Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

WP4 Toolchain for heterogeneous multi-node HPC platforms

D4.5 Inter-FPGA Communication SW Stack

http://textarossa.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D4.5 Inter-FPGA Communication SW Stack

Project Start Date: 01/04/2021 **Duration**: 36 months

Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE - ENEA , Italy.*

Deliverable No	D4.5
WP No:	WP4
WP Leader:	INRIA
Due date:	M30
Delivery date:	30/11/2023

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (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)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

Document summary information

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP4 Toolchain for heterogeneous multi-node HPC platforms

Deliverable number:	D4.5					
Deliverable title:	Inter-FPGA Communication SW Stack					
Due date:	M30					
Actual submission date:	02/12/2023					
Editor:	Michele Martinelli					
Authors:	Michele Martinelli, Alessandro Lonardo, Cristian Rossi					
Work package:	WP4					
Dissemination Level:	Public					
No. pages:	30					
Authorized (date):						
Responsible person:	Michele Martinelli					
Status:	Plan	Draft	Working	Final	[✓ Submitted]	Approved

Revision history:

Version	Date	Author	Comment
0.1	25/10/2023	M. Martinelli	Draft structure
0.2	04/11/2023	C. Rossi	Revised and added section "Supervised RAIDER Application"
0.3	17/11/2023	A. Lonardo	Final internal revision.
1.0	30/11/2023	A. Lonardo	Final version.

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Federico Terraneo, Francesco Iannone	28 nov. 2023
Checked by Task Leader	A. Lonardo	28 nov. 2023
Checked by WP Leader	Berenger Bramas	30 nov. 2023
Checked by Project Coordinator	Massimo Celino	30 nov. 2023

COPYRIGHT

Acknowledgements

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.

Please see http://textarossa.eu for more information on the TEXTAROSSA project.

The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).

The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of contents

List of Figures

List of Tables

Partner Report Activity

Tasks 4.4	Development of the software stack for the support of the communication between Vitis HLS kernels through the INFN Communication IP developed in WP2.
Github address	The software and the benchmarks carried out during the activity are publicly accessible on github: https://github.com/APE-group/APEIRON
Technology	Vitis HLS, APEIRON Framework, INFN Communication IP.
Technical development	The technical development has been performed by INFN.

List of Acronyms

ABI	Application Binary Interfaces
AI	Artificial Intelligence
API	Application Program Interface
ASIC	Application Specific Integrating Circuit
AXI	Advanced eXtensible Interface (Xilinx IP)
CNN	Convolution Neural Network
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HDL	Hardware Description Language
HEP	High Energy Physics
HLL	High-Level Language
HLS	High Level Synthesis
HPC	High Performance Computing
HPDA	High Performance Data Analytics
IP	Intellectual Property
ISA	Instruction Set Architecture
KPN	Kahn Process Network
ML	Machine Learning
NN	Neural Network
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
TDAQ	Trigger & Data Acquisition
TDP	Thermal Design Power

Executive Summary

This document describes the APEIRON software stack, consisting in: 1) the host code needed to configure, control and monitor the execution of distributed Vitis HLS applications deployed on multiple FPGAs interconnected through the INFN Communication IP, 2) an HLS library implementing the communication primitives and the interfacing between the Communication IP and the HLS computing kernels, and 3) the software tool that generates the target design, linking automatically the Communication IP and the HLS computing kernels, to allow the generation of the FPGA bitstreams.

In section 2 we describe the architecture of the runtime software stack and introduce the Supervisor component in charge of monitoring the status of the nodes and the execution of the distributed HLS applications.

In the APEIRON framework the communication between HLS kernels is expressed through HAPECOM: a header-only lightweight C++ library based on non-blocking *send()* and blocking *receive()* communication primitives; the library is described in section 3.

Instead of using the standard Vitis flow to generate the final integrated design, users must just prepare a YAML configuration file describing the attributes of each HLS computational kernel with APEIRON. Starting from this, the framework links the Communication IP and the HLS kernels that are connected to it and generates the bitstream for the overall design, as outlined in section 4.

Finally, in section 5 we describe the implementation of a distributed HLS application to show the capabilities of the software stack enabling the scaling of performance with respect to the number of interconnected FPGAs.

1. Introduction

The APEIRON system software consists of three main components: a host software stack providing runtime support for the multi-FPGA execution platform, a lightweight C++ HLS communication library (HAPECOM) based on non-blocking *send()* and blocking *receive()* operations, and of a tool for the automatic linking of computational tasks and Communication IP for FPGA bitstream generation. The APEIRON framework mainly offers hardware and software support to execute real-time dataflow applications on a network of interconnected FPGAs. Developers can deploy scalable applications on a multi-FPGAs system via a simple dataflow programming model inspired by Kahn processing networks, and can run these applications on multiple devices, monitoring the whole execution from a single node.

1.1. Relationship with Project Objectives

The foundational ideas motivating the APEIRON framework, and thus driving its software stack design and development, are the following:

1. The direct communication between computing tasks deployed on FPGAs avoids the involvement of the CPUs and system bus resources in the data transfers, improving the energy efficiency of the execution platform.
2. Bypassing the intervention of the host network stack, communication latency is reduced while bandwidth for small massages is increased.
3. Since communication operations are implemented on a completely “hardware” path, deterministic latency is achieved, in accordance with the real-time requirements.

These considerations are strictly related to the TEXTAROSSA project objectives:

- **Objective 1 - Energy efficiency**. APEIRON addresses this objective enabling the complete offload of the streaming processing to FPGA devices. Furthermore, avoiding the involvement of the CPUs and system bus resources in data transfers improves the energy efficiency of the multi-FPGA execution platform.
- **Objective 2 - Sustained application performance**. The sustained application performance of distributed streaming applications, such as the RAIDER use case, are strongly affected by the performance of the network system. Implementing a direct FPGA to FPGA interconnect and bypassing the host network stack, allows to keep the communication latency in the sub-microsecond range and to increase the bandwidth for small messages.
- **Objective 4 - Seamless integration of reconfigurable accelerators**. The APEIRON framework leverages the Vitis HLS workflow, extending it to a multi-FPGA execution platform through a lightweight HLS communication library (HAPECOM) at programming level, and through a simple configuration system for the deployment of the distributed application to the multi-FPGA execution platform.
- **Objective 5 - Development of new IPs**. The INFN Communication IP is the key enabling technology behind the APEIRON framework, allowing direct low-latency intra/inter FPGA communications between HLS kernels.

- **Objective 6 - Integrated Development Platform**. The ARMv8 based IDV-E represents the reference execution platform for the APEIRON runtime in the TEXTAROSSA project. Nevertheless, the framework has been developed and extensively tested on a X86_64 based small scale cluster in our lab, demonstrating the portability of the software stack to different host ISAs.

The objectives are also related to the strategic goals of the project:

- **Strategic Goal #2**: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The APEIRON framework software is developed following the open-source model and is freely available in its GitHub repository (https://github.com/APE-group/APEIRON).

- **Strategic Goal #3**: Opening of new usage domains. The APEIRON frameworks aims at offering hardware and software support for running real-time dataflow applications on a network of interconnected FPGAs, leveraging on the Vitis HLS tool. We believe that it has the potential to ease the development and to support the efficient execution of a wide class of applications suited to be executed on a multi-FPGA platform, such as but not limited to real-time HPDA ones.

2. Host Software Stack

The software stack architecture was already presented in deliverable D4.1, we report it in Figure 2.1 for convenience.

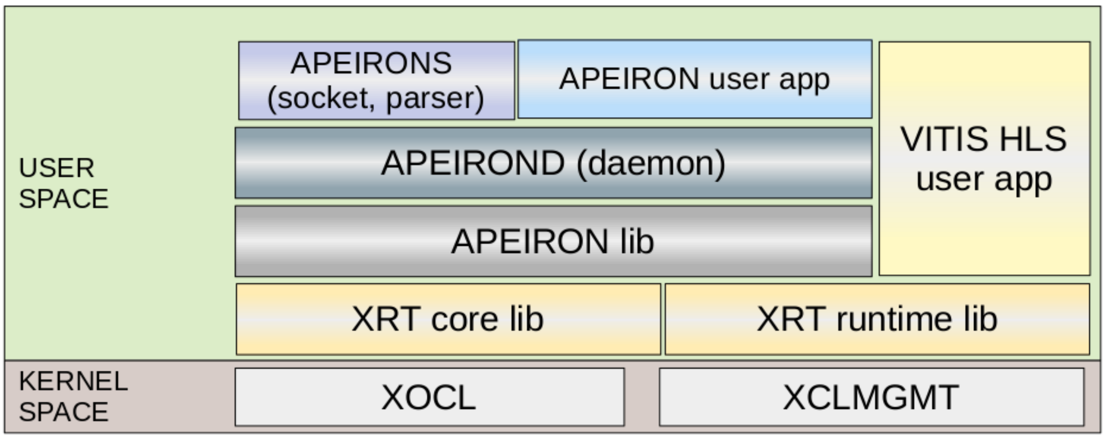


Figure 2.1 APEIRON Software Stack scheme

XOCL and XCLMGMT modules are developed by Xilinx. The first (*XOCL*, PCIe User Physical Function Driver Interfaces) defines IOCTL system call command codes and associated structures for interacting with FPGA platforms, while the latter (*XCLMGMT*, PCIe Management Physical Function) is the PCIe Kernel Driver for Management Physical Function. Xilinx Runtime library (XRT) is an open-source easy to use software stack that facilitates management and usage of FPGA/ACAP devices built by Xilinx [1].

