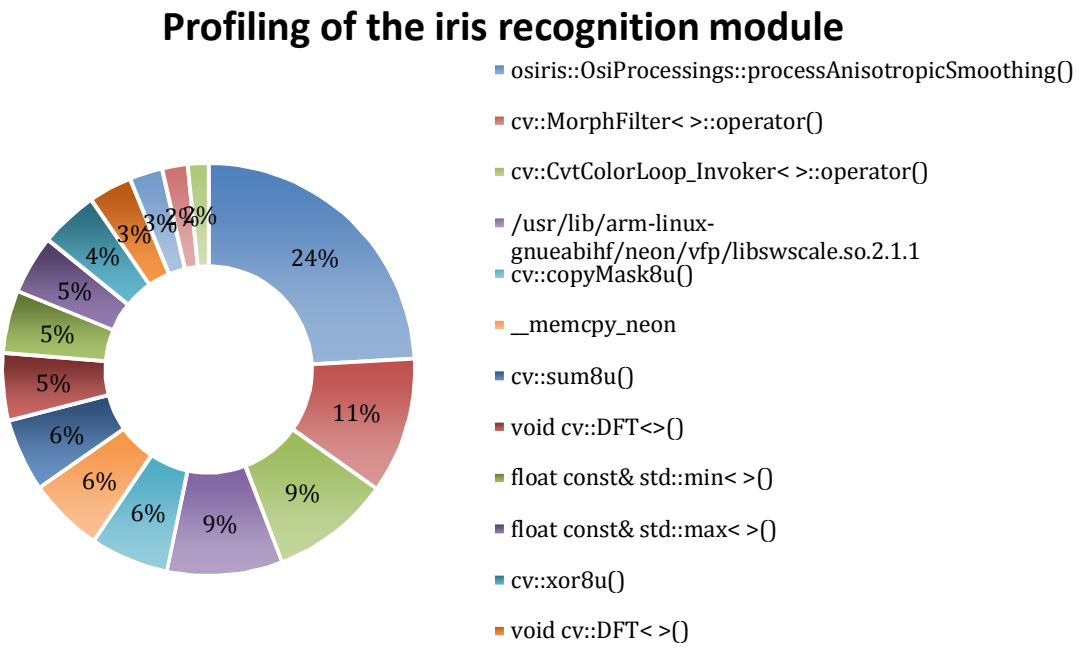

Figure 24. *Profiling of the iris recognition kernel. The profile was done in the AEP using the operf and the opreport tools.*

Also in this module, a further increase of the performance is expected when porting the code to the final AXIOM board. Additionally, the code also exposes enough parallelism to be offloaded to a cluster built from several AXIOM boards.

Unfortunately, the high workload and complexity of the code do not enable to efficiently map tasks of this module into the FPGA PL resources. However, the iris recognition module incorporates eye segmentation and so the anisotropic smoothing module. In Figure 24, profiling information of the iris recognition module shows the anisotropic function is the most resource-intensive part of iris recognition thus making it a good candidate for FPGA offloading. Therefore, the iris recognition module could create tasks that may take advantage of the underlying resources available in the nodes of the AXIOM cluster.

Finally, the whole application will be again reviewed in the future for further parallelization optimizations on FPGA PL resources. For instance, operations such as `max` and `min` functions represent about 10% of the execution time. During the coming months, we will explore the possibility of offloading other parts of the code in order to further increase the speedup.

4 App/Service Prototyping

This activity aims at producing interactive prototypes of Apps/Services, based on the challenges addressed in the two case studies (Smart Home Living and Smart Video Surveillance), for envisioning solutions with an appealing user experience.

4.1 Rapid prototyping tools for Cyber Physical Systems

Prototyping massively interconnected objects, devices, and sensors raises equally massive challenges regarding the resources that will allow designers to manage the complexity of such systems, and to exploit the opportunities such technologies will open up.

An unavoidable challenge in designing IoT solutions is the need of authoring environments and architectural infrastructures for supporting end-user programming and promoting tinkering. Such authoring environments should enable the creation of domain-specific applications, supporting designer to connect IoT appliances and to specify behaviors and the presentation of information in a highly-personalized manner, which should be tailored to the end-user and its context of use [21].

Designing for the CPSs is more complex than designing for regular web services or applications. Nowadays, by relying on mature and consolidated design patterns and graphical frameworks, it is relatively easy to design beautiful user interfaces. However, users could still have a poor experience of the IoT products as a whole. Designing a great connected product requires a different approach to user experience [22].

Moreover, in AXIOM the challenges addressed by the two case studies (Smart Home Living and Smart Video Surveillance) rest on machine learning solutions. More into the detail, the Smart Home Living scenario is related to a recent domain named *Interactive Machine Learning* [23]; the Smart Video Surveillance uses neural networks (CNN) in order to produce accurate results. Furthermore, the Smart Video Surveillance and the Smart Home Living Scenarios are envisioned to merge in Marketing or Edutainment scenarios, either of them based on Interactive Machine Learning.

In respect to this, we must acknowledge that today there are no authoring tools for rapid prototyping of Apps or Services based on Interactive Machine Learning.

4.2 Interactive machine learning

Interactive Machine Learning is a new field, which lives at the intersection of User Experience and Machine Learning research. Human application of machine learning algorithms to real-world problems requires embedding the algorithms in software or hardware tools of some sort. Even though the form and the usability of these tools impact the feasibility and the efficacy of applied machine learning work, research at the intersection of HCI and machine learning is still a very young area. Notwithstanding this, as recently remarked [24], almost anyone wondering how to incorporate AI into their own business, creative tool, software product or design practice - would be better off studying this field than maybe any other part of the AI landscape.

Machine learning is a powerful tool for transforming data into computational models that can power up user-facing applications. However, potential users of such applications have limited involvement in the process of developing them.

The intricacies of applying machine learning techniques to everyday problems have largely restricted their use to skilled practitioners. In the traditional applied machine learning workflow, these practitioners collect data, select features to represent the data, pre-process and transform the data, choose a representation and learning algorithm to construct the model, tune parameters of the algorithm, and finally assess the quality of the resulting model. This assessment often leads to further iterations on many of the previous steps. Typically, any end-user involvement in this process is mediated by the practitioners and is limited to providing data, answering domain-related questions, or giving feedback about the learned model. This results in a design process with lengthy and asynchronous iterations, which limits the end-users' ability to impact on the resulting models.

