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

WP2 New accelerator designs exploiting mixed precision

D2.8 IP for low-latency internode communication links, part 1

Revised version

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: D2.8 IP for low-latency internode communication links, part 1

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	D2.8
WP No:	WP2
WP Leader:	CINI-UNIPI
Due date:	M18 (September 30, 2022)
Delivery date:	20/05/2023 (revised)

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

WP 2 New accelerator designs exploiting mixed precision

Deliverable number:	D2.8					
Deliverable title:	IP for low-latency inter-node communication links, part 1					
Due date:	M18 (30 Sept. 2022)					
Actual submission date:	20/05/2023 (revised)					
Editor:	Francesca Lo Cicero					
Authors:	F. Lo Cicero, A. Lonardo, C. Rossi, M. Martinelli, F. Simula					
Work package:	WP2					
Dissemination Level:	Public					
No. pages:	38					
Authorized (date):	15/05/2023 (revised)					
Responsible person:	Francesca Lo Cicero					
Status:	Plan	Draft	Working	Final	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	2022-10-04	F. Lo Cicero	Draft structure
0.2	2022-10-07	A. Lonardo	Added several contributions, reviewed the document.
0.3	2022-10-08	M. Martinelli	Added Bandwidth test section
0.4	2022-10-09	C. Rossi	Added Latency test section
0.5	2022-10-10	F. Lo Cicero	First Release
0.6	2022-10-12	C. Rossi, F. Lo Cicero	Changes due to Kulczewski's comments

0.7	2022-10-18	F. Simula	Added ToC, fixed references to captions
1.1	2023-04-23	F. Lo Cicero, A. Lonardo, C. Rossi	Updated to address reviewers' observations (extended Introduction, added Appendixes B and C, general review).

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Michal Kulczewski	
Checked by Task Leader	Francesca Lo Cicero	24/10/2022
Checked by WP Leader	Sergio Saponara	26/04/2023
Checked by Project Coordinator	Massimo Celino	15/05/2023

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

List of Acronyms

Acronym	Definition
IP	Intellectual Property
FPGA	Field Programmable Gate Array
HLS	High-Level Synthesis
VCT	Virtual Cut-Through
DOR	Dimension Order Routing
FSM	Finite State Machine
API	Application Programming Interface
BRAM	Block Random Access Memory
DDR (SDRAM)	Double Data Rate (Synchronous Dynamic Random Access Memory)

Executive Summary

This document reports on the activities done by TEXTAROSSA partner INFN with reference to the design of the internode communication IP in WP2.

The INFN Communication IP, developed in VHDL, allows data transfers between processing tasks hosted in the same node (intra-node communications) or in different nodes (inter-node communications), implementing a direct network for FPGA accelerators and enabling the distributed implementation of dataflow applications in the APEIRON framework.

The Communication IP was tested in the Vivado design Suite and its AXI-Lite interface (used to read/write internal registers) and was verified using the Xilinx AXI Verification IP (VIP).

After behavioural simulation, it was implemented as an RTL kernel in Xilinx Vitis and integrated with kernels written in HLS in the APEIRON framework.

The synthesis results, both for U200 and U280 Alveo card, showed a low resources occupancy of the IP, allowing us to add new features in the future. For example, we will increase internal datapath and fifo depth (thus increasing the intranode communication bandwidth and avoiding loss of performance due to fifo filling), and the number of intraNode ports. For the InterNode communication, we foresee to improve the bandwidth increasing the number of channels' lanes and implementing new channel interface.

We also performed tests on two U200 cards connected by QSFP+ cable and on a U280 card (using a channel termination) to measure the performance, in terms of latency and bandwidth, of the Communication IP.

The IP project database synthesizable both on the Alveo U200 and U280 platforms is publicly available on the deliverable section of the TEXTAROSSA project website (https://textarossa.eu/dissemination/deliverables/).

1 Introduction

The INFN Communication IP implements a direct network for FPGA accelerators, allowing low-latency data transfer between processing tasks deployed on the same FPGA (intra-node communication) and on different FPGAs (inter-node communication), and enabling the distributed implementation of dataflow applications in the APEIRON framework.

This document describes the Communication IP in detail and shows preliminary data for its synthesis on the two reference platforms (Xilinx U200 and U280), along with results of tests developed to validate the design and assess its current performance.

Section 2 shows the design architecture complementing the information already available in Deliverable 2.1 – Consolidated specs of accelerators IPs.

Section 3 introduces the APEIRON framework, defined as the general architecture of an FPGA-based distributed stream processing platform and the corresponding software stack. The Communication IP was co-designed with the APEIRON software stack in order to achieve very low-latency and scalable bandwidth (via IP design reconfiguration) between processing tasks defined as High-Level Synthesis Kernels.

Section 4 highlights the implementation results in terms of FPGA resource usage (for both Alveo U280 and U200) of the Communication IP and of the HLS kernels of the testbench.

Section 5 reports design validation test results and performance measurement for the intermediate release of the Communication IP.

Section 6 sketches some conclusions about the work and the results presented in this document, indicating the foreseen activities regarding the development of the Communication IP for the remaining part of the project.

Appendix A reports the pseudo-code for the performance tests used to collect results showed in Section 5.

Appendix B provides a simple instruction manual to assist users in integrating the Communication IP in their designs using the Vitis environment.

Finally, Appendix C illustrates the usage of the APEIRON framework, using as example the design of the testbench described in Section 5.2.2, that integrates the Communication IP and two instantiations of an HLS kernels, in a single FPGA configuration.

This document, along with the Communication IP packaged as Xilinx object (XO) file for both the U200 and U280 platform, and a demo video showing the performance tests described in section 5.2, is publicly available for download on the deliverable section of the TEXTAROSSA web site (https://textarossa.eu/dissemination/deliverables/).

2 IP Design

The Communication IP allows data transfers between processing tasks hosted in the same node (intra-node communications) or in different nodes (inter-node communications), see Figure 2.1. In the context of the APEIRON framework, processing tasks are implemented by HLS kernels with Xilinx Vitis. The details of the interface between HLS kernels – the endpoints of the communication – and the Communication IP are described in Section 3.

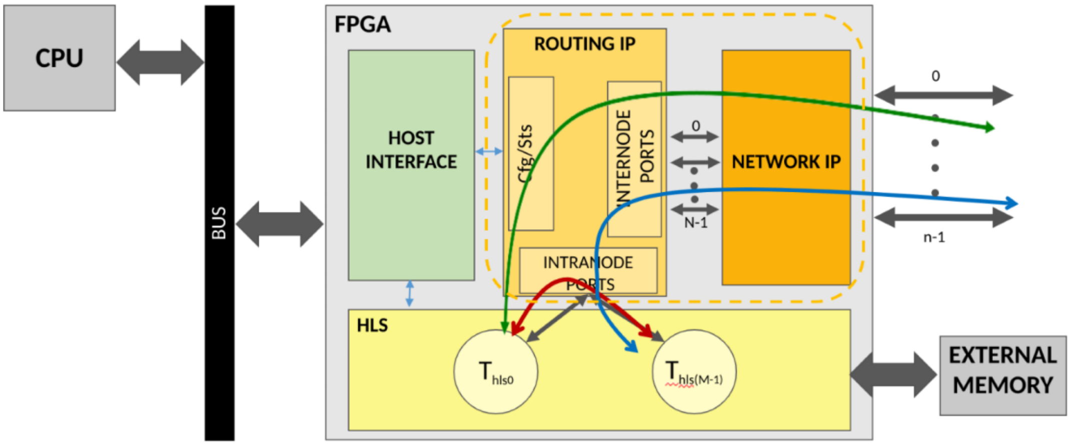

Figure 2.1 Example of intra-node (in red) and inter-node (in blue/green) data transfers between tasks

Figure 2.2 shows its hardware block structure, which contains a **Network_IP** and a **Routing_IP**, both developed in VHDL for TEXTAROSSA target platforms (Xilinx Alveo U200 and U280 cards).