The problem using the XRT core lib and XRT runtime lib (which interacts directly with XOCL and XCLMGMT exposed APIs) is that some functions (i.e. register read and register write) require a "handler" returned during the bitstream board flashing. This implies that only one process is allowed to "read" or "write" hardware registers and the board must be re-initialized (and then the execution restarted) for every new or concurrent process. This is obviously not compatible with a library, which needs to allow different processes to operate on the same board concurrently.

2.1. Apeironlib

This module has two main objectives: wrapping the XRT functions and implementing a standard output redirect to have a circular buffer to be used as log.

1. wrapper of the XRT functions, allowing the host user code to acquire a *handler* over the physical hardware. This partially solves the problem of having a single process allowed to use the hardware without re-flashing the board to handle the kernels. With respect to the deliverable D4.1, the library was extended by adding more functions, the complete list is described in Table 2.1.
2. a logging circular buffer, very useful when used in conjunction with the Supervisor (see section 2.4). This buffer is automatically filled when the user log information. An apeiron::cout is provided to be used as a normal std::cout object, which is in charge of simultaneously printing the information on the screen and storing it in the circular buffer.

Function name	return	Description
device_open	error_t	Open the device saving the "pointer" to the hardware inside the private attributes of the handler class
device_dump	void	Used to print hardware attributes, mainly for debug purposes
device_reset	error_t	Uses XRT core library functions to reset the FPGA
xclbin_load(std::string xclbin_fnm)	error_t	Loads a xclb files into the FPGA board
read_register(const xrt::ip& kernel, uint32_t reg)	uint32_t	Read an IP-exposed register
write_register(xrt::ip kernel, uint32_t reg, uint32_t val)	error_t	Write a certain value to a IP-exposed register
report_power_consumption	float	Measure the current power consumption (XRT API used)
report_fpga_thermal	float	Measure the current thermal status (XRT API used)
report_kernel_state	state	Current kernel status (e.g. running, waiting, syncing...)
get_current_xclbin	string	Return the current XCLBIN file used to flash the board
get_current_deviceID	int	In systems where multiple boards are installed, this functions returns the ID associated with the current FPGAs.

Table 2.1 The APEIRON lib API

2.2. Apeirond

Apeirond is a persistent daemon / server used to manage multiple access requests from user apps to the board. It uses the apeironlib exposed APIs to operate on the physical hardware.
Apeirond component has a persistent handler (an instance of the apeironlib) over the FPGA board, which is used to perform the actions exposed by the library.
Note that it’s possible to have multiple boards installed on the same host, in this case one handler is created for each board during the initialization phase. The user application then specifies the ID of the board to use when connecting.

The principle of functioning is a standard server/client model: the server is always waiting for new connections from clients on a specific port. At the arrival of a new client connection, a thread is generated to handle the request. This allows the server to answer multiple requests (from the same client or from different clients) at the same time. For example, during the execution of one kernel it is still possible to read registers in real time.
Every client connection is handled by a different file descriptor, used to keep track of the connected clients. Periodically the file descriptors are monitored to assure the connection persistency and clean data structures in case of disconnection.
A “connectionHandler” object is used to keep track of the connection status, basically reading requests from the client and sending back the responses.

2.3. Apeirons

The component responsible for the connection handling and the request parsing is called apeirons.
This module receives commands from through the network via apeirons socket, exposing the apeironlib APIs to users over the network. The available commands are the same discussed for apeironlib, but exposed over the network.
The protocol used is based on a TCP/IP socket and the messages are serialized and deserialized in JSON format to simplify the parsing phase.

For example, a “read register” command from a client is serialized as follows (request for the value of register “9” of the default IP):

```
{"register":"9","request":"read_reg"}
```

A possible apeirond answer would be (register 0 contains value 0xe):

```
{"register":"0","response":"read_reg","value":"0xe"}
```

2.4. Supervisor

This is the component used to monitor the status of the nodes in the network.
The idea is to have a supervisor to manage multiple instances of the software stack distributed over the network, as it is represented in Figure 2.2.

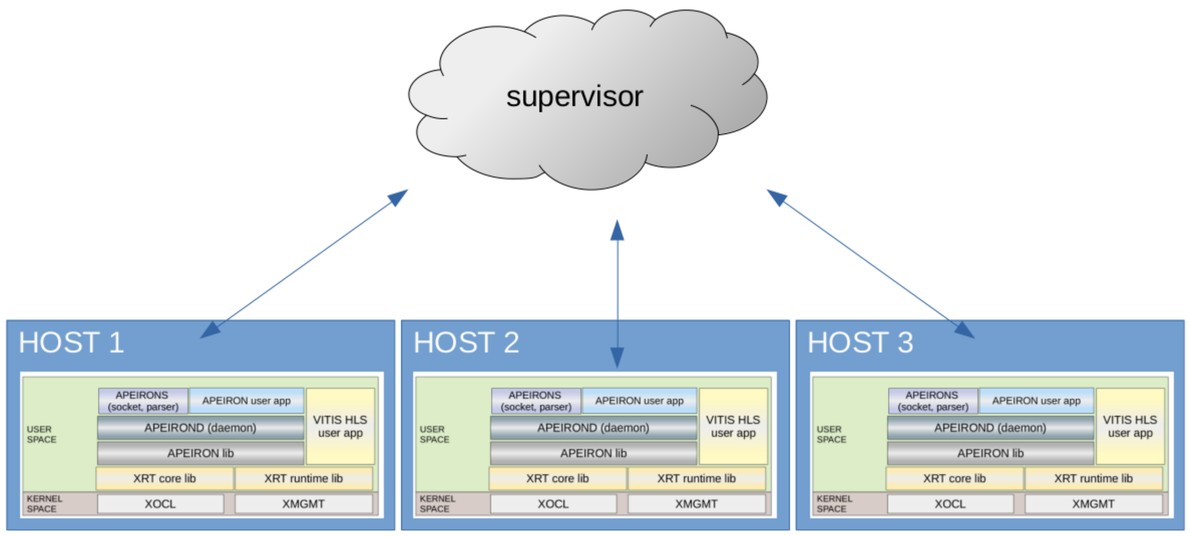
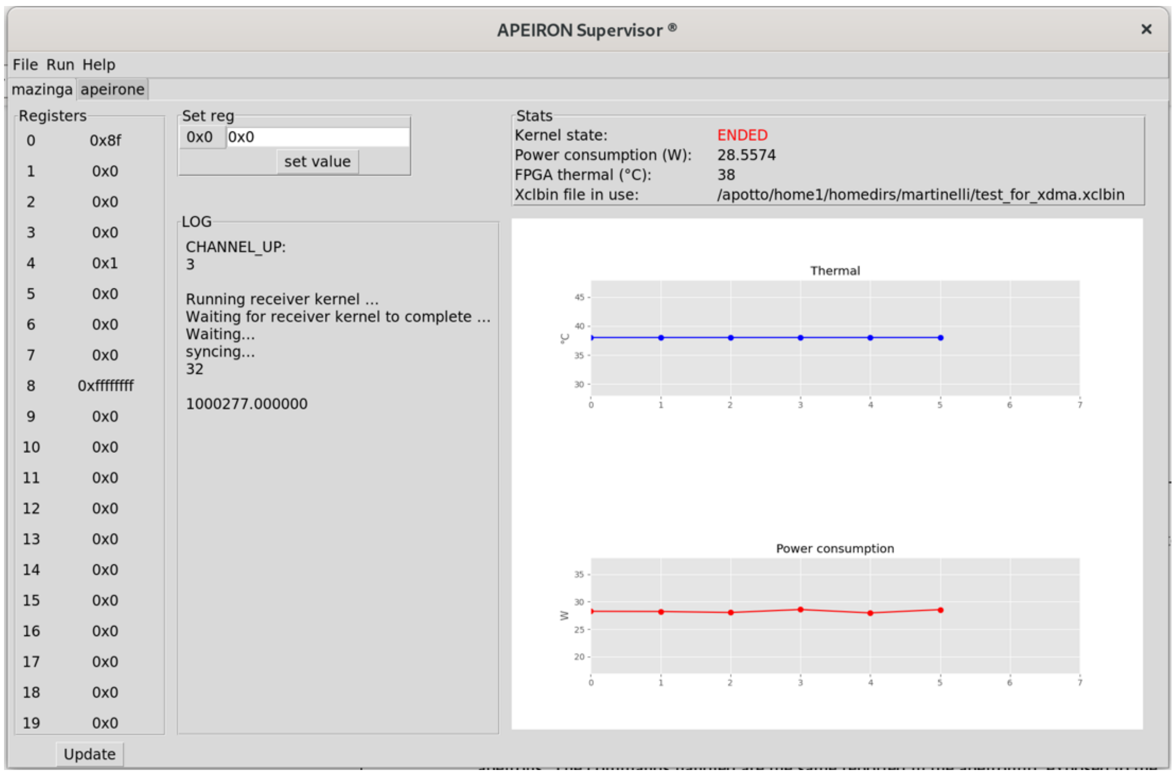