Instead, in Interactive machine learning, learning cycles involve more rapid, focused, and incremental model updates than in the traditional machine learning process (see Figure 25). These properties enable everyday users to interactively explore the model space through trial-and-error and drive the system towards an intended behavior, reducing the need for supervision by practitioners. Consequently, interactive machine learning can empower end-users to create machine learning-based systems for their own needs and purposes. However, enabling effective end-user interaction with interactive machine learning introduces new challenges that require a better understanding of end-user capabilities, behaviors, needs and, first of all, rapid prototyping tools for jointly exploring the design space.

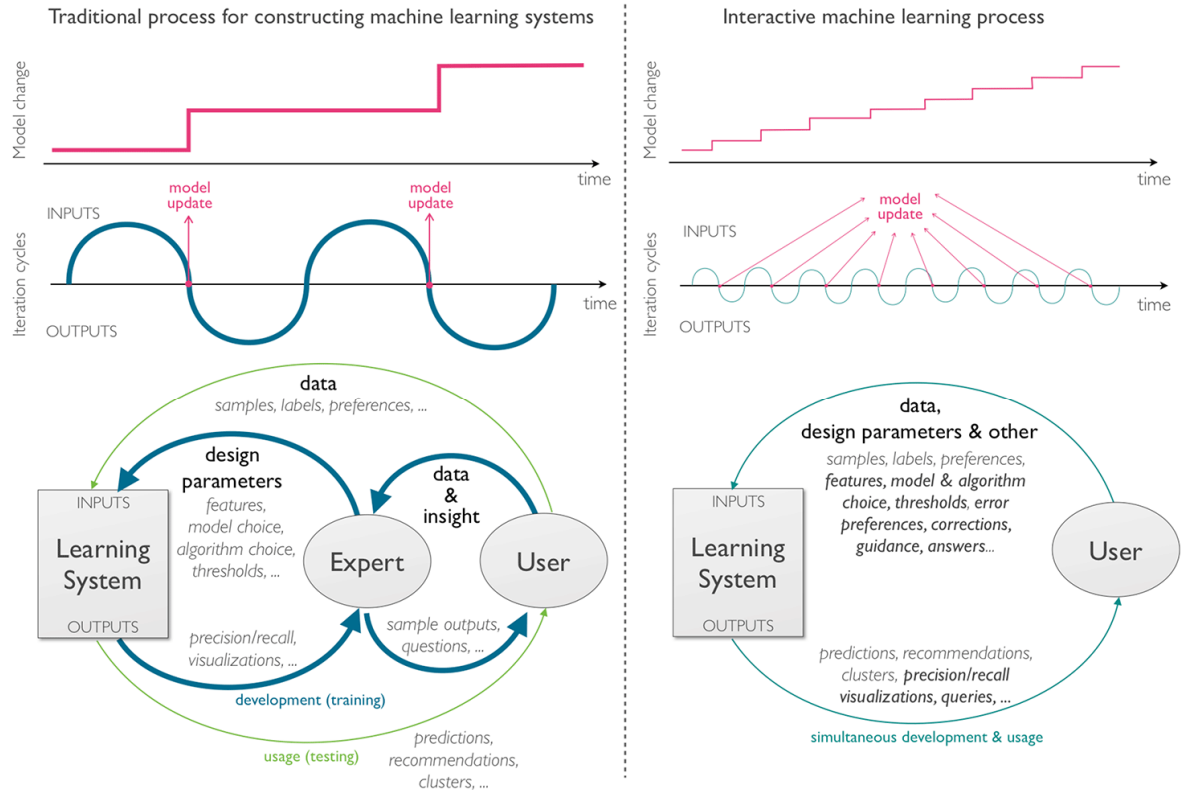

Figure 25. *In machine learning, people iteratively supply information to a learning system and then observe and interpret the outputs of the system to inform subsequent iterations. In interactive machine learning, these iterations are more focused, frequent and incremental than traditional machine learning. The tighter interaction between users and learning systems in interactive machine learning necessitates an increased focus on studying the user's involvement in the process.*

Prototyping tools should focus on supporting human-computer interaction in the context of creating machine learning systems, where users are engaged in several tasks, including choosing and training a learning algorithm, evaluating and comparing models, and supplying training data. The scope of relevant users includes researchers applying machine learning techniques to data analysis in an application domain of their expertise, developers of user interfaces containing machine learning components, and end-users of software tools that directly engage users in controlling some aspects of a machine learning system, such as providing training data and evaluating trained models.

Therefore, we decided to design a flexible and easy-to-integrate technology for prototyping CPS/IoT solutions based on interactive Machine Learning. To the best of our knowledge this is the first ever prototyping platform for such domain.

4.3 UAPPI

In the design of our prototyping environment, we found that the best opportunities were offered by App Inventor, an open source Web IDE. App Inventor for Android is a visual programming platform for creating Android applications. It was developed at Google Labs by an MIT team led by Hal Abelson [25]. Developing apps in App Inventor does not require writing classic source code. The look and behavior of the app is developed "visually", using a series of building blocks for each intended component. The visual nature of its language reduces the syntax problems common among programming beginners first starting to design an app.

A key feature of the programming environment is live programming. Code changes are immediately and continually reflected in a constantly running program. Liveness makes program development more interactive by incorporating the effects of program changes more quickly than if they are incorporated in the traditional edit-compile-run-test approach. The other key feature is Event-driven programming.

Our extension of App Inventor, UAPPI, gives novices the tools to develop applications by providing must-have functionalities like GUI, network access and storage on databases and by incorporating the popular Arduino sensors and actuators. UAPPI integrates two worlds, Android and Arduino, by mean of a powerful and easy to learn visual programming platform. UAPPI can be understood as an extension of App Inventor for the Cyber-Physical world.

UAPPI uses a Java web server to expose the web-based IDE user interface, which allows Interaction Designers to prototype and develop applications for the AXIOM scenarios. The projects are saved as soon as the user works, stored in the UAPPI server itself, and can also be exported and reimported for backup or sharing purposes.

