INNOVATING SOLUTIONS

AUTONN-SW EXECUTIVE SUMMARY

AUTONN-SW

Prepared by:	AUTONN-SW Team
Approved by:	N. Pérez
Authorized by:	N. Pérez

Code:	AUTONN-SW-GMV-ESR-0001
Version:	1.0
Date:	23/01/2024
Internal code:	GMV 20725/24 V1/24

DOCUMENT STATUS SHEET

Version	Date	Pages	Changes
1.0	23/01/2024	14	First issue of the document prepared for the Final meeting

TABLE OF CONTENTS

1 INTRODUCTION....5
1.1 PURPOSE....5
1.2 SCOPE....5
1.3 DEFINITIONS AND ACRONYMS....5
2 REFERENCES....6
2.1 APPLICABLE DOCUMENTS....6
2.2 REFERENCE DOCUMENTS....6
3 PROJECT SUMMARY....7
3.1 GENERATION OF C CODE....7
3.2 GENERATION OF HDL....8
3.3 PRELIMINARY TASTE INTEGRATION....10

LIST OF TABLES AND FIGURES

Table 1-1 Acronyms 5
Table 2-1 Applicable Documents 6
Table 2-2 Reference Documents 6

Figure 3-2. CAELUM tool structure 7
Figure 4-13: Iterative use of the pool variables for memory optimization 8
Figure 5-11. Test structure. 8
Figure 4-17: CAELUM HDL Architecture 9
Figure 4-48: TASTE execution flow. 10
Figure 4-50: Caelum components and functions allocation 11
Figure 4-51: GUI Component Execution 12
Figure 4-52: Hardware SDL diagram 12
Figure 4-54: software SDL Diagram 13

1 INTRODUCTION

1.1 PURPOSE

This document is the Executive Summary of AUTONN-SW project where the main findings of the projects are summarized.

1.2 SCOPE

The scope of this Executive Summary is the whole activity carried out in the frame of AUTONN-SW contract.

1.3 DEFINITIONS AND ACRONYMS

Acronyms used in this document and needing a definition are included in the following table:

Table 1-1 Acronyms

Acronym	Definition
CNN	Convolutional Neural Network
HDL	Hardware Description Language
ICD	Interface Control Document
MBSE	Model Based System Engineering
OBSW	On-Board Software
ONNX	Open Neural Network Exchange

2 REFERENCES

2.1 APPLICABLE DOCUMENTS

The following documents, of the exact issue shown, form part of this document to the extent specified herein. Applicable documents are those referenced in the Contract or approved by the Approval Authority. They are referenced in this document in the form [AD.x]:

Table 2-1 Applicable Documents

Ref.	Title	Code	Version	Date
[AD.1]	Software Requirements Specification	AUTONN-SW-GMV-RS-0001	1.0	24/02/2023
[AD.2]				

2.2 REFERENCE DOCUMENTS

The following documents, although not part of this document, amplify or clarify its contents. Reference documents are those not applicable and referenced within this document. They are referenced in this document in the form [RD.x]:

Table 2-2 Reference Documents

Ref.	Title	Code	Version	Date
[RD.1]	AUTONN-SW Final Report	AUTONN-SW-GMV-RP-0003	1.0	23/01/2024
[RD.2]	ONNX standard definition	https://onnx.ai/		
[RD.3]	Numpy	https://numpy.org/		
[RD.4]	BAMBU	https://panda.deib.polimi.it/?page_id=31		

3 PROJECT SUMMARY

Convolutional Neural Networks (CNNs) are more and more being used to solve very different kind of problems on ground and are also suitable to solve Space problems, e.g. visual based tasks for rovers. Embedding a CNN into an operational on-board software (OBSW) or hardware is challenging as the CNNs are composed by hundreds of operations and they request an extensive amount of memory that need to be used efficiently.

Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms. Most popular machine learning development frameworks have the capability to export the definition of the neural network to ONNX.

In the frame of AUTONN-SW project a proof of concept of a potential future tool (CAELUM) to auto-code CNN defined in ONNX format and integrate this code in and OBSW or HW using HDL has been developed.