Figure 2.2: Supervisor working scheme on different host executions

Host names (and board id, if multiple FPGAs are available in the remote system) are passed as command line parameters to the script in charge of creating the GUI and handling the user inputs.

In the current implementation, the supervisor is written in Python language, using TKinter package [2] to generate the Graphical User Interface.

The user interface is represented in Figure 2.3.



Registers

0	0x8f
1	0x0
2	0x0
3	0x0
4	0x1
5	0x0
6	0x0
7	0x0
8	0xffffffff
9	0x0
10	0x0
11	0x0
12	0x0
13	0x0
14	0x0
15	0x0
16	0x0
17	0x0
18	0x0
19	0x0

Set reg: 0x0 0x0

LOG
```
CHANNEL_UP:
3

Running receiver kernel ...
Waiting for receiver kernel to complete ...
Waiting...
syncing...
32

1000277.000000
```

Stats

Kernel state:	ENDED
Power consumption (W):	28.5574
FPGA thermal (°C):	38
Xclbin file in use:	/apotto/home1/homedirs/martinelli/test_for_xdma.xclbin

Figure 2.3: Supervisor user interface

The internal registers (on each node where apeirond daemon is running) can be monitored, and the register content can be set by using the “Set reg” input text area.

The LOG section is used to monitor the standard output of the remote machine: using a circular buffer the output is saved by the remote host and sent to the supervisor periodically, together with the “kernel status”, “power consumption” and “thermal” information for each node. Power consumption and thermal status are also plotted in real time.

In the “file” menu the user can flash a bitstream on each of the remote hosts using the dedicated GUI, selecting the right bitstream file.

In the “run” menu the user can select the “kernel” file to run on a specific host or on all the remote hosts (more details in the next section).

It’s important to note that both the bitstream file and the XRT host program must be in a filesystem shared by the supervisor and the remote host.

2.5. Running host application remotely

The mechanism used for running the host application controlling the HLS kernels is the dynamic linking loader. In the apeirond server, the function dlopen() loads the dynamic library file specified by the user, running the function named “run_kernel” by using the dlsym() standard call.

The user application controlling the HLS kernels will be compiled as dynamic library (usually a .so file) and then the file is used to run the application through the apeirond server.

From the user perspective, the only argument passed to the “run_kernel” function is the “fpga_handler”, which is an instance of the apeiron_handler object, to be used to call the function inside the user kernel using the functions list reported in the apeironlib section.

In the following pseudocode we report an example of a simple “send/receive” host application to be compiled as dynamic library:

```
extern "C" void run_kernel (apeiron::Handler *fpga_handler) {

/* in this moment fpga_handler is already initialized by the upper level*/

fpga_handler->set_state_initializing();

// create the kernel handle
 xrt::kernel user_kernel(fpga_handler->get_device(), fpga_handler->get_uuid(),
"krnl_usr:{krnl_user_1}");

//Receive buffer intialization
xrt::bo recv_buffer(fpga_handler->get_device(), buf_size, user_kernel.group_id(6));
stream_data_t *recv_buffer_map = recv_buffer.map<stream_data_t*>();

//Send buffer initialization
xrt::bo send_buffer(fpga_handler->get_device(), packet_size, user_kernel.group_id(5));
stream_data_t *send_buffer_map = send_buffer.map<stream_data_t*>();
memset(send_buffer_map, 0, packet_size);

//set some application-specific registers
fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),4, 0x1);
fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),6, 0);

//write some LOG messages
apeiron::cout << "CHANNEL_UP: " << std::to_string(fpga_handler->read_register(fpga_handler-
>get_ip("TextaRossa_switch"),67)) ;
apeiron::cout << "Running receiver kernel ...\n";

//start send test
gettimeofday(&startTime,NULL);
send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
xrt::run  krnl_usr_run = user_kernel(npackets, npackets, packet_size, send_buffer,
recv_buffer);

//start receiver
apeiron::cout << "Waiting for receiver kernel to complete ...";
fpga_handler->set_state_waiting();
krnl_usr_run.wait();
gettimeofday(&endTime,NULL);

//LOG the results
double elapsedTime = elapsed(startTime,endTime);
apeiron::cout <<  std::to_string(packet_size) << " \t "<<
std::to_string(elapsedTime/(2*npackets));
}
```
Listing 2.1: Supervised host application example

3. Communication API: HAPECOM

The communication between kernels is expressed through HAPECOM: a lightweight C++ API based on non-blocking *send()* and blocking *receive()* operations. This simple API allows the HLS developer to perform communications between kernels, either deployed on the same FPGA (intra-node communication) or on different FPGAs (inter-node communication), as represented in Figure 3.1, without knowing the details of the underlying packet communication protocol.

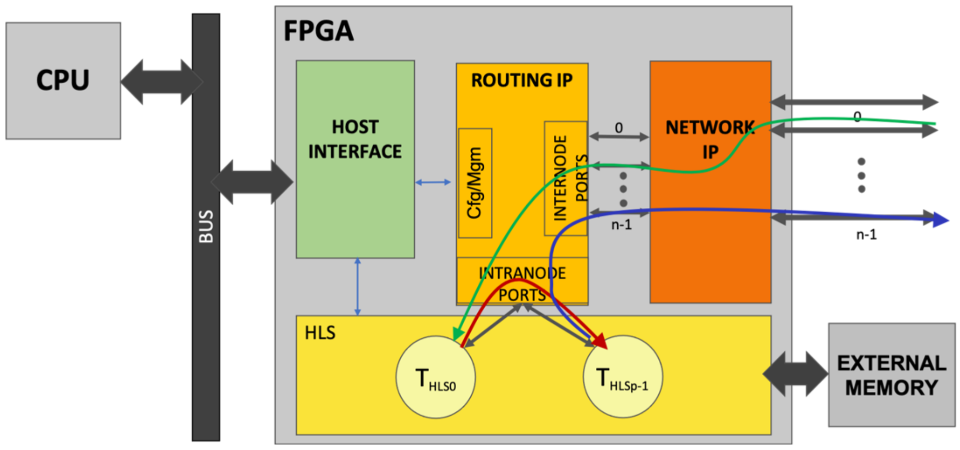


Figure 3.1: HLS kernels performing intra-node (red line) and inter-node (green line – receive, blue line– send) communications

The HAPECOM Communication API can be represented with the following pseudo-code:

```
size_t send (msg, size, dest_node, task_id, ch_id);
size_t receive (ch_id, recv_buf);
```

where:

- `msg` is the message to be sent and `size` is its size in Bytes;
- `dest_node` is the n-Dim coordinate of the destination node (FPGA) in the n-Dim torus network;
- `task_id` is the local-to-node receiving task (kernel) identifier (0-3);
- `ch_id` is the local-to-task receiving FIFO (channel) identifier (0-127);
- `recv_buf` is the receive buffer of the destination HLS kernel.