Live programming is implemented by means of a special app running on the UDOO board, the UAPPI Companion. Although final apps can ultimately be compiled to produce ordinary `apk` files, browser interaction during live development is accomplished by the Companion runtime, which serves as an interpreter for the UAPPI code. The UAPPI Companion is an app which embeds all the UAPPI components (GUI elements, storage options, libraries, etc.) and receives the UAPPI code from the development computer connected to the IDE (see Figure 26). The UAPPI code arranges graphical elements on the screen, sets variables and properties, defines procedures and event handlers.

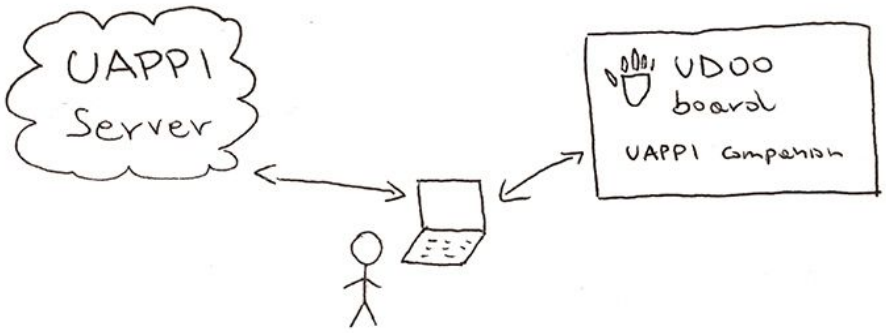

Figure 26. *An Interaction Designer can develop prototypes using UAPPI running on his computer's browser, and later run them on the UDOO board for testing.*

UAPPI has been developed on the UDOO board since it provides the required hardware to mock the AXIOM scenarios: Full HD camera compatibility, microphone input, and an Arduino interface. This is not intended as a replacement of the AXIOM platform, which provides a different architecture: accelerated-computing on a cluster instead of low-computing on a single node, Linux instead of Android, Arduino soft-core IP instead of an ASIC chip, etc. The UAPPI comes into play for design purposes only. The final applications and services derived from the envisioned scenarios will be implemented on the final AXIOM architecture at a later stage in the project, so this tools will easy such process.

Table 7. *Arduino components available in UAPPI with the related implemented features.*

UAPPI Component	Features provided
`UdooQuad`	Low-level Arduino API for custom analog and digital logic (`pinMode`, `digitalWrite`, `digitalRead`, `analogWrite`, `analogRead`, `delay`, `map`, `attachInterrupt`, `...`)
`UdooProximitySensor`	HC-SR04 Ultrasonic proximity sensor (5-200cm) compatibility
`UdooThermoSensor`	DHT-11 and DHT-22 temperature and humidity sensors compatibility
`UdooColorSensor`	TCS34725 RGB color sensor compatibility
`UdooServo`	PWM-controlled servo motors compatibility

4.3.1 Interaction with Arduino

Easy interfacing with the Cyber-Physical world is a key objective of the AXIOM project. To achieve this, the board integrates an Arduino. Partner SECO is developing a soft-core Arduino IP in T6.4, and the Arduino UNO pinout will guarantee support for a plenty of pluggable expansion board (so-called "shields").

Arduino compatibility is very useful during the prototyping phase. It allows to easily connect sensors and actuators, and even mock real-world objects. For instance, the video door entry system of the Smart Home Living scenario can control a door lock implemented with servo motors in minutes.

In UAPPI, we have developed several components to control the Arduino microcontroller of the UDOO board. The preliminary Arduino compatibility in UAPPI is summarized in Table 7.

4.3.2 Prototyping interactions with machine learning

The first ML class of algorithm considered for prototyping interactions is Support Vector Machine (SVM). Support Vector Machine [26] is a machine learning technique used in data classification problems. Its simplicity of use, robustness and ability to generalize, has built a reputation making it one of the most used machine learning algorithms. It is also used by partner HERTA in the currently deployed version of their BioMarketing software product [27].

The SVM is trained by providing it some data samples. Each data sample is composed of a features vector and a tag. When some examples are provided, the machine computes a mapping function able to classify unlabeled feature vectors, never seen during the training phase.

UAPPI provides a component, `UdooSvm`, which allow both the training phase and the classification using libsvm [28] for the underlying implementation. For the separation of concerns principle, this component is not aware of what the feature vectors represents or how they can be generated. Feature vectors can be generated by other blocks, specific to the required task, like analyzing a video or audio frame or reading data from some Arduino digital and/or analogic pins.

Google Vision [29] is a framework for finding objects in photos and video using real-time on-device vision technology.

The mobile face API [30] finds human faces in photos, tracks positions of facial landmarks (eyes, nose, and mouth) and provides information about the state of facial features -- are the subject's eyes open? Are they smiling?

UAPPI integrates this library in a component, `UdooVision`. Using an USB camera, it can detect the prominent face in the frame. The component exposes values between 0 and 1 proportional to how much the eyes are open and how much the person is smiling (0 means closed eyes / no smile; 1 means eyes fully opened / full smile).

4.4 Example

Using this prototyping environment, in a few minutes it is possible to create a smart lamp. Winking the left eye, the lamp enters in “programming mode”, where it powers up an RGB LED strip proportionally to the smile. Winking again exits the programming mode.

The programming blocks for this lamp are shown in Figure 27 depicted below.