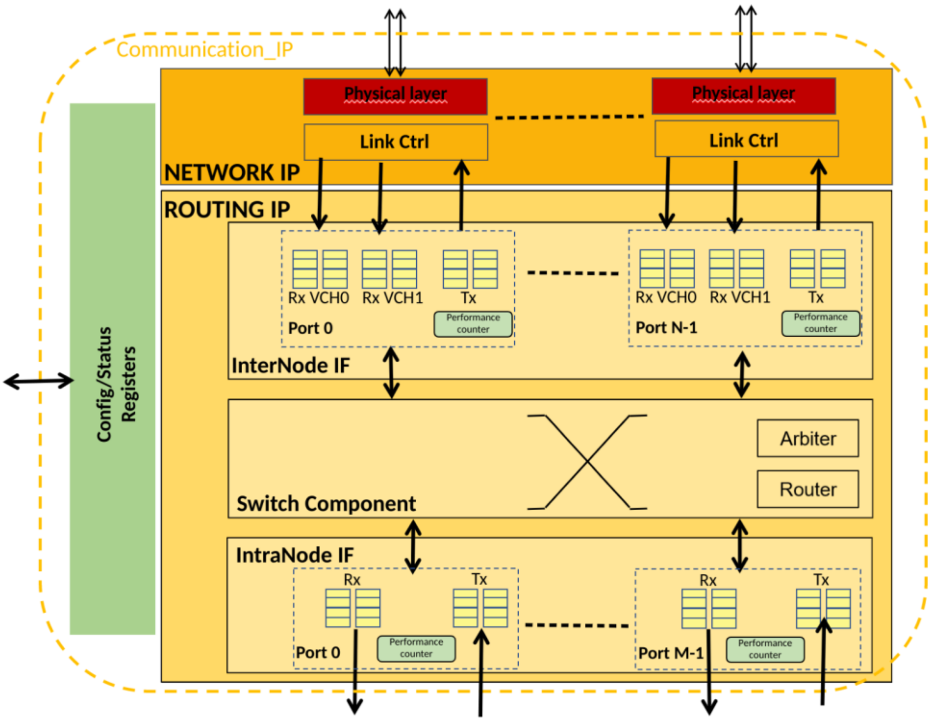

Figure 2.2 Architectural partition of Communication IP

The **Routing_IP** defines the switching technique and routing algorithm; its main components are the Switch_component Block, the Configuration/Status Registers and the InterNode and IntraNode IFs.

The Switch component dynamically interconnects all ports of the IP, implementing a channel between source and destination ports.

Dynamic links are managed by routing logic together with arbitration logic: the Router configures the proper path across the switch while the Arbiter is in charge of solving contentions between packets requiring the same port.

For inter-node communications, the routing policy applied is the dimension-order one: it consists in reducing the offset along one dimension to zero before considering the offset in the next dimension.
The employed switching technique — i.e., when and how messages are transferred — is Virtual Cut-Through (VCT) [1]: the router starts forwarding the packet as soon as the algorithm has picked a direction and the buffer used to store the packet has enough space. The deadlock-avoidance of DOR routing is guaranteed by the implementation of two virtual channels for each physical channel (with no fault-tolerance guaranteed) [2].

The transmission is packet-based, meaning that the Communication IP sends, receives and routes packets with a header (Figure 2.3), a variable size payload and a footer.

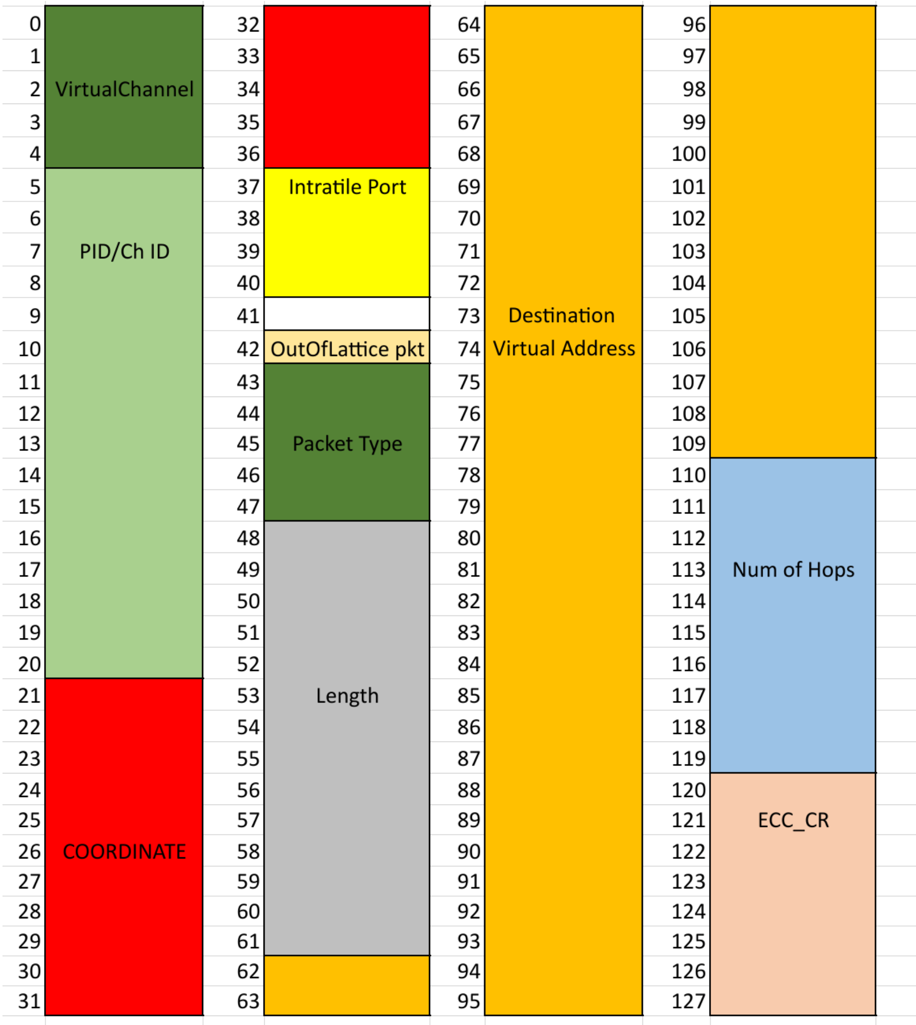

Figure 2.3 Packet header format

In the **Network IP**, the physical layer blocks define the data encoding scheme for the serialization of the messages over the cable and shape the network topology. They provide point-to-point bidirectional, full-duplex communication channels of each node with its neighbors along the available directions.

For the serialization of the messages over the cable we used Xilinx Aurora 64B/66B cores.

The number of lanes making up a communication channel can be customized at design time (from 1 to 2) to match the requirements of the integrated target execution platform.

Link_Ctrl blocks instead establish the logical link between nodes and guarantee reliable communication, eventually performing error detection and correction.

The whole architecture is based on a layer model, as shown in Figure 2.4, including physical, data link, network and transport layers of the OSI model.