CAELUM is a promising tool that will easy the introduction of Convolutional Neural Networks in operational environments where, up to now, it was not possible due to limited resources or development constraints that, for example, forbid the use of available runtimes such as onnxruntime.

Its proposed architecture provides flexibility adapting the obtained outputs to a wide range of CNN models:

3.1 GENERATION OF C CODE

The C generator tool has been developed in python with the following objectives:

- Read an ONNX file
- Process each neuron of the model, storing the relevant information to generate the source code
- Extract and store operators additional inputs in a binary file
- Perform a memory budget analysis to evaluate the feasibility of implementing the network in an operational environment
- Generate the network in C code

It extracts information from an ONNX file (by means of ONNX python package provided by the ONNX official repository) and generates an associated C source code file and header file as well a memory evolution report and a binary file.

The inputs and outputs for the onnxtoC tool are shown in the **Error! Reference source not found.**:

Figure 3-1. CAELUM tool structure

One important achievement of this tool is the memory optimization included in the CNN autogenerated file. This optimization is based on a pool of variables that are statically allocated in the memory that is available for the neural network. The variables of this pool are in charge to store the result of every operation of the network. The benefit of knowing the topology of the network, given by the ONNX file, is that the tool can understand when a variable is no longer needed. This variable value can be overwritten by another ones, so this way no new extra variables need to be defined. An example of this iterative process is shown in the following diagram:

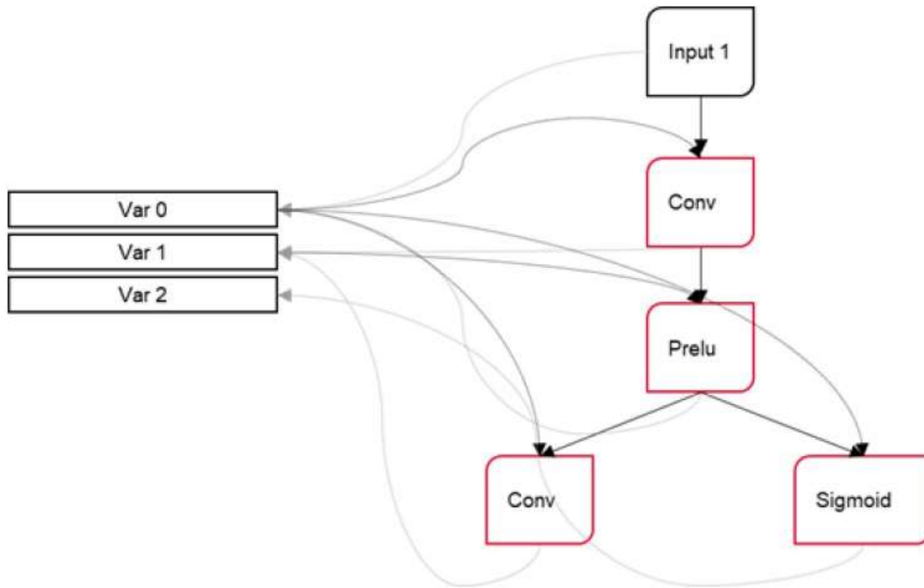

Figure 3-2: Iterative use of the pool variables for memory optimization

Finally, network operators have been implemented in a standalone .c/.h files allow their reuse between different networks since operator's implementation is independent from the network under use, being the only modification the order the specific network code calls them and with which inputs they are called. An example of how all the source code produced in the frame of CAELUM tool is provided in the following figure:

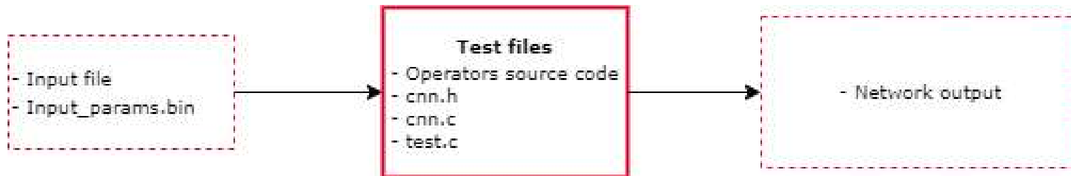

Figure 3-3. Test structure.