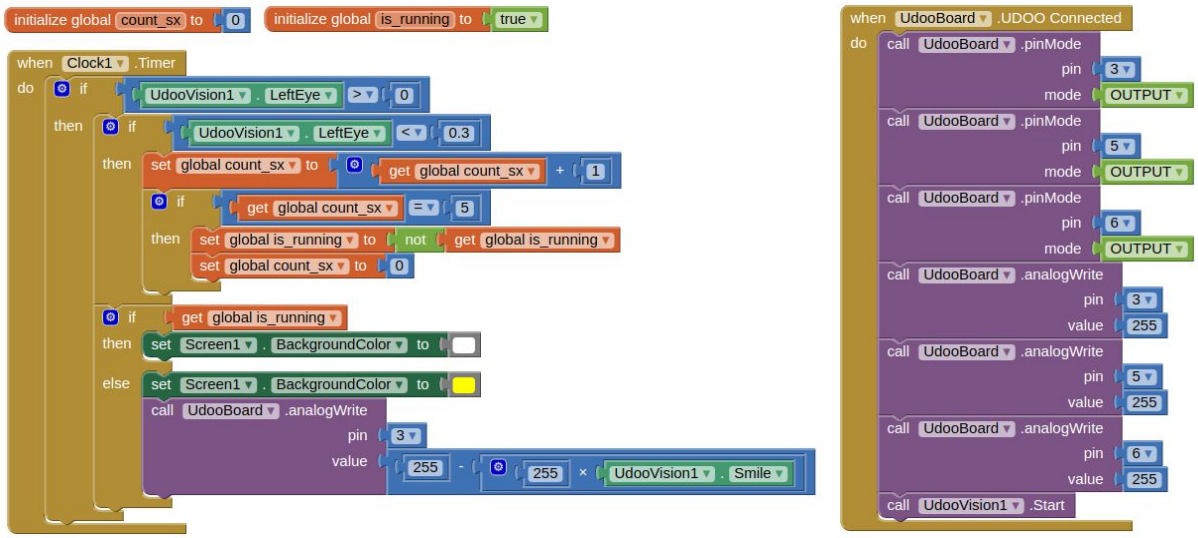

Figure 27. *UAPPI blocks for a lamp that can be powered up smiling in front a camera.*

5 Building a data set for prototyping exploration

In order to facilitate the setting up of the selected use-case scenarios, we started to collect data that is going to be used for building machine learning models (e.g. Smart Home or Video Surveillance use-cases). Additionally, it could be also used not only for improving both HERTA and VIMAR's product portfolio but also for enabling research to third-party institutions.

5.1 Photo archive for facial recognition system

The first data set includes faces of random people that agreed to collaborate with the AXIOM project. These faces are needed for training and fine-tuning the deep neural networks used for demographic estimation, and related to HERTA's WP3 smart marketing use-case scenario. In order to generate such data set, we captured pictures of random people by following the protocol defined in the Ethics Screening Report. Therefore, collected data will be anonymized, and later distributed on public repositories using a permissive license: OpenAIRE [31] (funded by the EC), and Zenodo [32] (funded by both CERN and the EC).

Collected face images included various individuals from different backgrounds:

- Different ages (from 8 years old to 87);
- Different gender (male, female);
- Different ethnicities: "white", "asian" and "black" (Indian people and similar ethnicities were tagged as "white");
- Occlusions (glasses, sunglasses, hat/cap, scarf, hand, smoking);
- Changing illumination conditions;

Initially, we planned to capture images of at least 100 persons but thanks to the agreement of people to participate in the experiment, we decided to extend this goal to 150 individuals to better refine the system. As it is known, state-of-the-art neural network models trained with deep learning techniques yield higher accuracy as the amount of training images is increased.

Regarding the resolution used for capturing pictures, we selected the maximum possible resolution allowed by the camera device (i.e. 4K resolution). This process was automated by means of the software application developed by HERTA (see Section 2.3). The application's user interface features buttons, automatically manage persistence, and handle I/O to speed up the annotation/tagging of demographic characteristics (age, ethnicity and coordinates/location) of subjects.

For each person stored in the AXIOM database, we captured at least 25 pictures:

- A single very high-quality frontal image;
- Several images, with small rotations: 6 x top, 6 x bottom, 6 x left, 6 x right (up to 30 degrees);
- Dynamic lighting conditions: for both frontal and each rotated pose, we captured pictures using normal and minimal lighting. Some pictures were captured by turning the lights off or with poor illumination conditions (i.e. by using differently positioned lamps, directional lights, and colored lights);
- Faces were partially occluded. In some cases, we invited the subject to wear glasses (different models/design/trademarks), sunglasses, hats/caps, scarfs, moustache/beard. We paid attention to not use always the same glasses/sunglasses or hats. Otherwise, the obtained neural network would be prone to overfitting and could not learn well how to generalize features.

In addition to that, we captured several HD/4K videos of groups of people moving around. Generally, we invited 3-6 persons to appear on a single frame. These videos were recorded with the purpose of being used later as a validation data set for evaluating the accuracy of the trained neural network models. Also, they are useful for testing and optimizing the kernel's source code on the FPGA without constantly relying on an external camera pointing to people.

Finally, each picture was annotated using both a unique anonymous ID assigned to each participant and the number of frame within the video sequence. The subject ID will prove useful to detect possible errors in certain annotated person-dependent cues (such as gender or ethnicity).

5.2 Audio recordings

The second data set concerned the recording of audio samples, collected from the same people involved in the first data set. These audio samples are required for the training and testing of VIMAR's algorithms implemented for the SHL scenario. The data was collected during two sessions.

FIRST SESSION

The recording speaker was settled to capture:

- Gender (female/male)
- Age (range of ages)

- Participant ID
- Time and date of the registration

The initial target was set to 100 speakers. In order to reach this ambitious goal, we selected participants from a sample of students, researchers, professors, and their relatives.

For every speaker, the following information was recorded (16-bit, 16 KHz sample, Word/Impostor):

- Natural speech for 15 minutes (two times) for a total of 30 minutes' sample;
- Three registrations of 5 minutes captured during different days;

The environment for the recording was a silent room with no background noise.

SECOND SESSION

The recording speaker was settled to capture people during their everyday work with these variables:

- A subject working alone using his/her PC workstation, mobile devices or papers;
- People working in a group;
- People talking by phone or having a Skype call;
- Events interfering with the ordinary activities: strange sound, people speaking loud or background noises.

6 Extra achievements

In order to facilitate the implementation, porting and integration of the SVS and SHL scenarios developed by HERTA and VIMAR, following the protocol defined in the Ethics Screening Report, partner UNISI took the responsibility of building the data sets required for training models for the facial demographic analysis and speaker identification algorithms.

UNISI shot pictures of random individual's faces. These pictures are going to be used during month m26 for retraining the facial analysis CNNs developed by HERTA so they can be publicly released and meet the open access data policy.