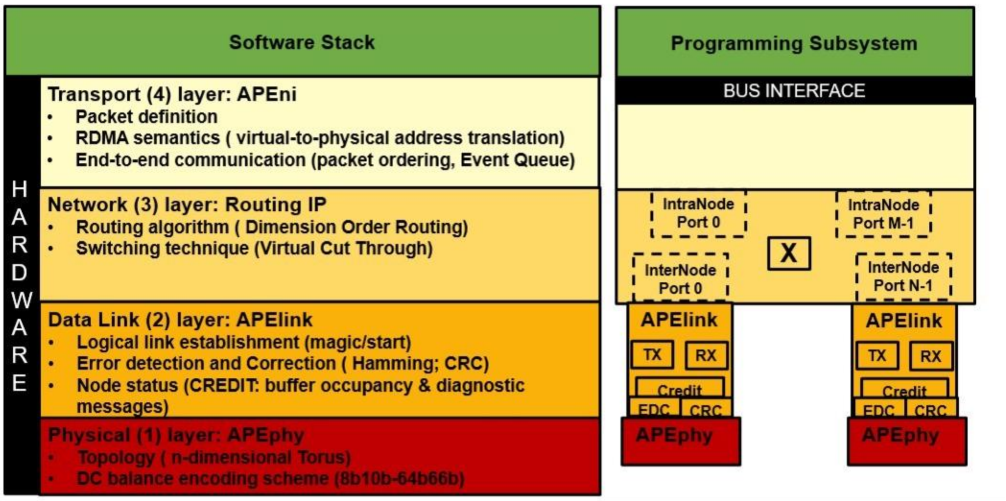

Figure 2.4 The layered architecture of Communication IP

The Communication IP exposes two sets of interfaces, i.e., IntraNode and InterNode IF; the number of ports within these interfaces (M and N) can be customized at design time.
The IntraNode IF manages data flow to (RX) and from (TX) local tasks; each port consists of two FIFOs for each direction, so that header/footer and data use a dedicated FIFO.
The InterNode IF, with the Network_IP block, oversees managing data flow over the serial links between FPGAs.
Both IntraNode and InterNode IFs are provided with a self-test mechanism to measure the latency and bandwidth achieved. The self-test mechanism is composed by three simple IPs: (i) the Packet_Generator generates packets and fills the transmitting FIFOs; (ii) the Consumer flushes the receiving FIFOs avoiding their overflow and checks payload of received packets; (iii) the Performance Counter stores the clock cycles needed to complete the data transfers.

The traffic generated by the Packet_Generator can be configured at runtime writing appropriate registers which define the number of packets, size and destination along X coordinate.
In addition to these parameters, the Communication IP offers to the users the possibility to set at run time few key features and to read status information by exposing a window of 32-bit registers (Configuration/Status Registers Block).

The list of registers with the associated addresses are presented in Table 2.1.:

N	Offset	Name	Description	Default value	Register type
4	0x00000010	RESET_REG	Bit 0: write ‘1’ to reset; Self-clear (‘0’ after 200 clock’s cycle)	0x0000000	RW
5	0x00000014	REVISION_REG	Bit 15 downto 0: Revision ID Bit 31 downto 16: Version ID	0x0000000	RO
6	0x00000018	COORDME_REG	3D node’s coordinates Bit 5 downto 0: X coordinate	0x0000000	RW
8	0x00000020	LATTICESIZE_REG	Lattice size Bit 5 downto 0: X direction	0xfffffff	RW

12	0x00000030	PERF_INTRANODE_CFG	Perf_Block configuration register: Bit 7 downto 0: IntraNode Packet_Generator enable Bit 15 downto 8: IntraNode Consumer enable	0x0000000	RW
13	0x00000034	PERF_INTERNODE_CFG	Perf_Block configuration register: Bit 7 downto 0: InterNode Packet_Generator enable Bit 15 downto 8: InterNode Consumer enable	0x00000000	RW
14	0x00000038	PKTGEN_CONFIG_0	Packet_generator configuration register: Bit 15 downto 0: number of packets generated Bit 29 downto 16: packet length (in byte) Bit 31: header only packet generated	0x00000000	RW
16	0x00000040	PKTGEN_CONFIG_1	Destination of packet Bit 5 downto 0: X direction	0x00000000	RW
20	0x00000050	PERF_INTRANODE_STS	Bit 3 downto 0: packet_generator status (intraNode 0) Bit 7 downto 4: Consumer status (intraNode 0) Bit 11 downto 8: packet_generator status (intraNode 1) Bit 15 downto 12: Consumer status (intraNode 1) Bit 19 downto 16: packet_generator status (intraNode 2) Bit 23 downto 20: Consumer status (intraNode 2) Bit 27 downto 24: packet_generator status (intraNode 3) Bit 31 downto 28: Consumer status (intraNode 3) *Packet_generator status: "0000" SM_STATE = OFF "0001" SM_STATE = IDLE "0010" SM_STATE = TX_HEADER "0011" SM_STATE = TX_PAYLOAD "0100" SM_STATE = TX_FOOTER **Consumer status Bit 0 = Test ok! (All packets received with correct payload) Bit 3 downto 1: SM STATE "000" SM_STATE = OFF "001" SM_STATE = IDLE "010" SM_STATE = COUNT		RO
21	0x00000054	PERF_INTERNODE_STS	Bit 3 downto 0: packet_generator status (see register PERF_INTRANODE_STS) Link 0 Bit 7 downto 4: Consumer status Link 0 Bit 11 downto 8: packet_generator status Link 1 Bit 15 downto 12: Consumer status Link 1		RO
22	0x00000058	PERF_INTRANODE_CNT0	TxRx clock counter (first packet written, last packet read) IntraNode 0		RO
23	0x0000005C	PERF_INTRANODE_CNT1	TxRx clock counter (first packet written, last packet read) IntraNode 1		RO
24	0x00000060	PERF_INTRANODE_CNT2	TxRx clock counter (first packet written, last packet read) IntraNode 2		RO
25	0x00000064	PERF_INTRANODE_CNT3	TxRx clock counter (first packet written, last packet read) IntraNode 3		RO
26	0x00000068	PERF_INTERNODE_CNT0	TxRx clock counter (first packet written, last packet read) InterNode 0		RO

27	0x0000006C	PERF_INTERNODE_CNT1	TxRx clock counter (first packet written, last packet read) InterNode 1		RO
28	0x00000070	INTRANODE_FIFO_STS_RX_0	Bit 31 downto 16: Fifo IntraNode 0 Data Rx UsedWord Bit 15 downto 0: Fifo IntraNode 0 Header Rx UsedWord		RO
29	0x00000074	INTRANODE_FIFO_STS_TX_0	Bit 31 downto 16: Fifo IntraNode 0 Data Tx UsedWord Bit 15 downto 0: Fifo IntraNode 0 Header Tx UsedWord		RO
30	0x00000078	INTRANODE_FIFO_CNT_HD_TX_RD_0	Fifo IntraNode 0 Header Tx read counter		RO
31	0x0000007C	INTRANODE_FIFO_CNT_HD_TX_WR_0	Fifo IntraNode 0 Header Tx write counter		RO
32	0x00000080	INTRANODE_FIFO_CNT_HD_RX_RD_0	Fifo IntraNode 0 Header Rx read counter		RO
33	0x00000084	INTRANODE_FIFO_CNT_HD_RX_WR_0	Fifo IntraNode 0 Header Rx write counter		RO
34	0x00000088	INTRANODE_FIFO_CNT_DT_TX_RD_0	IntraNode 0 Data Tx read counter		RO
35	0x0000008c	INTRANODE_FIFO_CNT_DT_TX_WR_0	IntraNode 0 Data Tx write counter		RO
36	0x00000090	INTRANODE_FIFO_CNT_DT_RX_RD_0	IntraNode 0 Data Rx read counter		RO
37	0x00000094	INTRANODE_FIFO_CNT_DT_RX_WR_0	IntraNode 0 Data Rx write counter		RO
38 47	0x00000098- 0x000000BC	INTRANODE_FIFO_*_1	Fifo counter register IntraNode 1		RO
65		FIFO_INTRANODE_EXC	Bit 7 downto 0 = IntraNode TX HD write exception Bit 15 downto 8 = IntraNode TX DT write exception Bit 23 downto 16 = IntraNode RX HD write exception Bit 31 downto 24 = IntraNode RX DT write exception		RO
66	0x00000108	FIFO_REGISTER	Bit 31 downto 24: Fifo Header Rx exp width Bit 23 downto 16: Fifo Data Rx exp width Bit 15 downto 8: Fifo Header Tx exp width Bit 7 downto 0: Fifo Data Tx exp width		RO
67	0x0000010C	TRANSCEIVER_STATUS	Bit 0: InterNode 0 channel up Bit 1: InterNode 1 channel up Bit 16: InterNode 0 transceiver's error Bit 17: InterNode 1 transceiver's error		RO
68	0x00000110	LINK_0_CONFIG_0	Bit 31 downto 28: Edac enable InterNode 1 "0000" NO EDAC "1111" EDAC Bit 27 downto 24 = Edac enable InterNode 0 Bit 17 = Use new destination in InterNode 1 Bit 16 = Use new destination in InterNode 0 Bit 15 downto 0: New destination (15-11: Z; 10-6: Y; 5-0:X).	0x00000000	RW
69	0x00000114	LINK_0_CONFIG_1	Bit 25 downto 16: Red threshold for Data Bit 7 downto 0: Red threshold for Header	0x00000000	RW
70	0x00000118	LINK_0_CONFIG_2	Bit 15 downto 8: Tx new credit cycle Bit 7 downto 0: Tx waiting cycle	0x00000000	RW
71	0x0000011C	LINK_0_CONFIG_3	Header error gen	0x00000000	RO