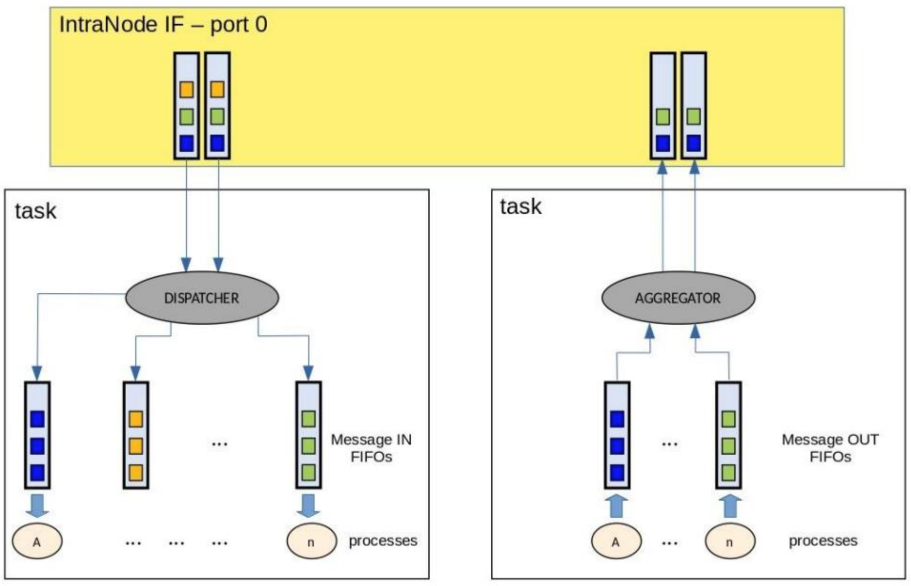


Figure 3.2: Interface between Intranode Port 0 and the corresponding HLS Task mediated by Aggregator and Dispatcher

The Communication Library leverages AXI4-Stream Side-Channels to encode all the information needed to forge the packet header. Two APEIRON HLS IPs manage the adaptation toward/from IntraNode ports of the Routing IP: they are called Aggregator and Dispatcher, see Figure 3.2. The Dispatcher receives incoming packets from the Routing IP and forwards them to the destination receive channel, according to the relevant fields of the header. The Aggregator receives outgoing packets from the task and forges the packet header, then filling the header/data FIFOs of the Routing IP IntraNode port.

Diving into the HAPECOM library code, Listing 3.1 shows the data structure representing the header and footer apenet protocol packet adopted in TEXTAROSSA [3].

```
typedef union {
        struct __attribute__((packed)) {

                unsigned long virt_chan      :  5;
                unsigned long proc_id        : 16;
                unsigned long dest_x         :  6;
                unsigned long dest_y         :  5;
                unsigned long dest_z         :  5;
                unsigned long intra_dest     :  4;
                unsigned long reserved       :  1;
                unsigned long out_of_lattice :  1;
                unsigned long packet_type    :  5;
                unsigned long packet_size    : 14;
                unsigned long dest_addr      : 48;
                unsigned long num_of_hops    : 10;
                unsigned long edac           :  8;

        } s;
        uint32_t l[8];
        uint64_t u[4];
} apenet_header_t;

typedef union {
        uint32_t l[8];
        uint64_t u[4];
} apenet_footer_t;
```

Listing 3.1: apenet protocol packet header and footer.

The usage of the various fields of the header is shown in Listing 3.2 where the implementation of the send() and receive() functions is reported. The main part of each of these functions is to manage the apenet protocol header and footer: in the send() function, we can see how the header is forged by filling each of its significant fields with the parameters of given at the function code; while in the receive() function, the header bit address relative to the packet size (hd.s.packet_size, bounded by size_start_bitpos and size_end_bitpos enumerations) is read to complete the reception.
Since the words composing the packets, sent from the HLS task to the dispatcher travel on the same streaming channels (defined as an hls::stream<ap_uint<256>>), in send() operations we must manage in a univocal way header, payload and footer words while filling the FIFOs. For this purpose, the apenet_2_word() function has the task of converting the apenet protocol packet header and footer data structure to a plain representation as a 256 bit unsigned word (defined in HAPECOM as word_t).

```
typedef ap_uint<256> word_t;
typedef hls::stream<word_t> message_stream_t;
typedef hls::stream<apenet_header_t> header_stream_t;
typedef short channel_id_t;
typedef short task_id_t;

int receive(channel_id_t ch_id, word_t *buff,
            message_stream_t message_data_in[N_INPUT_CHANNELS]){

        word_t hdr = message_data_in[ch_id].read();

        unsigned size = hdr.range(size_start_bitpos, size_end_bitpos);

        unsigned nwords = (size & (sizeof(word_t)-1)) ? (size/sizeof(word_t)+1) :
size/sizeof(word_t);

        for (unsigned i = 0; i < nwords; ++i){
        #pragma HLS pipeline
                buff[i] = message_data_in[ch_id].read();
        }

        word_t ftr = message_data_in[ch_id].read();

        return size;

}

size_t send(word_t *buff, size_t size, int coord,
            task_id_t task_id, channel_id_t ch_id,
            message_stream_t message_data_out[N_OUTPUT_CHANNELS])
```

```
{

  //create and write hdr + data + footer
  if(size>0){
        apenet_header_t tmp_hd = {0};
        tmp_hd.s.dest_x = coord & 0b111111;
        tmp_hd.s.dest_y = (coord>>6) & 0b11111;
        tmp_hd.s.dest_z = (coord>>11) & 0b11111;
        tmp_hd.s.intra_dest = task_id;
        tmp_hd.s.packet_size = size; //packet_size
        tmp_hd.s.dest_addr = 0xfafbfcfd;
        tmp_hd.s.proc_id = ch_id;
        word_t tmp_header = apenet_2_word(tmp_hd);

        message_data_out[ch_id].write(tmp_header); //header

        unsigned nwords = (size & (sizeof(word_t)-1)) ? (size/sizeof(word_t)+1) :
size/sizeof(word_t);

        for (unsigned i = 0; i < nwords; ++i){
           #pragma HLS pipeline
           #pragma HLS LOOP_TRIPCOUNT min=1 max=128
           message_data_out[ch_id].write(buff[i]); //payload
        }

        apenet_header_t tmp_ftr = {0};
        tmp_ftr.s.dest_addr = 0xaaaeabac;
        tmp_ftr.s.edac = 0x99;

        word_t tmp_footer = apenet_2_word(tmp_ftr);
        message_data_out[ch_id].write(tmp_footer); //footer
  }
}
```

Listing 3.2: HAPECOM send/receive implementation.

To conclude this section, we report the implementation of the Aggregator (Listing 3.3) and of the Dispatcher (Listing 3.4) HLS IPs managing the interface between the HLS computing kernels and the Routing IP, as shown in Figure 3.2.

```
template <unsigned NCHAN>
void aggregator_template(
                message_stream_t fifo_data_in[NCHAN],
                header_stream_t &fifo_hdr_out,
                message_stream_t &fifo_data_out)
{
  #pragma HLS INLINE

  for(unsigned ch=0; ch<NCHAN; ch++){
  #pragma HLS LOOP_TRIPCOUNT min=1 max=NCHAN
  #pragma HLS unroll

    if(!fifo_data_in[ch].empty()){

      //Send header
      apenet_header_t hdr = {0};
      auto tmp = fifo_data_in[ch].read();
      hdr = word_2_apenet(tmp);
      fifo_hdr_out.write(hdr);

      unsigned size = hdr.s.packet_size;
      unsigned nwords = (size & (sizeof(word_t)-1)) ? (size/sizeof(word_t)+1) :
size/sizeof(word_t);

      for(unsigned i=0; i<nwords; i++){
      #pragma HLS LOOP_TRIPCOUNT min=1 max=128
        fifo_data_out.write(fifo_data_in[ch].read());
      }

      apenet_header_t ftr = {0};
      auto tmp2 = fifo_data_in[ch].read();

      ftr = word_2_apenet(tmp2);

      fifo_hdr_out.write(ftr);
    }
  }
}
```
Listing 3.3: Implementation of the Aggregator HLS IP