The second set of data contains the recording of audio samples, collected from the same people involved in the first data set, with the aim of training and validating the required models for the SHL scenario.

Further details are available in Section 5.

7 Confirmation of DoA objectives

PLANNED	DELIVERED
DELIVERABLE: SCENARIOS REFINEMENT	
• Scenarios refinement using the AXIOM CPS platform	Already delivered in D3.1
• Benchmark set definition	Already delivered in D3.1
• Services/system integration and appealing user experience	Interactive prototypes production for the two case studies is addressed in Section 4
DELIVERABLE: SCENARIOS PORTING	
• Porting of the SVS Application to the OmpSs Programming Model	Implementation steps and results are shown in Section 2
• Porting of the SHL Application to the OmpSs Programming Model	Implementation steps and results are shown in Section 3

8 Conclusions

In this deliverable, we have detailed the design decisions taken when designing and coding the required applications for the SVS and SHL case studies. Additionally, it also described the initial experiences when parallelizing the most time-consuming kernels using the directives offered by the OmpSs programming model. The initial performance results show that by relying on OmpSs directives, it is possible to achieve a reasonable speed up in sequential kernels with minimal efforts.

The SVS use case also involved the architectural design and training of two state-of-the-art convolutional neural networks for estimating the age and gender of detected faces. It is expected that in order to achieve good performance results when using the OmpSs@FPGA backend, it will be necessary to perform low-level optimizations at the HLS level for enabling real-time performance of the convolutional neural network inference engine. Significant efforts will be also needed for reducing the latency of the LBP kernel to the maximum extent on the PL logic, as it currently requires roughly 500ms for analyzing a picture with 24 simultaneous faces on the CPU cores of the Zynq-7000 SoC. In order to meet the target of real-time performance, these combined figures must be reduced to significantly less than 40 ms per frame.

Another task that also needs to be further explored is the H.264 decoding process. It is expected that this task will be managed either by a third-party IP logic block or by a dedicated subset of the ARM Cortex A53 cores that are available on the Ultrascale+ platform.

The experience obtained when studying the SHL scenario has led to combine several open source libraries to develop the required application. VIMAR's work brought out that code parallelization requires a deep understanding of the algorithms utilized, and also of the third-party code used for implementing the needed libraries.

Algorithms that have not been implemented for parallel execution normally require some modification to obtain good performance results before introducing OmpSs directives. One specific case is the feature extraction module in which the architecture and some variables have been modified.

A detailed analysis of the granularity of the tasks created with OmpSs directives is needed in order to gain increase performance, as it is shown on the obtained results of the anisotropic smoothing module.

BSC's visualization tools have proved their high potentiality and productivity when carrying out this type of analysis.

The introduction of OmpSs directives to create tasks with coarse-grained parallelism was a smart choice, and succeeded in obtaining speed ups with minimal efforts. This was the case of the iris recognition module, and LBP cascade evaluation kernel.

All the studies carried out in this deliverable did not involve the usage of FPGA PL resources. A preliminary FPGA mapping study has just started in collaboration with BSC's researchers. From our experience, it was immediately clear that in order to exploit those resources, an additional in-depth analysis and code modifications will be necessary. These initial unreported experiments have shown that a non-careful naïve mapping leads to a rapid exhaustion of the available FPGA PL resources. In the coming months, this activity will focus on studying mapping of tasks to the FPGA PL using OmpSs@FPGA.

Other publications of the project [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] are reported in the reference list.

References

[1] Qiu, Jiantao, et al. "Going deeper with embedded FPGA platform for convolutional neural network." Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2016.

[2] LibAv – Open source audio and video processing tools, https://libav.org/

[3] Schapire, Robert E., and Yoav Freund. *Boosting: Foundations and algorithms*. MIT press, 2012.

[4] NVIDIA Tesla P4 and P40 inferencing accelerators, http://www.nvidia.com/object/accelerate-inference.html

[5] Xilinx Kintex Ultrascale+, https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale-plus.html

[6] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

[7] Szegedy, Christian, et al. "Going deeper with convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015.

[8] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016.

[9] Ren, Shaoqing, et al. "Faster R-CNN: Towards real-time object detection with region proposal networks." *Advances in neural information processing systems*. 2015.

[10] Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014.

[11] Collobert, Ronan, Samy Bengio, and Johnny Mariéthoz. *Torch: a modular machine learning software library*. No. EPFL-REPORT-82802. Idiap, 2002.

[12] Abadi, Martín, et al. "TensorFlow: A system for large-scale machine learning." *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*. 2016.

[13] ATLAS – Automatically tuned linear algebra software, http://math-atlas.sourceforge.net/

[14] GEMM-lowp – A small self-contained low-precision GEMM library, https://github.com/google/gemmlowp

[15] Deep Learning with INT8 operations on Xilinx FPGAs, https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[16] Alize, http://alize.univ-avignon.fr/

[17] SPro, https://www.irisa.fr/metiss/guig/spro/spro-4.0.1/spro.html

[18] Osiris, http://svnext.it-sudparis.eu/svnview2-eph/ref_syst/Iris_Osiris/

[19] OpenCV, http://opencv.org/

[20] GStreamer Conference 2016, https://gstreamer.freedesktop.org/conference/2016/

[21] Jenkins, Tom. "Designing the Things of the IoT" Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction. ACM, 2015.

[22] Rowland, Claire, et al. Designing connected products: UX for the consumer Internet of Things. " O'Reilly Media, Inc.", 2015.

[23] Amershi, Saleema, Maya Cakmak, William Bradley Knox, and Todd Kulesza. "Power to the people: The role of humans in interactive machine learning." AI Magazine 35, no. 4 (2014): 105-120.

[24] https://medium.com/@atduskgreg/power-to-the-people-how-one-unknown-group-of-researchers-holds-the-key-to-using-ai-to-solve-real-cc9e75b1f334#.x394jvgx0

[25] Abelson H.: App Inventor for Android, https://research.googleblog.com/2009/07/appinventor-for-android.html

[26] Andrew, Alex M. "An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods by Nello Christianini and John Shawe-Taylor, Cambridge University Press, Cambridge, 2000, xiii+ 189 pp., ISBN 0-521-78019-5 (Hbk,£ 27.50)." (2000): 687-689.