80	0x00000140	LINK_0_STATUS_0	Bit 15 downto 12: Rx status; Bit 11 downto 8: Tx footer status Bit 7 downto 4: Tx payload status Bit 3 downto 0: Tx header status		RO
81	0x00000144	LINK_0_ERROR	Bit 31 downto 16: Rx header error counter Bit 15 downto 0: Rx header fatal error counter		RO
82	0x00000148	LINK_0_TX_MAGIC	Tx magic counter		RO
83	0x0000014C	LINK_0_TX_START	Tx start counter		RO
84	0x00000150	LINK_0_TX_HDR	Tx header counter		RO
85	0x00000154	LINK_0_TX_FTR	Tx footer counter		RO
86	0x00000158	LINK_0_RX_MAGIC	Rx magic counter		RO
87	0x0000015c	LINK_0_RX_START	Rx start counter		RO
88	0x00000160	LINK_0_RX_HEADER	Rx header counter		RO
89	0x00000164	LINK_0_RX_FOOTER	Rx footer counter		RO
90 99	0x00000168- 0x00000185	LINK_1_REGISTERS			

Table 2.1 Configuration/status Registers list

2.1 Communication IP simulation

The Communication IP was verified in the Vivado design Suite by using the Packet generator Block and the Consumer Block (which respectively generate packets filling IntraNode_0 transmitting FIFO and flush the receiving FIFOs checking payload of received packets).
To simulate registers been read or written we implemented a Xilinx IP AXI traffic generator, which provides an AXI4-Lite Master interface and issues AXI4-Lite transactions reading two coefficient (COE) files provided by the user:

- Address COE File – Provides the sequence of addresses to be issued
- Data COE File – Provides the sequence of data corresponding to the address specified in Address COE File

We also checked protocol compliance of AXI interfaces using the AXI Verification IP (VIP).

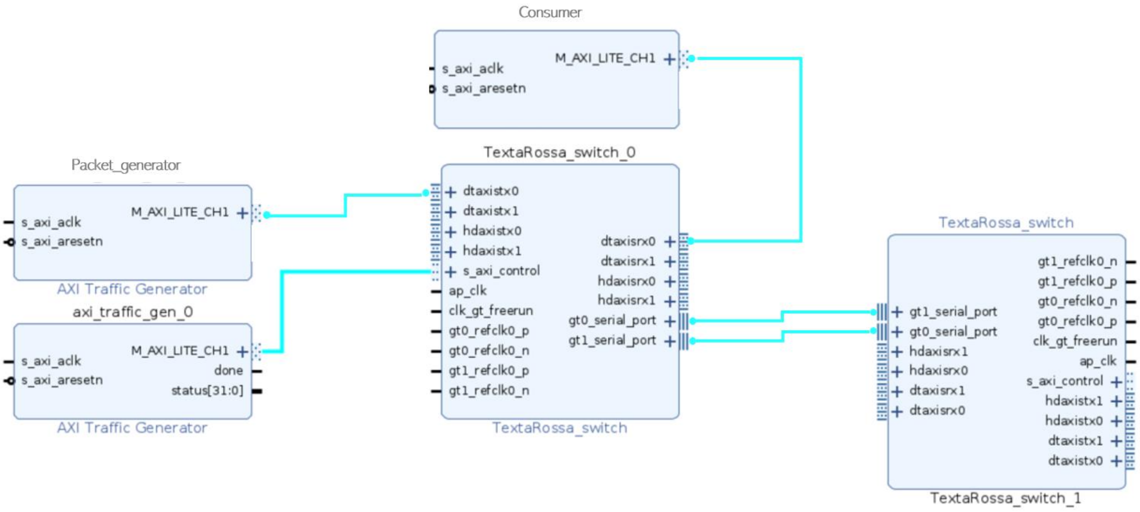

Figure 2.5 Scheme of Communication IP testbench

The latency introduced by the Routing_IP — i.e., from the footer's writing in intraNode TX FIFO to the footer's reading in the intraNode RX FIFO in a loopback communication — is shown in Figure 2.6.

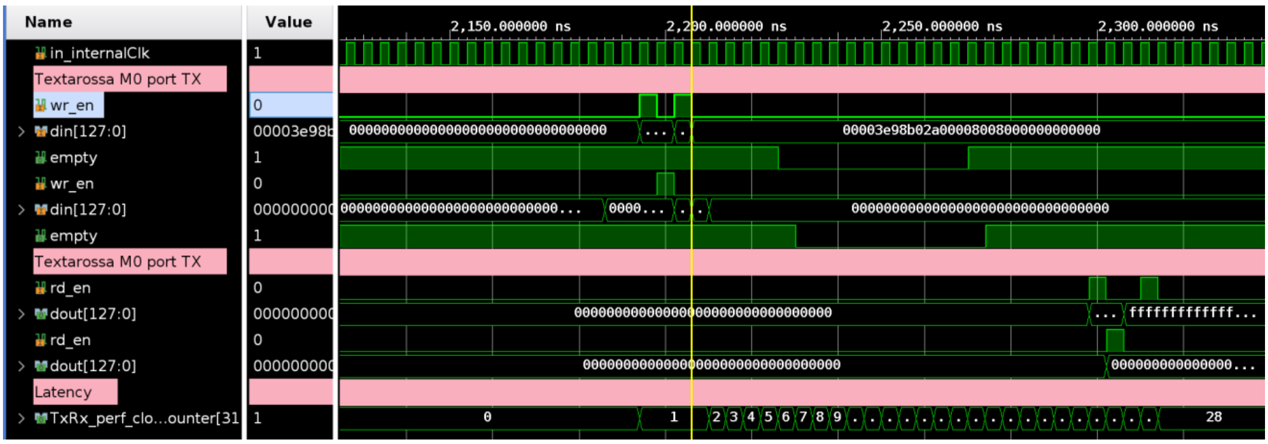

Figure 2.6 Intra-node TX FIFO – RX FIFO latency in Vivado Behavioural simulation GUI

The number of clock cycles is equal to 28 (about 280ns at the current operating frequency of 100MHz) for all the internal path (from one TX port to an RX port), reduced to 220ns if taking into account the latency of the FIFO (60ns between FIFO writing and empty signal low).