The Aggregator collects data from all the output channels of an HLS kernel and forwards them to the attached port of the Routing IP, while the Dispatcher collects packets from the connected Routing IP port and streams them to the input channel of the HLS kernel. Similarly to what has been described for the `apenet_2_word()` function, the `word_2_apenet()` function is used to convert apenet protocol header and footer, arriving to the Aggregator as 256 bit unsigned words, back to their proper data structure (Listing 3.1) which can be streamed to the connected Routing IP port. This is because each intranode Communication IP port has two separate channels: one for header and footer stream (`hls::stream<apenet_header_t>` or `header_stream_t`), and a second one for payload stream (`hls::stream<word_t>` or `message_stream_t`).

```
  template <unsigned NCHAN>
  void reader(
              header_stream_t &fifo_hdr_in,
              message_stream_t &fifo_data_in,
              message_stream_t fifo_data_out[NCHAN])
  {
    apenet_header_t hdr = fifo_hdr_in.read();
    unsigned input_channel = hdr.s.proc_id;
    unsigned size = hdr.s.packet_size;
    word_t tmp_header = apenet_2_word(hdr);
    fifo_data_out[input_channel].write(tmp_header);

    unsigned nwords = (size & (sizeof(word_t)-1)) ? (size/sizeof(word_t)+1) :
size/sizeof(word_t);

    for (unsigned i = 0; i < nwords; ++i) {
      #pragma HLS PIPELINE II=1
      #pragma HLS LOOP_TRIPCOUNT min=1 max=256
      fifo_data_out[input_channel].write(fifo_data_in.read());
    }

    apenet_header_t ftr = fifo_hdr_in.read();
    word_t tmp_footer = apenet_2_word(ftr);
    fifo_data_out[input_channel].write(tmp_footer); //footer

  }
```

```
template <unsigned NCHAN, unsigned HD_DEPTH, unsigned DT_DEPTH>
void dispatcher_template(
                header_stream_t &fifo_hdr_in,
                message_stream_t &fifo_data_in,
                message_stream_t fifo_data_out[NCHAN])
{
  #pragma HLS INLINE
  reader<NCHAN>(fifo_hdr_in, fifo_data_in, fifo_data_out);
}
```

Listing 3.4: Implementation of the Dispatcher HLS IP

4. FPGA bitstream generation

APEIRON framework links the Communication IP and the HLS kernels of the system design requested by using a YAML configuration file describing the attributes of each HLS kernel, namely:

- Number of input and output channels.
- IntraNode port of the Communication IP to which the kernel is connected to.
- Global clock frequency of the system.

An example of this configuration file is reported in Listing 4.1, for a design integrating the Communication IP and a single HLS task (named `example_apeiron_task_1`) connected to intra-node port 0 of the Routing IP through an Aggregator/Dispatcher having four input/output channels; furthermore, the design sports two inter-node channels and its target operating clock frequency is 150 MHz.

```
kernels:
 -name: example_apeiron_task_1
    input_channels: 4
    output_channels: 4
    switch_port: 0

config:
  freq: 150
  links: 2
```

Listing 4.1: Example of APEIRON YAML configuration file

From this YAML description, the APEIRON framework links the Communication IP and the HLS kernels that are connected to it and generates the bitstream for the overall design. HLS kernels written by the user to be linked to the Communication IP must implement a prototype of this form:

```
void example_apeiron_task(
                          [optional kernel-specific list of parameters]
                          message_stream_t message_data_in[N_INPUT_CHANNELS],
                          message_stream_t message_data_out[N_OUTPUT_CHANNELS])
```

In this way, the HLS kernel implements a generic stream interface for each communication channel based on the AXI4-Stream protocol that is properly connected to its corresponding Communication IP intranode port through the Aggregator and Dispatcher components in the final design.

If the instantiation interval on the receiving side must be kept low to maximize the design throughput, it is not advisable to use the blocking `receive()` function, and the direct access to the `message_data_in` stream through the `read()` method should be used instead, parallelizing data reads and processing, as shown in Listing 5.3.

5. The APEIRON software stack in action

We describe a preliminary version of the INFN RAIDER application as a demonstration of the use of the APEIRON software stack to develop, run and monitor a distributed multi-FPGA Vitis HLS application.

RAIDERS's task is to perform particle identification (PID) on the stream of events generated by the RICH (Ring Imaging CHerenkov) detector in the CERN NA62 experiment [4] at a rate of about 10 MHz, using neural networks.

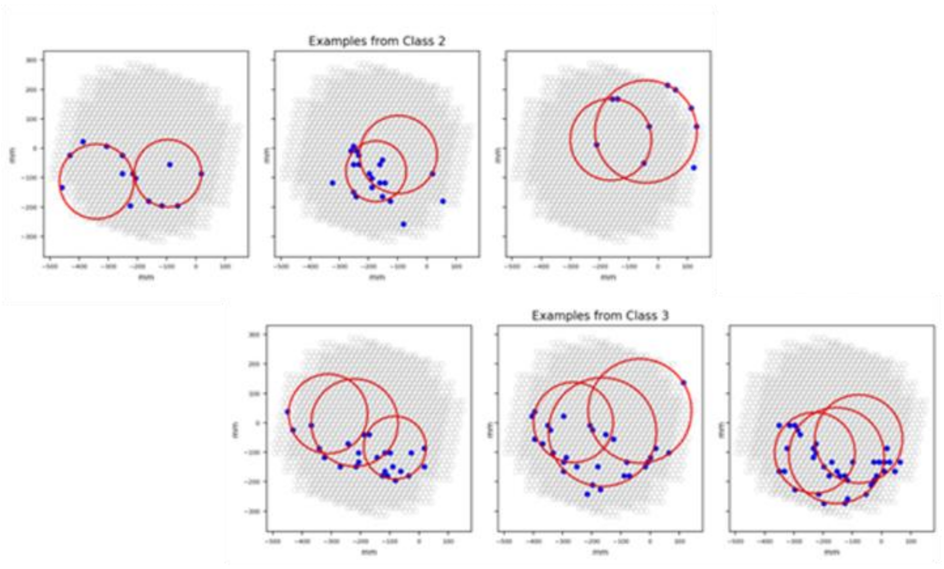


Figure 5.1: Examples of events belonging to class 2 and 3 (2 or >=3 charged particles) as detected by the array of RICH photomultipliers (blu dots are the hit photomultipliers, red circles are the tracks reconstructed offline by the NA62 experiment offline analysis software framework)

The inference task consists in providing an estimate for the number of charged particles (0, 1, 2, >=3) for any RICH detector event, that corresponds to the number of ring tracks that can be reconstructed from the pattern of photomultipliers that have been illuminated (hit) by the Cherenkov light cone emitted by a charged particle traversing the detector, as shown in Figure 5.1. The inference task is implemented with a preprocessing stage (*Imagifier*) followed by a Convolutional Neural Network (*CNN*). The CNN model has been developed using Tensorflow/Keras and deployed on FPGA with the HLS4ML [5] software package, refer to deliverable D4.8 - Framework for efficient CNNs inference on a TEXTAROSSA node for a complete description of this workflow. The CNN receives in input the output

of the Imagifier kernel, a 16x16 image of the hit photomultipliers (PMTs) map for each physics event and produces an estimate for the number of charged particles it contains. Considering the high event rate of the experiment, sustaining an adequate processing throughput is the main challenge for such a system.
In deliverable D6.2 – Initial Application Benchmarks and Results we reported results obtained on two single-FPGA implementations of the application, including one and two inference pipelines respectively. Here we scale the number of Xilinx Alveo U200 FPGAs from 2 to 4, in order to increase further the reconstruction throughput, deploying the HLS processing tasks according to what is shown in Figure 5.2.

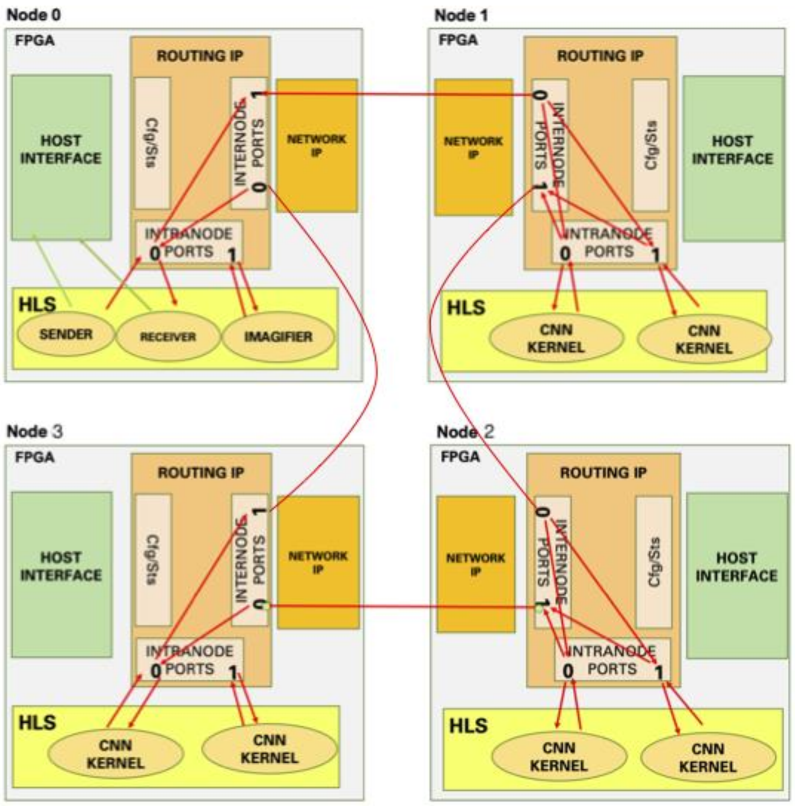


Figure 5.2: RAIDER HLS processing tasks deployed on the 4 FPGAs execution platform.