[27] BioMarketing – Herta Security, http://www.hertasecurity.com/en/products/biomarketing

[28] Chih-Chung Chang and Chih-Jen Lin, LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[29] Mobile Vision – Google Developers, https://developers.google.com/vision/

[30] Android com.google.android.gms.vision.face APIs, https://developers.google.com/android/reference/com/google/android/gms/vision/face/package-summary

[31] Openaire, https://www.openaire.eu

[32] Zenodo, https://zenodo.org

[33] R. Giorgi, "Transactional memory on a dataflow architecture for accelerating Haskell," WSEAS Trans. Computers, vol. 14, pp. 794–805, 2015.

[34] 2. Giorgi and A. Scionti, "A scalable thread scheduling co-processor based on data-flow principles," ELSEVIER Future Generation Computer Systems, vol. 53, pp. 100–108, July 2015.

[35] D. Theodoropoulos et al., "The AXIOM project (agile, extensible, fast I/O module)," in IEEE Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation, July 2015.

[36] R. Giorgi, "Scalable Embedded Systems: Towards the Convergence of High-Performance and Em-bedded Computing", Proc. 13th IEEE/IFIP Int.l Conf. on Embedded and Ubiquitous Computing (EUC 2015), Oct. 2015.

[37] R. Giorgi, "Exploring Dataflow-based Thread Level Parallelism in Cyber-physical Systems", Proc. ACM Int.l Conf. on Computing Frontiers, New York, NY, USA, 2016, pp. 6.

[38] A. Rizzo, G. Burresi, F. Montefoschi, M. Caporali, R. Giorgi, "Making IoT with UDOO", Interac-tion Design and Architecture(s), vol. 1, no. 30, Dec. 2016, pp. 95-112.

[39] L. Verdoscia, R. Giorgi, "A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs", Mathematical Problems in Engineering, vol. 2016, no. 1, Apr. 2016, pp. 1-21.

[40] S. Mazumdar, E. Ayguade, N. Bettin, S. Bueno J. and Ermini, A. Filgueras, D. Jimenez-Gonzalez, C. Martinez, X. Martorell, F. Montefoschi, D. Oro, D. Pnevmatikatos, A. Rizzo, D. Theodoropou-los, R. Giorgi, "AXIOM: A Hardware-Software Platform for Cyber Physical Systems", 2016 Eu-romicro Conf. on Digital System Design (DSD), Aug 2016, pp. 539-546.

[41] R. Giorgi, N. Bettin, P. Gai, X. Martorell, A. Rizzo, "AXIOM: A Flexible Platform for the Smart Home", Springer Int.l Publishing, Cham, 2016, pp. 57-74.

[42] P. Burgio, C. Alvarez, E. Ayguade, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, R. Giorgi, "Simulating next-generation cyber-physical computing platforms", Ada User Journal, vol. 37, no. 1, Mar. 2016, pp. 59-63.

[43] Jimenez-Gonzalez, Daniel; Alvarez-Martinez, Carlos; Filgueras, Antonio; Martorell, Xavier; Langer, Jan; Noguera, Juanjo; Vissers, Kees, "Coarse-Grain Performance Estimator for Heterogeneous Parallel Computing Architectures like Zynq All-Programmable SoC" (Journal Article) Second International Workshop on FPGAs for Software Programmers FSP 2015, CoRR , 2015.

[44] R. Giorgi, S. Mazumdar, S. Viola, P. Gai, S. Garzarella, B. Morelli, D. Pnevmatikatos Dionisios and Theodoropoulos, C. Alvarez, E. Ayguade, J. Bueno, D. Filgueras Antonio and Jimenez-Gonzalez, X. Martorell, "Modeling Multi-Board Communication in the AXIOM Cyber-Physical System", Ada User Journal, vol. 37, no. 4, December 2016, pp. 228-235.

Appendix

SVS workloads

CNN inference

The C++ source code of the convolve operation is included below as a reference. The SGEMM operation is highlighted in bold. The auxiliary `Im2Col` function is a required transformation for implementing convolutions using the SGEMM operation. Possible additional nonlinearity to be applied after SGEMM is removed from the source code.

```
void dnn::Convolve(const dnn::Tensor& input, const dnn::Tensor& weights, const dnn::Tensor& bias, dnn::Padding padding,
    dnn::Stride stride, dnn::Nonlinearity nonlin, dnn::Tensor& output)
{
    assert(input.num_ == 1);

    // Prepare Toeplitz matrix (input with kernel deviations)
    dnn::Tensor col;
    Im2Col(input, dnn::Kernel(weights.width_, weights.height_), padding, stride, col);

    // Prepare output
    output = dnn::Tensor(col.width_, col.height_, weights.num_, 1);

    // Carry out convolution by simple matrix multiplication
    int spatial_dims = col.width_ * col.height_;
    int kernel_dims = weights.channels_ * weights.height_ * weights.width_;
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                output.channels_, spatial_dims, kernel_dims,
                1.f, weights.values_.data(), kernel_dims,
                     col.values_.data(), spatial_dims,
                0.f, output.values_.data(), spatial_dims);

    if (nonlin == dnn::E_NONLIN_ID)
    {
        for (int c = 0; c < output.channels_; ++c)
        {
            int coffset_out = c * spatial_dims;
            for (int px = 0; px < spatial_dims; ++px)
            {
                // Identity function (no non-linearity), just add biases
                output.values_[coffset_out + px] += bias.values_[c];
            }
        }
    }

    /* Additional non-linear functions are removed in order to keep simplicity */
}
```