3.2 GENERATION OF HDL

Using as input to BAMBU tool the operators source code developed for the C code autogeneration, a hybrid hardware/software architecture which implement single kernel accelerators for relevant operations while the remaining operators are implemented in SW. This approach would generate several digital circuits, one per each accelerated kernel. This approach is much more easily scalable, since the kernels take up a much smaller FPGA real estate when compared to the full network, and they can be reused across different layers of the same type, and across different models using the same layer. If many different models use the same type of kernel, it is not necessary to reconfigure the FPGA when switching models. Also, even multiple kernels of the same type can be implemented in the FPGA to increase parallelism if Bambu is unable to optimize enough. Further, it is much easier to optimize these kernels as opposed to the full network, since in the case of the full network, Bambu will optimize each kernel in combination with all other layers.

The architecture is presented in the following figure:

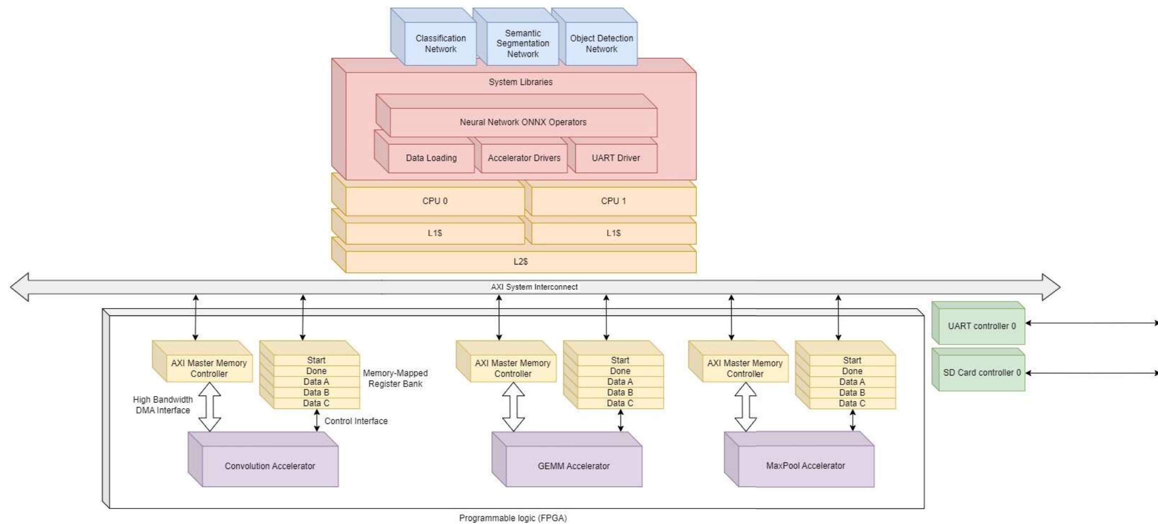

Figure 3-4: CAELUM HDL Architecture

The platform is a dual-core ARM Cortex A9 CPU, included in the Zynq 7000 SoC from AMD. The SoC holds a UART controller and driver for input and output, as well as an SD card controller for permanent data storage. This is where the network weights are stored, and the input images used during the validation tests.

The SoC also features an embedded FPGA, which is used to implement three kernels. Each kernel has a control interface and a high-bandwidth data interface. The control interface is based on an AXI4 Lite protocol, for which a dedicated interface was written in VHDL. The data interface is based on an AXI4 full Master protocol.

The control interface is a set of memory-mapped registers to configure and control the kernel. Using the registers, the application software can:

a. Inform the kernel of the location in memory of the input data, and the location in memory to store the output data in.
b. Configure kernel options, for instance if group convolution is used.
c. Start the kernel computation.
d. Get the busy or idle status of the kernel.

Through the high-bandwidth data interface, the kernel gathers both input data and input weights for the corresponding layer.