2.2 Communication IP RTL kernel

The INFN Communication IP, developed in VHDL, is implemented as an RTL-kernel in Xilinx Vitis, a High-Level Synthesis framework which allows to develop, debug and optimize accelerated applications using standard programming languages for both software and hardware components.
In the Vitis application, an RTL IP from the Vivado Design Suite is packaged as Xilinx object form (XO) file for implementation in the programmable logic (PL) region of the target platform.

In Figure 2.7 the packaged IP generated within the Vivado Design Suite is depicted.

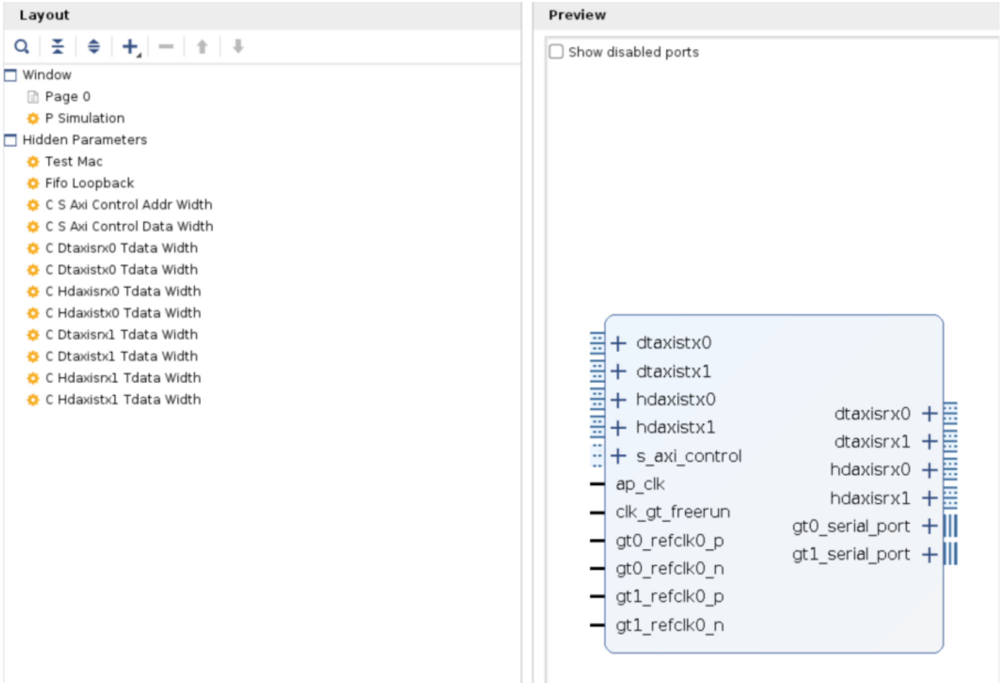

Figure 2.7 Communication IP with 2 IntraNode and 2 InterNode ports packaged in Vivado Design Suite

The kernel interfaces are used to exchange data with the host application, other kernels or device I/Os:

- s_axi_control is the AXI4-Lite slave interface that allows a host application to interact with kernels by reading or writing registers. The I/O ports of this interface is reported in Table 2.2
- Dtaxis* and Hdaxis* are streaming interfaces used to transfer data directly from/to other kernels. Since an AXI4-Stream interface transfers data in a sequential streaming manner, it cannot be used with arguments that are both read and written, two interfaces are requested for each IntraNode port (Tx and Rx interfaces are shown in tables 2.3 and 2.4).
- Gt*_serial_ports are streaming interfaces connected to QSFP+ ports, used to communicate with other devices (interNode ports).
  Each InterNode port requires a low-jitter reference clock (gt*_refclock0_p/n @ 161.13 MHz) for generating and recovering high-speed serial clocks, while a single stable clock (clk_gt_freerun @ 100 MHz) is used for mixed-mode clock manager (MMCM) synchronization.
- Ap_clk is the clock for the switch/ register logic.

Signal Name	I/O	Description
S_AXI_AWADDR (11 downto 0)	I	Write Address
S_AXI_AWVALID	I	Write Address Valid
S_AXI_AWREADY	O	Write Address Ready
S_AXI_WDATA (31 downto 0)	I	Write Data
S_AXI_WSTB (31 downto 0)	I	Write Strobes
S_AXI_WVALID	I	Write Valid

S_AXI_WREADY	O	Write Ready
S_AXI_BRESP (1 downto 0)	O	Write Response
S_AXI_BREADY	I	Write Response Ready
S_AXI_BVALID	O	Write Response Valid
S_AXI_ARADDR (11 downto 0)	I	Read Address
S_AXI_ARVALID	I	Read Address Valid
S_AXI_ARREADY	O	Read Address Ready
S_AXI_RDATA (31 downto 0)	O	Read Data
S_AXI_RRESP (1 downto 0)	O	Read Response
S_AXI_RREADY	I	Read Valid
S_AXI_RVALID	O	Read Ready

Table 2.2: AXI4-Lite Slave Interface signals

Signal Name	I/O	Description
TDATA	I	Write Data
TVALID	I	Write Data Valid
TREADY	O	Write Data Ready
TKEEP	I	Write Data byte qualifier
TLAST	I	Write last byte

Table 2.3: Tx streaming Interfaces signals

Signal Name	I/O	Description
TDATA	O	Read Data
TVALID	O	Read Data Valid
TREADY	I	Read Data Ready
TKEEP	O	Read Data byte qualifier
TLAST	O	Read last byte

Table 2.4: Rx streaming Interfaces signals

The Communication IP kernel is integrated with kernels written in HLS in a framework called APEIRON (see Section 3).

3 APEIRON

The Communication IP is the main enabling component for the APEIRON framework, defined as the general architecture of an FPGA-based distributed stream processing platform and the corresponding software stack. The Communication IP was co-designed with the APEIRON software stack in order to achieve very low-latency and scalable bandwidth (via IP design reconfiguration) between processing tasks defined as High-Level Synthesis Kernels.
Implementing direct communication between tasks deployed on FPGAs without involving host CPU and system bus resources, the Communication IP improves the energy efficiency of the execution platform (Objective Energy efficiency) for what concerns communication between accelerators.

Starting from a YAML configuration file describing the attributes of each HLS kernel, namely its number of input and output channels and the IntraNode port of the Communication IP to which it is connected, the APEIRON framework links the Communication IP and the HLS kernels that are connected to it and generates the bitstream for the overall design.
The only requisite that HLS kernels must satisfy is in the format of their prototype that must be in 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. The communication between kernels is expressed through a lightweight C++ API (HAPECOM) 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) without knowing the details of the underlying packet communication protocol.