LBP cascade evaluation kernel

The C++ pseudocode enclosed below includes the main LBP face detection kernel (`LBPCascadeEvaluation`) in charge of evaluating the boosted cascade of features, and annotated with OmpSs directives. Additionally, it also shows the code of the method (`FaceDetection_LBP`) responsible for evaluating the cascade of features for each scale of the synthetic pyramid, which is generated from the input picture. It should be noted that the pseudocode, and as such, it does not include the full source code. Confidential code parts protected under NDA were removed.

```
#pragma omp target device(fpga,smp) copy_deps
#pragma omp task in(scaledFrame) inout(detections)
void LBPCascadeEvaluation(unsigned char* scaledFrame, int sstep, int swidth, int sheight, float scale,
     float threshold, std::vector<Detection> &detections)
{
    int x, y;

    #pragma omp parallel for
```

```
    for(x = 0; x < swidth - WINDOW_SIZE; x+=2) {
        #pragma omp parallel for
        for(y = 0; y < sheight - WINDOW_SIZE; y+=2)
        {
            float score = 0.0;

            for (int stage = 0; stage < NUMSTAGES; stage++)
            {
                // Locate central pixel in image, store horizontal and vertical steps
                const char *filter = LBP_FILTERS[stage];
                uchar* p = scaledFrame + (x + filter[0]) + sstep * (y + filter[1]);
                int stepw = filter[2];
                int steph = filter[3] * sstep;
                int bp = 0;

                // Build LBP pattern, comparing to the neighbors stated in the filter
                uchar* pn = p - stepw;
                bp |= (*pn  > *p) << 7;  pn += steph;
                bp |= (*pn  > *p) << 6;  pn += stepw;
                bp |= (*pn  > *p) << 5;  pn += stepw;
                bp |= (*pn  > *p) << 4;  pn -= steph;
                bp |= (*pn  > *p) << 3;  pn -= steph;
                bp |= (*pn  > *p) << 2;  pn -= stepw;
                bp |= (*pn  > *p) << 1;  pn -= stepw;
                bp |= (*pn  > *p) << 0;

                // Accumulate scores
                score += LBP_SCORES[stage + bp];
                if (score < LBP_THRESHOLDS[stage]) { break; }
            }

            // If the score is greater than the threshold set by the user, the region must be classified as a face
            if( score > threshold )
              detections.push_back(Detection(x / scale, y / scale, WINDOW_SIZE / scale, WINDOW_SIZE / scale, score));
        }
    }
}


std::vector<Detection> FaceDetection_LBP(const std::vector<unsigned char>& frameIn, int width, int height)
{
     std::vector<Detection> faces;
     std::vector<unsigned char> scaledFrame;
     std::vector<unsigned char> img2 = frameIn;
     int swidth, sheight, sstep;
     int img2_width = width;
     int img2_height = height;
     float scale;



     // Process the image pyramid
     for (int idx=0; idx < NUM_SCALES; idx++)
     {
         scale = LBP_SCALES[idx];
         swidth  = width * scale;
         sheight = height * scale;

         // Image resizing
         scaledFrame = BilinearResize(img2, img2_width, img2_height, swidth, sheight);
         img2 = scaledFrame;
         sstep = swidth;
         img2_width = swidth;
         img2_height = sheight;

         // Launch the LBP cascade evaluation kernel to detect faces
         std::vector<Detection> detections;
         LBPCascadeEvaluation((uchar*) scaledFrame.data(), sstep, swidth, sheight, scale, THRESHOLD, detections);
         #pragma omp taskwait

         // Insert detected faces
         faces.insert(faces.end(), detections.begin(), detections.end() );
     }

     return faces;
}
```

YUV2RGB color space conversion kernel

Enclosed below is the full source code in ANSI C of the YUV2RGB kernel parallelized using OmpSs annotations. The code shows two alternative kernel implementations. The first one is implemented using single-precision floating point arithmetic (`yuv2rgb_fp`), whereas the second uses 8 and 16 bit integers (`yuv2rgb_int`).

```
/* Required for clamping color components to [0,255] range */
#define CLAMP(x) ((x < 0) ? 0 : ((x > 255) ? 255 : x))


/*
 * YUV 4:2:0 planar NV12 format to RGBA 8:8:8:8 conversion Kernel
 *
 * This kernel assumes an input array "yuvframe" containing a picture
 * with a plane of 8-bit Y samples followed by an interleaved U/V plane
 * containing 8-bit 2x2 subsampled color difference samples.
 *
 * As an example, a 4x4 picture encoded in YUV 4:2:0 (NV12) should be
 * stored in the input array as follows:
 *
 *          Y Y Y Y Y Y Y Y
 *          Y Y Y Y Y Y Y Y
 *          U V U V U V U V
 *
 * Thus, given a picture with WxH dimensions, the size of the "yuvframe"
 * array must be ((W x H) + (W x H) / 2).
 *
 * The color conversion implemented in the kernel is designed to be
 * executed on a CPU featuring a single-precision floating point ALU.
 *
 * The output RGBA 8:8:8:8 "rgbaframe" array could be then directly mapped
 * to a texture for displaying the picture on a screen.
 */

/* Floating point version */
void yuv2rgb_fp(uint8_t* yuvframe, uint32_t* rgbaframe, int w, int h)
{
  int i = 0, j = 0;
  float y, u, v;
  float r, g, b;
  int pitch = w * h;


  #pragma omp taskloop grainsize(16) private(i,j,y,u,v,r,g,b)
  for (j = 0; j < h; j++)
  {
    #ifdef DEBUG_ASSIGNMENT
       if ((j==0) || (j==h-1)) printf ("%d: Executing in %d (of %d)\n", j,
           omp_get_thread_num(), omp_get_num_threads());
    #endif

    for (i = 0; i < w; i++)
    {
      /* Input pixel (YUV 4:2:0) */
      y = (float) yuvframe[w * j + i];
      u = (float) yuvframe[pitch + (j / 2) * w + i - (i & 1)];
      v = (float) yuvframe[pitch + (j / 2) * w + i - (i & 1) + 1];

      /* Output pixel (RGB 24 bits per pixel + alpha channel component) */
      r = 1.164f * (y - 16.0f) + 1.596f * (v - 128.0f);
      g = 1.164f * (y - 16.0f) - 0.813f * (v - 128.0f) - 0.391f * (u - 128.0f);
      b = 1.164f * (y - 16.0f) + 2.018f * (u - 128.0f);

      /* Store the results in RGBA 8:8:8:8 format (opaque alpha component) */
      rgbaframe[w * j + i] = ((uint32_t) CLAMP(r)) << 24 |
           ((uint32_t) CLAMP(g)) << 16 | ((uint32_t) CLAMP(b)) << 8 | 0x000000FF;
    }
  }
}

/* Integer version */
void yuv2rgb_int(uint8_t* yuvframe, uint32_t* rgbaframe, int w, int h)
{
  int i = 0, j = 0;
  uint8_t y, u, v;
  int16_t r, g, b;
  int pitch = w * h;
```

```
  #pragma omp taskloop grainsize(16) private(i,j,y,u,v,r,g,b)
  for (j = 0; j < h; j++)
  {
    for (i = 0; i < w; i++)
    {
      /* Input pixel (YUV 4:2:0) */
      y = yuvframe[w * j + i];
      u = yuvframe[pitch + (j / 2) * w + i - (i & 1)];
      v = yuvframe[pitch + (j / 2) * w + i - (i & 1) + 1];

      /* Output pixel (RGB 24 bits per pixel + alpha channel component) */
      r = (298 * (y - 16) + 409 * (v - 128) + 128) >> 8;
      g = (298 * (y - 16) - 100 * (u - 128) - 208 * (v - 128) + 128) >> 8;
      b = (298 * (y - 16) + 516 * (u - 128) + 128) >> 8;

      /* Store the results in RGBA 8:8:8:8 format (opaque alpha component) */
      rgbaframe[w * j + i] = ((uint32_t) CLAMP(r)) << 24 |
           ((uint32_t) CLAMP(g)) << 16 | ((uint32_t) CLAMP(b)) << 8 | 0x000000FF;
    }
  }
}
```