Additional activities have been carried out to identify improvements in the C source code developed to optimize the kernels. The main modifications performed were:

1. **Hard-coded tensor dimensions**: In the original version, the convolution operator takes all involved tensor dimensions as a function argument. Since they become a variable parameter, it is not possible for the HLS tool to synthesize highly parallel code.
2. **No if in internal loop**: In the original version, since the tensor dimensions are variable, checks are made to verify that the computation loop does not go out of bounds. Once the tensor sizes are hard-coded, these checks can be removed, since having branching in the inner loop prevents parallelization.
3. **Loop unrolling**: Once the previous optimizations are made, it is possible to perform loop unrolling, e.g. telling the HLS tool to take the inner loop and run the different loop iterations in parallel.
4. **Load-Compute-Store model**: It is usually more efficient to first load all input data into the accelerator local memory, to be able to later operate on this data in parallel without the latency of the memory accesses. In this optimization, all input data is read first into local memory (which is a software scenario would be stored on the stack), and all output data is stored initially also in local memory, and only sent to the memory when the whole computation is finished.
5. **Integer precision**: In machine learning and FPGAs it is very common to avoid floating point arithmetic through quantization, sacrificing some arithmetic precision for lower inference latency and/or model size. The HLS tool fully supports floating point arithmetic, but the

performance would benefit greatly from using a quantizer to change to integer or fixed point arithmetic.

After all the previous modifications, the hardware accelerator for convolutional layers became **5.42 times faster** than the original version. This would make the accelerator performance equivalent to the LEON3-FT processor, or half the performance of the ARM Cortex A9.

3.3 PRELIMINARY TASTE INTEGRATION

Finally, the tool has been preliminary integrated within TASTE environment developing an export/import process that will allow the reuse of CAELUM component between different TASTE MBSE models.

The flow of the model is detailed in the Figure 3-5:

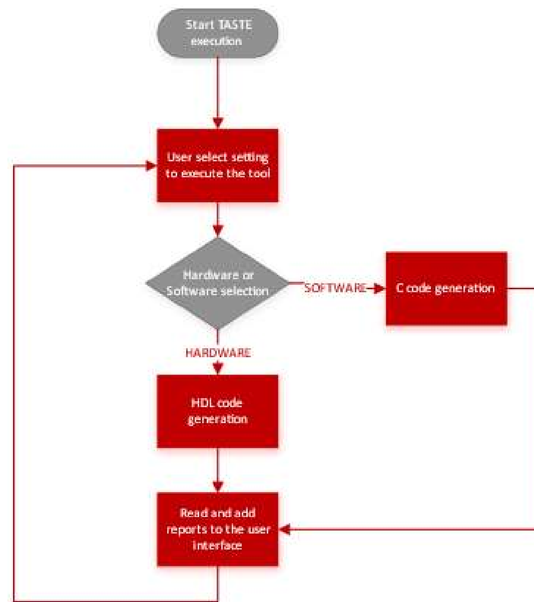

Figure 3-5: TASTE execution flow.

TASTE model has been developed in the Interface View, using component and functions implemented in C++, SDL and PySide to develop an interactive and functional model, whit the simulation capability. In the Interface View a main component named CAELUM has been defined to facilitate the reuse and integration of this component-model in other MBSE TASTE model without configuration effort.