As shown in Figure 5.2, there are two kinds of nodes (and hence the overall Multi-FPGA design includes two different bistreams):

1. **I/O and Preprocessing node**: data are loaded from Host memory and sent through the network via an HLS kernel ("sender"). Data are then processed by the Imagifier HLS kernel which turns the PMT hitlist information into a 256bit word (16x16 B&W image) that is sent to the Computing node through the external links. As a second task, this node is in charge of receiving the output of the CNN computation and storing it on Host memory via an HLS kernel ("receiver"). The processing time, from the first packet sent to the last received, is measured on this node host.

2. **Computing node**: images coming from external links are taken as input and dispatched to one or both the CNN HLS kernels (depending on the configuration) to compute the predictions. Results are then sent back to the I/O and preprocessing node.

Since for each type of node we need a different bitstream, two different YAML configuration files are needed for APEIRON framework to generate the firmware to be flashed on each kind of node. These two files are reported below (note that since this version of the application is based on the preliminary version of the Communication IP, described in deliverable D2.8 - IP for low-latency inter-node communication links, part 1, the clock frequency of the overall design to be synthesized is set to 100 MHz).

```
kernels:
 -name: sender
    input_channels: 0
    output_channels: 1
    switch_port: 0

 -name: receiver
    input_channels: 1
    output_channels: 0
    switch_port: 0

  -name: imagifier
    input_channels: 1
    output_channels: 1
    switch_port: 1

config:
  freq: 100
  links: 2
```

Listing 5.1: “Preprocessing node" YAML configuration file

The generated Aggregator and Dispatcher connected to Port 0 will have one input and one output channels respectively, as the ones connected to Port 1.

```
kernels:
 -name: cnn_kernel
    input_channels: 1
    output_channels: 1
    switch_port: 0
 -name: cnn_kernel
    input_channels: 1
    output_channels: 1
    switch_port: 1

config:
  freq: 100
  links: 2
```

Listing 5.2: “Computing node" YAML configuration file

As in the Preprocessing Node case, the generated Aggregator and Dispatcher connected to Port 0 will have one input and one output channels respectively, as the ones connected to Port 1.

```
extern "C" void imagifier (unsigned int nports, unsigned int nboards,
                           message_stream_t  message_data_out[N_INPUT_CHANNELS],
                           message_stream_t  message_data_in[N_OUTPUT_CHANNELS]) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=message_data_in
#pragma HLS interface axis port=message_data_out
        static unsigned ch_id = 0;
        word_t word;
        word_t buff_out[5];
        ap_uint<IMAGE_SIZE*IMAGE_SIZE> image = 0;
        size_t size=0;
        for (int i=0; i<MAX_WORD; i++) {
         //Streaming events reception ==> reading of HAPECOM and event headers
            if(size==0){
                   auto hd = message_data_in[ch_id].read();  //HAPECOM header

                   size = hd.range(size_start_bitpos,size_end_bitpos)/sizeof(word_t);
                   buff_out[0] = message_data_in[ch_id].read(); //event header
                   size--;
            }
            if(size>0) word = message_data_in[ch_id].read();

            for (int j=0; j<MAX_HIT_PER_WORD; j++) {
#pragma HLS pipeline
                if (size==0) continue;
                unsigned short pmt = word.range((j+1)*16-1, j*16);
                if (pmt==0) continue;
                auto x = x_bin[pmt];
                auto y = y_bin[pmt];
                if (x>=0 && y>=0)  image.set (x+IMAGE_SIZE*y);
            }
            if(size>0) size--;
            if (size==0){
                auto ftr = message_data_in[ch_id].read();
                break;
            }
        }

        if(size>0){
            while(size>0){
                auto flush = message_data_in[ch_id].read();
                size--;
            }
            auto ftr = message_data_in[ch_id].read();
        }

        buff_out[1] = image.range(127,0);
        buff_out[2] = image.range(255,128);

        static unsigned task_id = 0;
        static unsigned dest_coord = 1;

        send(buff_out, 3*sizeof(word_t), dest_coord, task_id, ch_id, message_data_out);

        ch_id = (ch_id + 1) % N_OUTPUT_CHANNELS;
        if (ch_id >= N_OUTPUT_CHANNELS-1)  task_id++;
        if(task_id >= nports){
                task_id = 0;
                dest_coord++;
                if(dest_coord >= nboards) dest_coord=1;
        }
}
}
```

Listing 5.3: Imagifier HLS Kernel implementation

Starting from the I/O and Preprocessing node, the "imagifier" HLS kernel is reported in Listing 5.3. From interfaces defined with Vitis pragmas (in particular "**`#pragma HLS interface ap_ctrl_none port=return`**"), we can notice that this is defined as a free-running kernel: a kernel which starts with the bitstream loading on the device, without any call by the CPU host (which is required by "sender" and "receiver", instead).

The "imagifier" works on packets of data coming from the network with the HAPECOM communication protocol, each of them corresponding to a single physics event. To increase the overall design throughput, and so to work in streaming mode, we decided to not use the HAPECOM receive() API by reading directly data from input channels with the Vitis read() function. However, in this way, to have the packet size information, we must access to a certain bit address of the HAPECOM header bounded by `size_start_bitpos` and `size_end_bitpos` variables. After that, we proceed with the reception of the single event header, which has the information relative to the number of words composing the event and the event timestamp, and then we work on each event word to obtain the PMT hitlist and to convert it to a 16x16 image. As last step of the preprocessing, this image (and the event timestamp) is sent via HAPECOM send() API to one of the Computing nodes in a "round robin" way, choosing each task of each node as possible destination.

```
Extern "C" void cnn_kernel(message_stream_t message_data_in[N_INPUT_CHANNELS],
message_stream_t message_data_out[N_OUTPUT_CHANNELS])
{
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=message_data_in
#pragma HLS interface axis port=message_data_out
#pragma HLS dataflow

    hls::stream<input_t> nnet_input;
    hls::stream<result_t> nnet_output;
    hls::stream<ap_axis<128,0,0,0>> stream_timestamp;
    read_from(message_data_in, stream_in, stream_timestamp);
    hwfunc(nnet_input, nnet_output);
    get_class(nnet_output, message_data_out, stream_timestamp);
}
```

Listing 5.4: “cnn_kernel" HLS code

As the main component of Computing nodes of the setup, the HLS code of “cnn_kernel” is reported in Listing 5.4. From interfaces, we can notice that this is defined as a free-running kernel (as for “imagifier” one) and it is composed by different task functions pipelined via HLS dataflow Vitis pragma. This allows functions to overlap in their operation, increasing the overall throughput of design by increasing concurrency of the RTL tasks implementation. In detail:

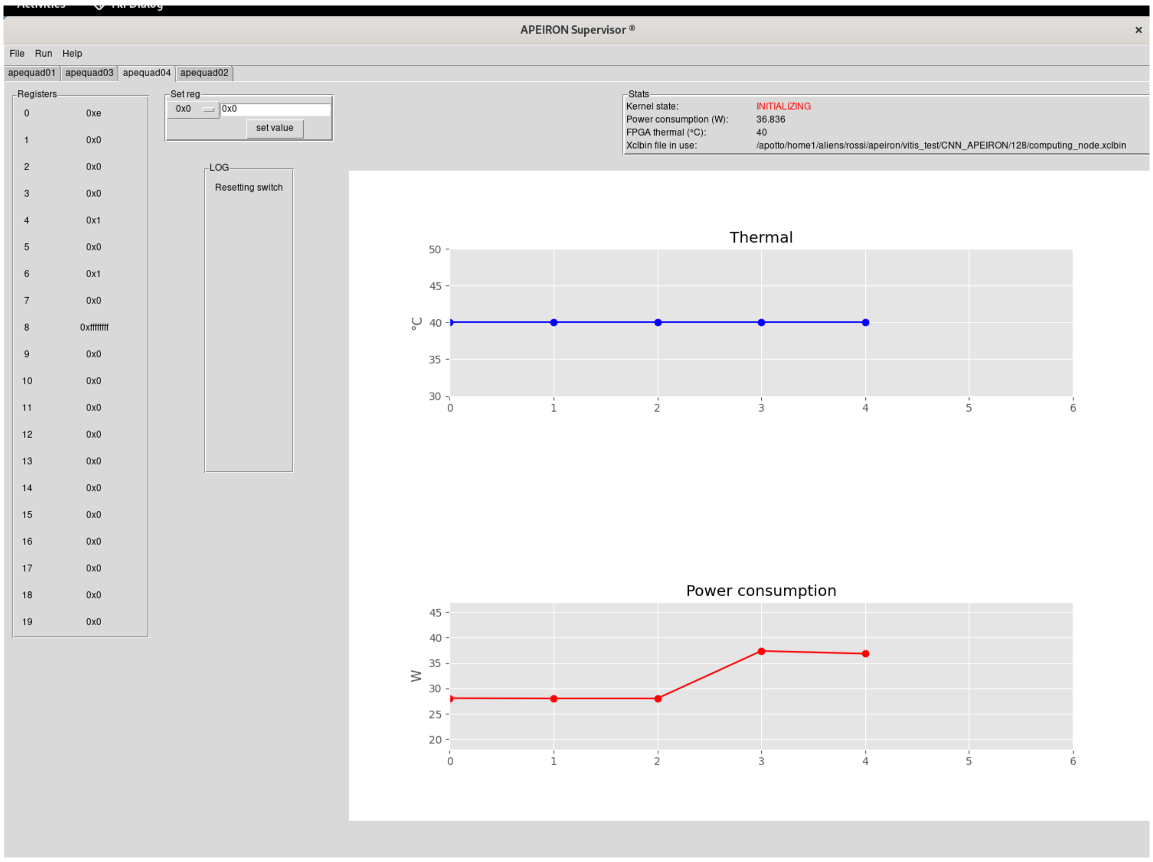
- “read_from” receives packets from the networks, obtaining information to be streamed as CNN input (256 bit image) or to be streamed to label the event processed (event timestamp)
- “hwfunc” is the task in which the FPGA implemented CNN (obtained from the HLS4ML framework) processes streaming input images
- “get_class” receives CNN output and obtains the predicted ring class. This is then sent through the network via the HAPECOM `send()` function.

Considering the requirement of a multiple-FPGAs execution platform for this application, we decided to use RAIDER setup to stress out and to validate the correct behavior of the APEIRON host software stack. In order to work on 4 boards, we need to start on each of nodes an “apeirond” daemon able to receive requests from the client on which we execute the supervisor (this can be any of the network nodes). To connect the “supervisor” to each of the daemons, we start the “supervisor” by passing from command line the hostnames of each execution node:

```
$ ./supervisor.py -H apequad01-1, apequad02-1, apequad03-0, apequad04-0
```

Listing 5.5: “supervisor" Python script command line example

As can be seen from Listing 5.5, it is possible to specify for each host the device id on which we want to load the bitstream for the application, this is useful in case there are more the one Alveo board connected on the PCIe bus (in particular, we worked on device with ID 1 on apequad01/02 and on device with ID 0 on apequad03/04).



```
Registers
0     0xe
1     0x0
2     0x0
3     0x0
4     0x1
5     0x0
6     0x1
7     0x0
8     0xffffffff
9     0x0
10    0x0
11    0x0
12    0x0
13    0x0
14    0x0
15    0x0
16    0x0
17    0x0
18    0x0
19    0x0
```

```
LOG
Resetting switch
```

```
Stats
Kernel state:               INITIALIZING
Power consumption (W):      36.836
FPGA thermal (°C):          40
Xclbin file in use:         /apotto/home1/aliens/rossi/apeiron/vitis_test/CNN_APEIRON/128/computing_node.xclbin
```

Figure 5.3: Monitoring execution on one computing node with the APEIRON Supervisor display

The Python script execution opens a GUI as the one displayed in Figure 5.3. Selecting the tab corresponding to the desired node, it is possible to open the "Run" drop-down box and select the "Run" command to load the specified host application, compiled as dynamic library (.so file). Two different host applications have been used respectively for the Preprocessing and the Computing nodes:

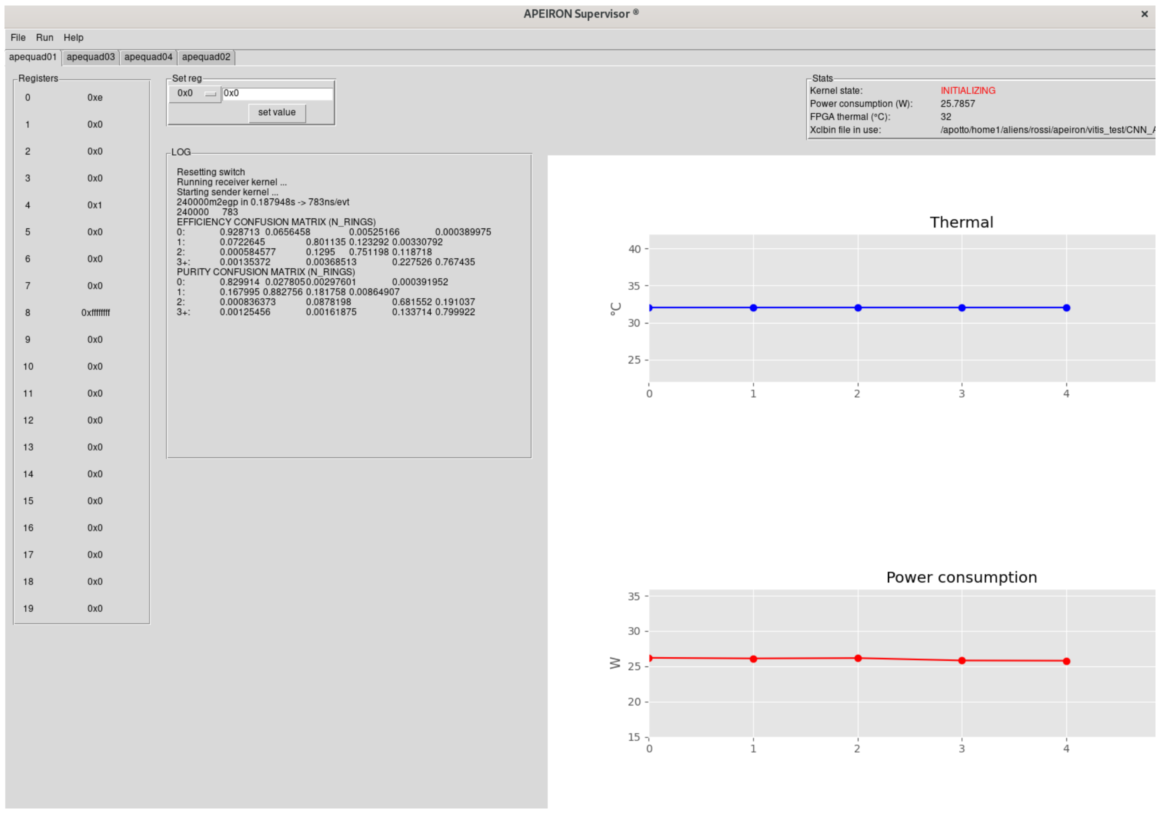
```
// **** PREPROCESSING NODE **** //

extern "C" void run_kernel (apeiron::Handler *fpga_handler)
{
      unsigned npackets = NPACKETS;
      unsigned xdest = XDEST;
      unsigned local_coord = LOCAL_COORD;


      fpga_handler->device_open(get_device_id());
      fpga_handler->device_dump();
      fpga_handler->xclbin_load("preprocessing_node.xclbin");

      fpga_handler->set_state_initializing();

      xrt::kernel kreceiver;
      xrt::kernel ksender;
      xrt::kernel kimage_sender;

      kreceiver = xrt::kernel(fpga_handler->get_device(),fpga_handler->get_uuid(),
"krnl_receiver:{krnl_receiver_1}");
      ksender = xrt::kernel(fpga_handler->get_device(),fpga_handler->get_uuid(),
"krnl_sender:{krnl_sender_1}");
```

```
      kimage_sender = xrt::kernel(fpga_handler->get_device(),fpga_handler->get_uuid(),
"image_sender:{image_sender_1}");

std::vector<stream_data_t> m2egp;
unsigned m2egp_count = 0;
m2egp_count += read_m2egp_file("clop_flow/dataset.dat", m2egp, npackets);
if (!m2egp_count) exit(EXIT_FAILURE);

size_t outbuf_size = m2egp_count * sizeof(stream_data_t);
size_t inbuf_size = m2egp.size() * sizeof(m2egp[0]);

xrt::bo recv_buffer(fpga_handler->get_device(), outbuf_size, kreceiver.group_id(0));
stream_data_t *recv_buffer_map = recv_buffer.map<stream_data_t*>();
memset(recv_buffer_map, 0, outbuf_size);

xrt::bo send_buffer(fpga_handler->get_device(), inbuf_size, ksender.group_id(2));
stream_data_t *send_buffer_map = send_buffer.map<stream_data_t*>();
memset(send_buffer_map, 0, inbuf_size);

for (unsigned i=0; i < m2egp.size(); ++i) {
           send_buffer_map[i].high = m2egp[i].high;
           send_buffer_map[i].low = m2egp[i].low;
       }
apeiron::cout<<"LOAD DATA ON FPGA\n";
send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);

apeiron::cout<<"Resetting switch\n";
fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),4, 0x1);
       // auto-toggle reset
sleep(1);
       fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),6,
local_coord); // 3D coordinate
fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),69, 0x01800060); //
threshold
       fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),70,
0x0000ff40); // new credit cycle


apeiron::cout<<"Running receiver kernel ...\n";
xrt::run kreceiver_run = kreceiver(recv_buffer, m2egp_count);
apeiron::cout<<"Starting sender kernel ...\n";
xrt::run kimage_sender_run = kimage_sender(NPORTS,NDEVICES);
xrt::run ksender_run = ksender(npackets, send_buffer);
auto tstart = std::chrono::high_resolution_clock::now();
apeiron::cout<<"Waiting for sender kernel to complete ...\n";
       fpga_handler->set_state_waiting();
ksender_run.wait();
apeiron::cout<<"Waiting for receiver kernel to complete ...\n";
kreceiver_run.wait();
auto tend = std::chrono::high_resolution_clock::now();
recv_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
auto deltat = tend - tstart;

float *rings;
float *evts;
rings = (float*)calloc(20,sizeof(float));
evts = (float*)calloc(5,sizeof(float));
for (unsigned i=0; i < m2egp_count; ++i) {
auto ring_RECO= (recv_buffer_map[i].high >> 48) & 0xFF;
if(ring_RECO>=3) ring_RECO=3;
evts[ring_RECO]++;
auto ring_NN= recv_buffer_map[i].low & 0xFF;
rings[4*ring_RECO+ring_NN]++;
}

std::stringstream ss;
ss << m2egp_count << "m2egp in " << deltat.count()/(1e9) << "s -> " <<
deltat.count()/(m2egp_count)<< "ns/evt\n";
ss << m2egp_count << " \t "<< deltat.count()/(m2egp_count ) << "\n";
apeiron::cout << ss.str();

std::stringstream ss1;
ss1 <<"EFFICIENCY CONFUSION MATRIX (N_RINGS) \n";
ss1 << efficiency_matrix(rings,evts) <<"\n";
```

```
ss1 <<"PURITY CONFUSION MATRIX (N_RINGS)\n";
ss1 << purity_matrix(rings) << "\n";
apeiron::cout << ss1.str();

std::printf("----- APEIRON LIB DUMP -------\n");
std::printf(apeiron::cout.circular_buffer_dump().c_str());
std::printf("------------\n");

}
```

```
// **** COMPUTING NODE **** //

extern "C" void run_kernel (apeiron::Handler *fpga_handler)
{
  unsigned local_coord = LOCAL_COORD;


  fpga_handler->device_open(fpga_handler->get_device_id());
  fpga_handler->device_dump();
  fpga_handler->xclbin_load("computing_node.xclbin");


  fpga_handler->set_state_initializing();

  apeiron::cout<<"Resetting switch\n";
  fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),4, 0x1); // auto-
toggle reset
   sleep(1);
   fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),6, local_coord);
// 3D coordinate
   fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),69, 0x01800060);
// threshold
    fpga_handler->write_register(fpga_handler->get_ip("TextaRossa_switch"),70, 0x0000ff40);
// new credit cycle


   fpga_handler->set_state_waiting();


   std::printf("----- APEIRON LIB DUMP -------\n");
   std::printf(apeiron::cout.circular_buffer_dump().c_str());
   std::printf("------------\n");

}
```