SHL workloads

Pseudocode of the feature extraction module

The pseudocode of the feature extraction function populated with OmpSs annotations is included below for reference.

```
/* ------------------------------------------------------------------- */
/* ----- int cepstral_analysis(sigstream_t *, spfstream_t *) ---------- */
/* ------------------------------------------------------------------- */
#define BUFF 20 // Number of 20 ms windows to be processed in parallel
int cepstral_analysis(sigstream_t *is, spfstream_t *os)
{
  Initialization of all required data structures
  /* ----- Loop on each frame ----- */
  while(1)
  {
           for(k=0; k<BUFF;k++)
           {
              get_next_sig_frame(is, .., buf)      //  Load data in buffArray[] vector from the "is" stream
           }
           if (k==0)  break;
           for(i=0; i<k;i++)
           {
                  #pragma omp task firstprivate(i)
                  {
                           Initialization of the data structures required
                           sig_weight();              /* Weight signal */
                           set_mel_idx();             /* mel idx init */
                           log_filter_bank();         /* Apply the filter bank */
                           dct();                     /* DCT process */
                           set_lifter();              /* Liftering */
                           for (j = 0; j < numceps; j++)
                           {
                              cArray[ii*(numceps+1)+j]*= *(r+j);
                           energy = sig_normalize()    /* Energy */
                           cArray[ii*(numceps+1)+numceps] =(2.0 * log(energy));
                           }
                           free the data structures used
                  } // PRAGMA
           } // FOR
    #pragma omp taskwait
    spf_stream_write(os, &cArray[0], k);  /* Save data in cArray[] vector to "os" stream */
    free the data structures used
    return(0);
} // WHILE
```

Pseudocode of the anisotropic smoothing module

The pseudocode of the anisotropic smoothing function with OmpSs annotations is enclosed below. In order to reduce complexity, the pseudocode shows only the specific case of row-level granularity.

```
/* ------------------------------------------------------------------------------------------------------------------------ */
/* ----- void processAnisotropicSmoothing ( const IplImage * pSrc, IplImage * pDst, int iterations, float lambda )-------- */
/* ------------------------------------------------------------------------------------------------------------------------ */
void processAnisotropicSmoothing ( const IplImage * pSrc, IplImage * pDst, int iterations, float lambda )
{
    Initialization of the data structures required
    // Loop on iterations
    for (int k = 0; k < iterations; k++)
    {
        // Odd pixels
        for (int row = 1; row < image ->height - 1; row++)
        {
            #pragma omp task
            {
             for (int col = 2 - row % 2; col < image->width - 1; col += 2)
             {
                    Read pixels in neighbourhood of original image
                    Compute weber coefficients
                    Write new value of point (row, col) into the image
```

```
            } // FOR
           } // PRAGMA
        } // FOR
        #pragma omp taskwait

        // Even pixels
        for (int row = 1; row < image ->height - 1; row++)
        {
            #pragma omp task
            {
                for (int col = 1 + row % 2; col < image->width - 1; col += 2)
                {
                    Read pixels in neighbourhood of original image
                    Compute weber coefficients
                    Write new value of point (row, col) into the image
                } // FOR
            } // PRAGMA
        }
        #pragma omp taskwait
        Copy the first and the last row on the image
    } // End of iterations (k)

    Process the borders of image
    Copy the data in the output structure
    Release memory

} // End of function
```

Pseudocode of the iris recognition module

The pseudocode of the main function and the iris recognition module with OmpSs annotations is shown below for reference.

```
/* ------------------------------------------------------------------ */
/* ----- int main ( int argc, char *argv[] )---------------------- */
/* ------------------------------------------------------------------ */
int main ( int argc, char *argv[] )
{
    Initialization all the data structures required
    while (true)
    {
       Get a new frame from the network
       osi.run ( frame );  // Start iris identification procedure
    }
#pragma omp taskwait
Release memory
return
}

/* ------------------------------------------------------------------ */
/* ----- void OsiManager::run ( cv::Mat &extFrame )--------------- */
/* ------------------------------------------------------------------ */
void OsiManager::run ( cv::Mat &extFrame )
{
       Detect the ROI inside the frame
       if (frontalEyeDetector ( leftRoi, leftRectEye ))
       {
              Initialization of the data structures required
              #pragma omp task
              {
                process the ROI (the left eye)
              } // PRAGMA
       }
       if (frontalEyeDetector ( rightRoi, rightRectEye ))
       {
              Initialization of the data structures required
              #pragma omp task
              {
                 process the ROI (the right eye)
              } // PRAGMA
       }
       Release memory
} // End of function
```