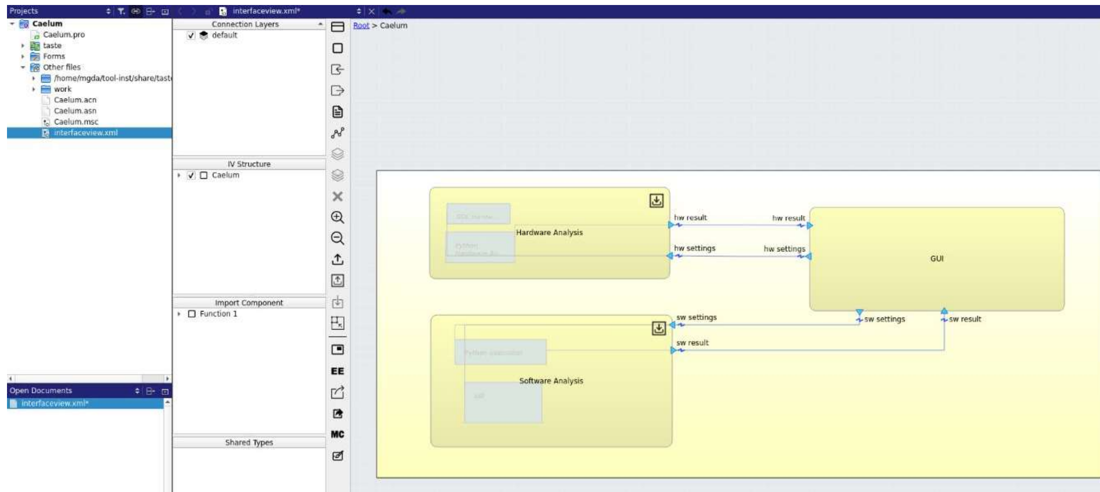

Figure 3-6: Caelum components and functions allocation

The main component contains 2 subcomponents, one for hardware implementation and a second for software implementation. It also contains a functionality named GUI to implement a user interface to control and modify settings during simulation. The hardware and software component also contains two functions each. The first contain a SDL implementation to explain the model subcomponet process and the second function contains a C++ implementation to execute the tool developed during this activity.

GUI Function: This Function has been developed in PySide selecting GUI language in the implementation options. A ui file has been defined to define the graphic interface with the user. The main objective of this function in send and receive the information thought the connections defined in the Interface View. The output GUI signals are defined in TASTE as telecommand and the input GUI signal as telemetry. The telecommand signals contains the required information of the settings selected by the user defined and the telemetries contains the required information to update the table depending on the user settings, this is especially necessary in the Hardware execution where user could execute an operation.

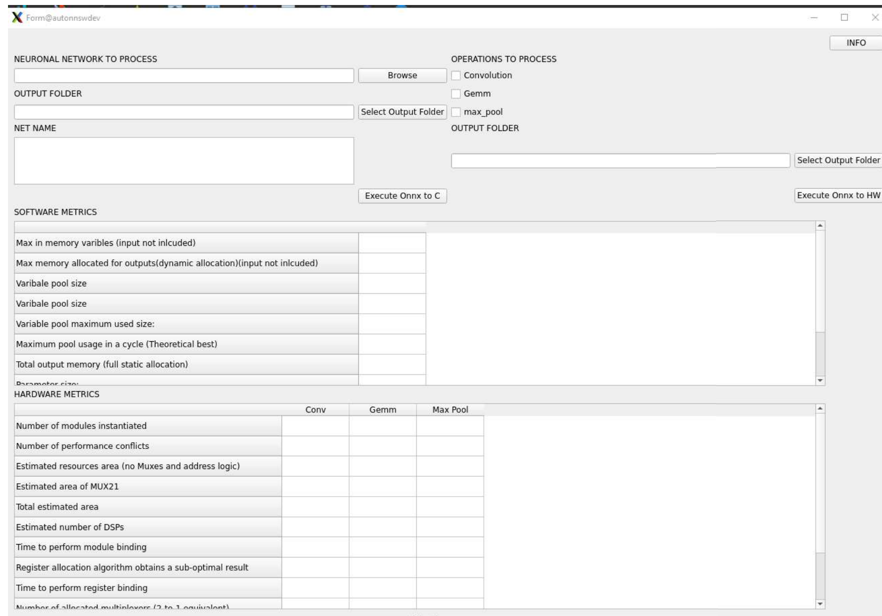

Figure 3-7: GUI Component Execution

Hardware component

- SDL Hardware: This function receives settings and associate parameters and check the status of each parameter to process or not depending on the user previous selection. It has been defined as a state behaviour diagram of the Figure 3-8.