The 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);
```

where:

`dest_node` is the n-Dim coordinate of the destination node (FPGA) in an 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).

The Communication Library leverages AXI4-Stream Side-Channels to encode all the information needed to forge the packet header.
Adaptation toward/from IntraNode ports of the Routing IP is done by two APEIRON IPs: Aggregator and Dispatcher, shown in Figure 3.1. The Dispatcher receives incoming packets from the Routing IP and forwards them to the right input channel, according to the relevant fields of the header. The Aggregator receives outgoing packets from the task and forges the packet header, filling then the header/data FIFOs of the Routing IP.

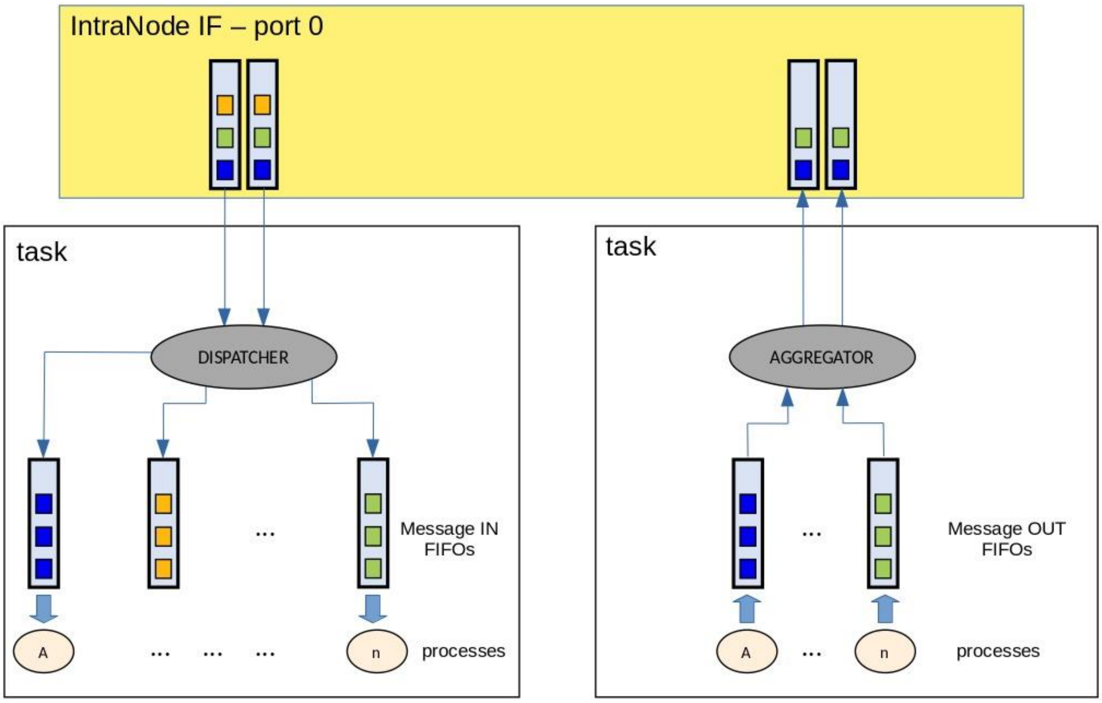

Figure 3.1 Interface between Intranode Port 0 and the corresponding HLS Task (task_id 0), Messages IN FIFOs are identified by the ch_id APIs parameter

4 Resource usage

In Tables 4.1 and 4.2 we report the resource utilization generated by the Vivado tool building the system for the U200 and the U280 cards for the Latency test setup.

For both cards, the occupancy (in terms of LUT, REG, BRAM) is very low, allowing us to easily add new features to the Communication IP while leaving a considerable fraction of the resources to the implementation of the HLS kernels.

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	203705 [17.23%]	23418 [3.96%]	288269 [12.19%]	328 [15.19%]	20 [2.08%]	7 [0.10%]
User Budget	978535 [100.00%]	568422 [100.00%]	2076211 [100.00%]	1832 [100.00%]	940 [100.00%]	6833 [100.00%]
Used Resources	17662 [1.80%]	1192 [0.38%]	25210 [1.21%]	54 [2.95%]	0 [0.00%]	0 [0.00%]
Unused Resources	960873 [98.20%]	567230 [99.79%]	2051001 [99.79%]	1778 [97.05%]	940 [100.00%]	6833 [100.00%]
aggregator_0	790 [0.08%]	0 [0.00%]	1536 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	790 [0.08%]	0 [0.00%]	1536 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	790 [0.08%]	0 [0.00%]	1536 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	790 [0.08%]	0 [0.00%]	1536 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	1147 [0.12%]	0 [0.00%]	2008 [0.10%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	1147 [0.12%]	0 [0.00%]	2008 [0.10%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_1	1147 [0.12%]	0 [0.00%]	2008 [0.10%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	1147 [0.12%]	0 [0.00%]	2008 [0.10%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
krnl_sr	13787 [1.41%]	1192 [0.21%]	18122 [0.87%]	38 [2.07%]	0 [0.00%]	0 [0.00%]
krnl_sr_1	6893 [0.70%]	596 [0.10%]	9061 [0.44%]	19 [1.04%]	0 [0.00%]	0 [0.00%]
krnl_sr_2	6894 [0.70%]	596 [0.10%]	9061 [0.44%]	19 [1.04%]	0 [0.00%]	0 [0.00%]

Table 4.1 Resource usage report of the performance test setup (see Figure 5.7) for Alveo U200 card

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	104132 [7.99%]	9988 [1.66%]	153221 [5.88%]	202 [10.02%]	20 [2.08%]	4 [0.10%]
User Budget	1198588 [100.00%]	590492 [100.00%]	2454139 [100.00%]	1814 [100.00%]	940 [100.00%]	9020 [100.00%]
Used Resources	60353 [5.04%]	16667 [2.82%]	82903 [3.38%]	179 [9.87%]	0 [0.00%]	0 [0.00%]
Unused Resources	1138235 [94.96%]	573825 [97.18%]	2371236 [96.62%]	1635 [90.13%]	940 [100.00%]	9020 [100.00%]
Textarossa_switch_synt	790 [0.07%]	15465 [2.62%]	57587 [2.35%]	125 [6.89%]	0 [0.00%]	0 [0.00%]
Textarossa_switch_1	790 [0.07%]	15465 [2.62%]	57587 [2.35%]	125 [6.89%]	0 [0.00%]	0 [0.00%]
aggregator_0	790 [0.07%]	0 [0.00%]	1536 [0.06%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	790 [0.07%]	0 [0.00%]	1536 [0.06%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	790 [0.07%]	0 [0.00%]	1536 [0.06%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	790 [0.07%]	0 [0.00%]	1536 [0.06%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	1147 [0.10%]	0 [0.00%]	2008 [0.08%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	1147 [0.10%]	0 [0.00%]	2008 [0.08%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_1	1147 [0.10%]	0 [0.00%]	2008 [0.08%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	1147 [0.10%]	0 [0.00%]	2008 [0.08%]	8 [0.44%]	0 [0.00%]	0 [0.00%]
krnl_sr	13856 [1.16%]	1202 [0.20%]	18228 [0.74%]	38 [2.09%]	0 [0.00%]	0 [0.00%]
krnl_sr_1	6925 [0.58%]	601 [0.10%]	9114 [0.37%]	19 [1.05%]	0 [0.00%]	0 [0.00%]
krnl_sr_2	6931 [0.58%]	601 [0.10%]	9114 [0.37%]	19 [1.05%]	0 [0.00%]	0 [0.00%]

Table 4.2 Resource usage report of the performance test setup (see Figure 5.7) for Alveo U280 card

5 Test and Performance

Here we describe the design validation and performance tests and report results and performance measurement for the intermediate release of the Communication IP.

As testbench, we used a system composed of 2 interconnected Xilinx Alveo U200 FPGAs managed by different hosts.

5.1 Design validation tests

To validate and start debugging of the Communication IP, we initially used the internal Packet_Generator. This block also allowed us to measure latency for one packet of length equal to 16 Bytes sent to the same node, using either internal loopback or channel termination (Figure 5.1 and Figure 5.2).

In both tests, *Test ok* shows the register 20 content (31 means “00110001”, that is Packet generator FSM in IDLE state, Consumer FSM in IDLE state and all packets were received with correct payload).