Listing 5.6: Host code for Preprocessing and Computing nodes



Registers

0	0xe
1	0x0
2	0x0
3	0x0
4	0x1
5	0x0
6	0x0
7	0x0
8	0xffffffff
9	0x0
10	0x0
11	0x0
12	0x0
13	0x0
14	0x0
15	0x0
16	0x0
17	0x0
18	0x0
19	0x0

LOG

```
Resetting switch
Running receiver kernel ...
Starting sender kernel ...
240000m2egp in 0.187948s -> 783ns/evt
240000    783
EFFICIENCY CONFUSION MATRIX (N_RINGS)
0:        0.928713  0.0656458          0.00525166          0.000389975
1:        0.0722645           0.801135 0.123292 0.00330792
2:        0.000584577         0.1295   0.751198 0.118718
3+:       0.00135372          0.00368513          0.227526 0.767435
PURITY CONFUSION MATRIX (N_RINGS)
0:        0.829914  0.02780500.00297601          0.000391952
1:        0.167995 0.882756 0.181758 0.00864907
2:        0.000836373         0.0878198           0.681552 0.191037
3+:       0.00125456          0.00161875          0.133714 0.799922
```

Stats

Kernel state:	INITIALIZING
Power consumption (W):	25.7857
FPGA thermal (°C):	32
Xclbin file in use:	/apotto/home1/aliens/rossi/apeiron/vitis_test/CNN_

Figure 5.4: Monitoring execution on Preprocessing node using the APEIRON Supervisor display

We have scaled the system from 2 nodes (one I/O and preprocessing and one computing) up to 4 increasing the number of computing nodes as shown in Figure 5.2 and measured the processing time per event and the integrated processing throughput of the system; results are tabulated for the former and plotted for the latter in Figure 5.5 (throughput is in millions of events per second, MHz in figure).

# nodes	# CNNs	Processing time per event (us)
2 nodes	1 CNN	3.440
	2 CNNs	1.720
3 nodes	2CNNs	1.720
	4CNNs	0.860
4 nodes	3CNNs	1.147
	6CNNs	0.731

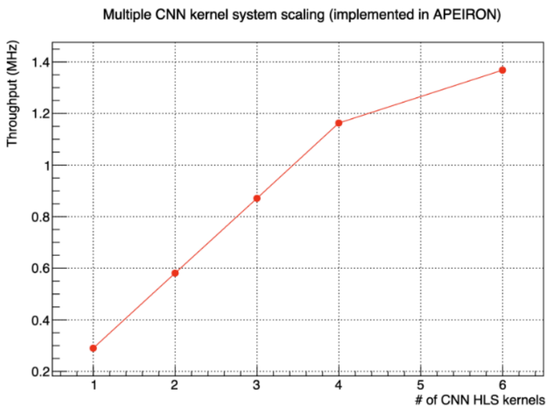


Figure 5.5: Scaling of processing time per event (left) and processing throughput (right) with the number of deployed CNN kernels distributed over 1 to 3 computing nodes

The presented results show the good scaling of system performance with the number of nodes. The flattening slope of the curve when the number of CNNs goes beyond 4 is mainly due to the saturation of the data injection rate in the krnl_sender on the preprocessing node.

6. Conclusion

We presented the three main components of the APEIRON framework: the host software stack, the HAPECOM lightweight C++ HLS communication library and the tool for the automatic linking of computational tasks and Communication IP for FPGA bitstream generation.

The APEIRON framework in its entirety allowed us to overcome the limits imposed by the original Xilinx XRT suite, such as the limited management of multiple processes that need to use the same hardware resources concurrently

We then used a real application developed at INFN (RAIDER) to validate the software stack, from the upper level “supervisor” component, used to deploy the application to four nodes equipped with multiple FGPAs down to the hardware description of the kernels. This demonstrates how it can be possible to set up the software environment to reach the expected overall system design. The co-design of APEIRON software stack along with its Communication IP allowed reaching very low and deterministic latency and a high fraction of the channel raw bandwidth for communications between FPGAs, addressing two fundamental bottlenecks for real-time distributed streaming applications at the same time, while allowing for a straightforward development and deployment of multi-FPGA HLS designs.

References

[1] https://xilinx.github.io/XRT/master/html/index.html

[2] https://docs.python.org/3/library/tkinter.html

[3] R. Ammendola, A. Biagioni, O. Frezza, A. Lonardo, F.L. Cicero, P.S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, P. Vicini, Journal of Instrumentation 8, C12022 (2013)

[4] E. Cortina Gil et al, *The beam and detector of the NA62 experiment at CERN*, 2017 *JINST* **12** P05025 **DOI** 10.1088/1748-0221/12/05/P05025

[5] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran et al., Journal of Instrumentation 13, P07027 (2018)