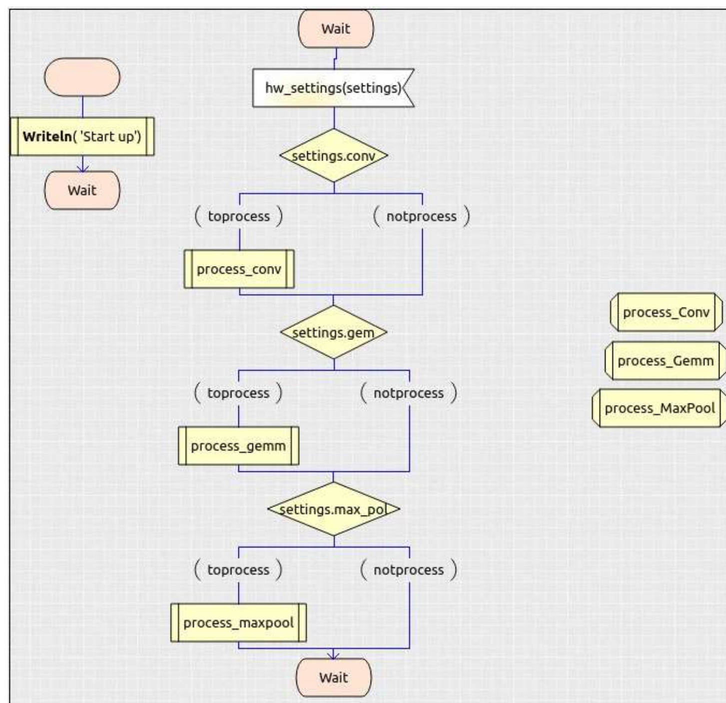

Figure 3-8: Hardware SDL diagram

- Hardware Analysis SH: this function receives the settings selected by the user and execute the hardware operations assigned to the selection. Once execution is ended it emits a signal to update the table whit the obtained results. The execution process could be checked all the moment by the information messages in the terminal.

Software component

- SDL Function: This function receives the software settings and associate parameters and check the validity of each parameter to be set into the configuration.ini file. It has been defined as a state behaviour diagram of the

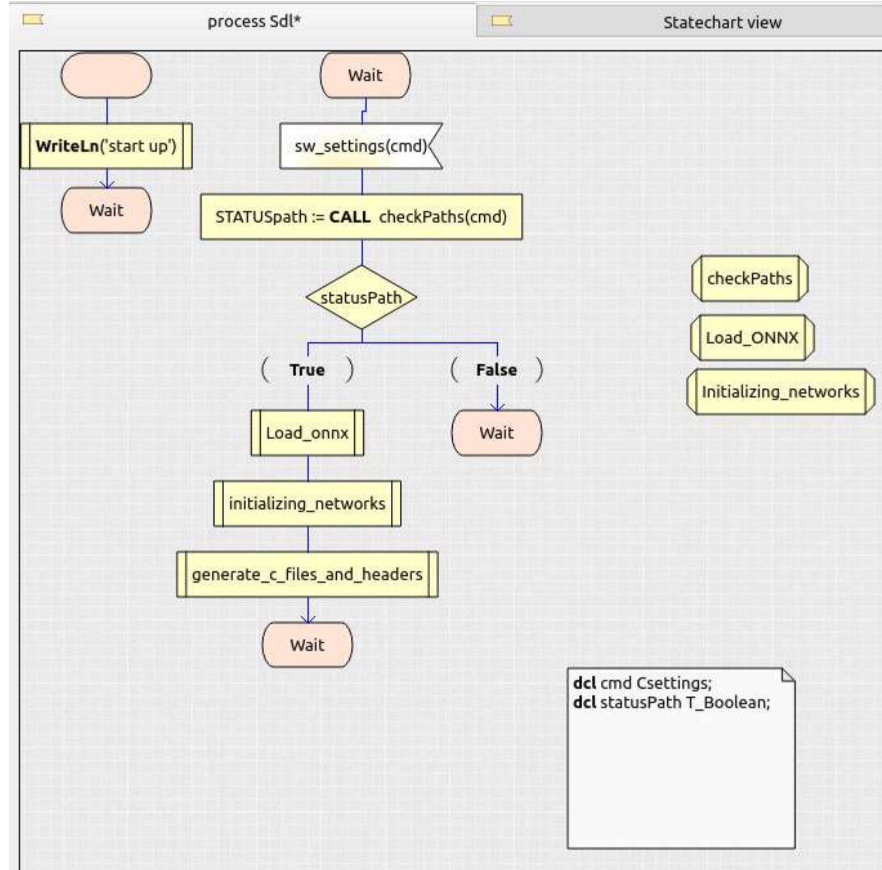

Figure 3-9: software SDL Diagram

- Python Execution Function: this Function receives the onnx setting and the output path and completes the configuration template for each execution. Once template is complete it executes the python script to perform the required onnx to c operations. When the execution is finished this function emits the signal to the gui Function to update the table with the obtained result. The execution process could be checked all the moment by the information messages in the terminal.

END OF DOCUMENT