```
[rossi@apeirone xdma]$ ./internal_loopback_test -b ~/test_FRA.xclbin -n 1 -l 16 -i
Device name: xilinx_u200_gen3x16_xdma_base_1
Device bdf: 0000:af:00.1
Device max freq: 100
Device m2m: 1
Device nodma: 0
Device kdma: 1

Press a key to continue

Resetting switch
Starting packet generator ...
Test ok: 31
Clock cycle: 28
```

Figure 5.1 IntraNode 0 TX FIFO towards IntraNode 0 RX FIFO latency for one packet (length = 16 byte) sent by internal packet_generator

```
[rossi@apeirone xdma]$ ./internal_loopback_test -b ~/test_FRA.xclbin -n 1 -l 16 -i -c
Device name: xilinx_u200_gen3x16_xdma_base_1
Device bdf: 0000:af:00.1
Device max freq: 100
Device m2m: 1
Device nodma: 0
Device kdma: 1

Press a key to continue

Resetting switch
Starting packet generator ...
Test ok: 31
Clock cycle: 90
```

Figure 5.2 IntraNode 0 TX FIFO towards IntraNode 0 RX FIFO in cable loopback setup, for one packet (16 byte) sent by internal packet generator

5.2 Performance tests

Here we report the measured performance of the intermediate release of the Communication IP in terms of bandwidth and latency of intra-node and inter-node communication operations between HLS kernels endpoints.

5.2.1 Bandwidth

A bandwidth test is carried out by transferring multiple data packets with fixed payload size from a “sender” HLS kernel which reads data from the source buffer in FPGA memory (either DDR or BRAM) and pushes them through the Communication IP to another FPGA, where a “receiver” HLS kernel writes data into the destination buffer in memory. After receiving the number of data packets whose integrated payload adds up to the size of the receive buffer, the second FPGA pings back a single “ACK” packet with minimal payload to confirm the reception, as shown in Figure 5.3. The total data sent during this test is summed and then divided by the time (measured on the sender node) elapsed between the start of the multiple packets send and the completion of the receive operation of the ACK packet.

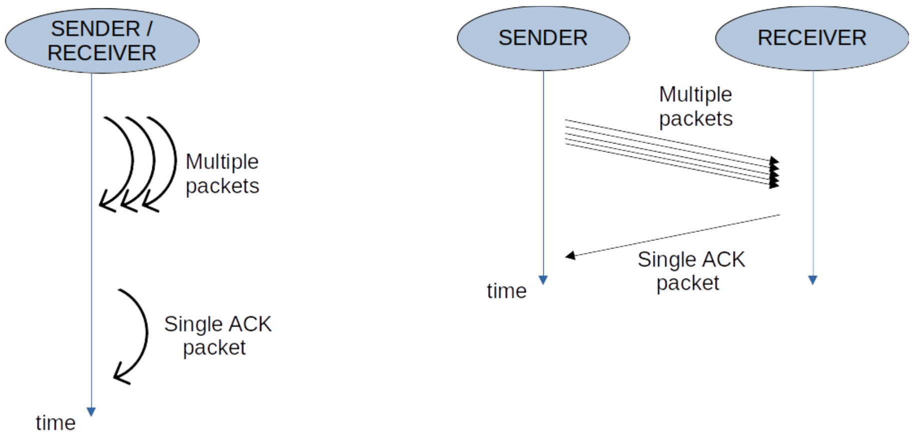

Figure 5.3 Illustration of the bandwidth test

Results measured with send and receive buffers allocated on the FPGA BRAM are shown in Figure 5.4.

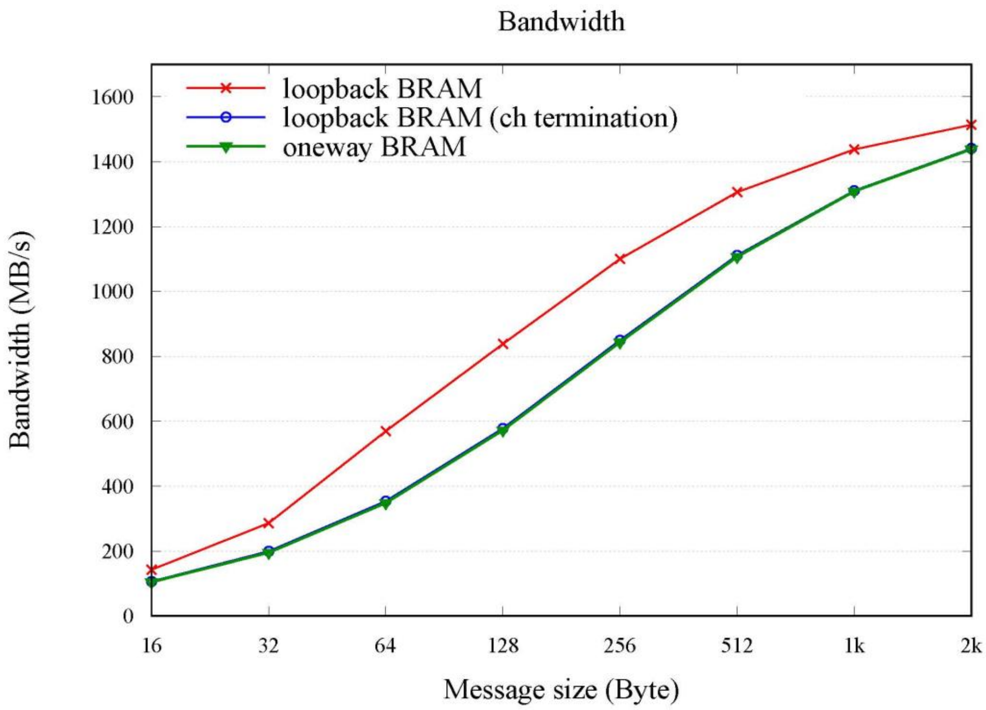

Figure 5.4 Measured bandwidth between HLS Kernels for an intra-node (loopback - red line) and inter-node communication (loopback - blue line, oneway – green line), with send and receive buffers allocated on BRAM memory. Note the blue line, representing the measurements taken emulating an inter-node communication on a single FPGA using a loopback termination on one QSFP+ port, is practically overlapped with the one measured between two nodes (oneway – green line)

The same set of measurements were repeated using the FPGA DDR to allocate send/receive buffers instead of BRAM. Results are reported in the plot in Figure 5.5, showing the limiting effect of the DDR memory controller on the overall reachable bandwidth. In addition, in Figure 5.5 it is possible to notice that, referring to the BRAM cases, the bandwidth tends to saturate while increasing the size of the packets sent. In particular, for packets of size 2 kB the bandwidth reaches a value of ~12.0 Gbps for the intra-node loopback BRAM case (blue line), with a maximum theoretical value of raw bandwidth equal to 12.8 Gbps: the difference is mainly due to the packet protocol overhead. For the slightly lower maximum value of 11.3 Gbps reached in the inter-node oneway BRAM case (fuchsia line), the overhead due to serialization and 64b/66b encoding over the external channel must be accounted.

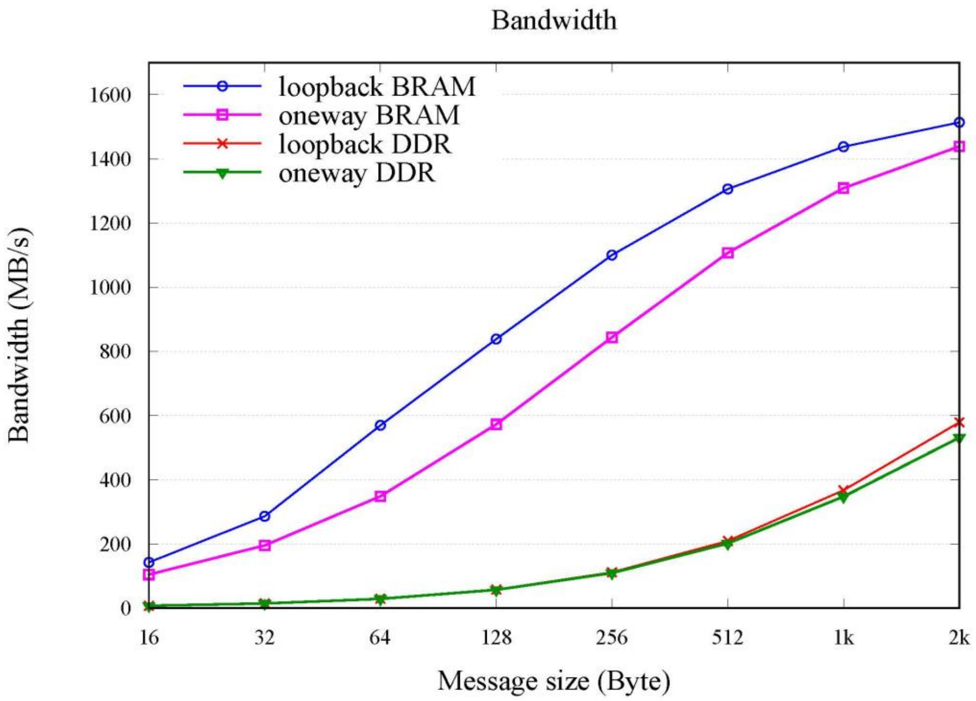

Figure 5.5 Comparison between measured bandwidth between HLS Kernels for an intra-node (loopback) communication and inter-node (oneway) communication using BRAM and DDR to allocate send/receive buffers

5.2.2 Latency

A latency test is performed using an HLS kernel (krnl_sr, as reported in tables 4.1 and 4.2), configurable by the host in different operating modes. In detail, in "send_receive" mode the kernel reads a payload data item from the FPGA memory (either BRAM or DDR) and sends and receives it through/from the Communication IP to/from a second interconnected FPGA, where an HLS kernel in "pipe" mode has the task of receiving a single packet and bouncing it back to the initiator FPGA (as shown in Figure 5.6), allowing the measurement of inter-node latency.

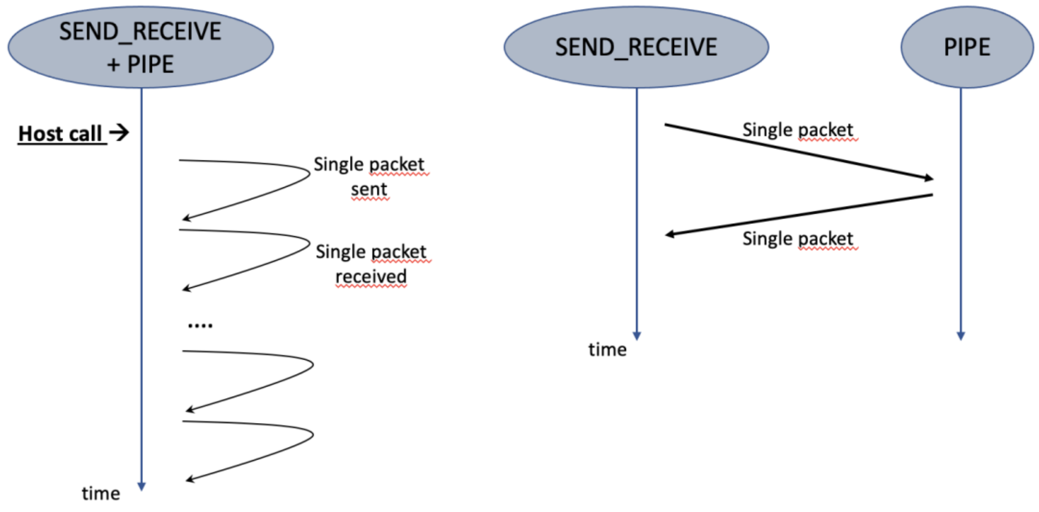

Figure 5.6 Illustration of the latency test

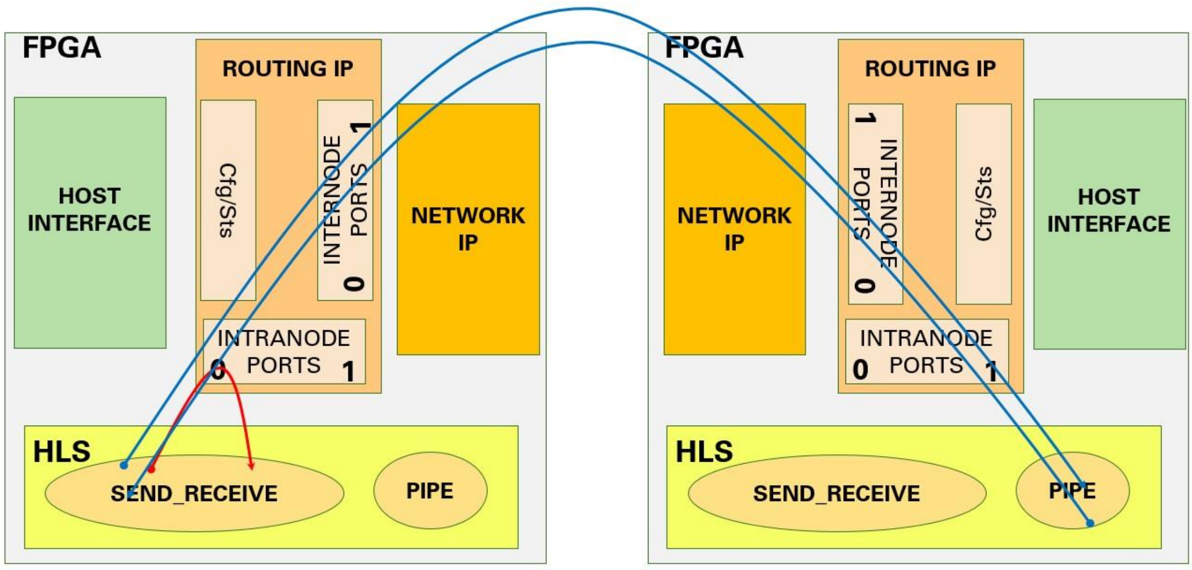

Figure 5.7 Testbench design illustration. The arrows describe different flows of data depending on the test performed: “Localloop, port 0 to port 0” (red arrow), “Roundtrip, port 0 to port1” (blue arrows)

Since the HLS kernel in "send_receive" mode on the initiator FPGA is started via host code while the HLS kernel in "pipe" mode is free-running, the former is launched with a repetition parameter of 1 million send/receive operations before termination in order to minimize the contribution of the host call overhead on the overall time elapsed from the start of the first packet send to the completion of the last packet receive (measured on the host). The latency is then obtained by dividing half the elapsed time measured by the number of packets.

As can be seen in Figure 5.7, the tests performed are basically two:

- Roundtrip, where packets are transmitted between different intranode ports of two interconnect FPGAs

- Locallop, where packets are transmitted back and forth on the same intranode port of a single FPGA

The results obtained are reported in Figure 5.8, indicating the type of tests performed and what kind of FPGA memory is used. In detail, the result obtained shows how the latency values get worse when working with DDR memory, due to overhead issues and to the time required to load the sent buffer from CPU on the FPGA and to read the received buffer from the FPGA to the CPU (we refer to these as “sync” operations, which are not present in the BRAM test cases). In accordance with the specifications reported in deliverable *D2.1- Consolidated specs of accelerators IPs*, latency reaches a value slightly below 1 us for 16 B payload packets in the inter-node roundtrip BRAM case (yellow line), and a value of ~250ns in the intra-node localloop BRAM case (blue line).

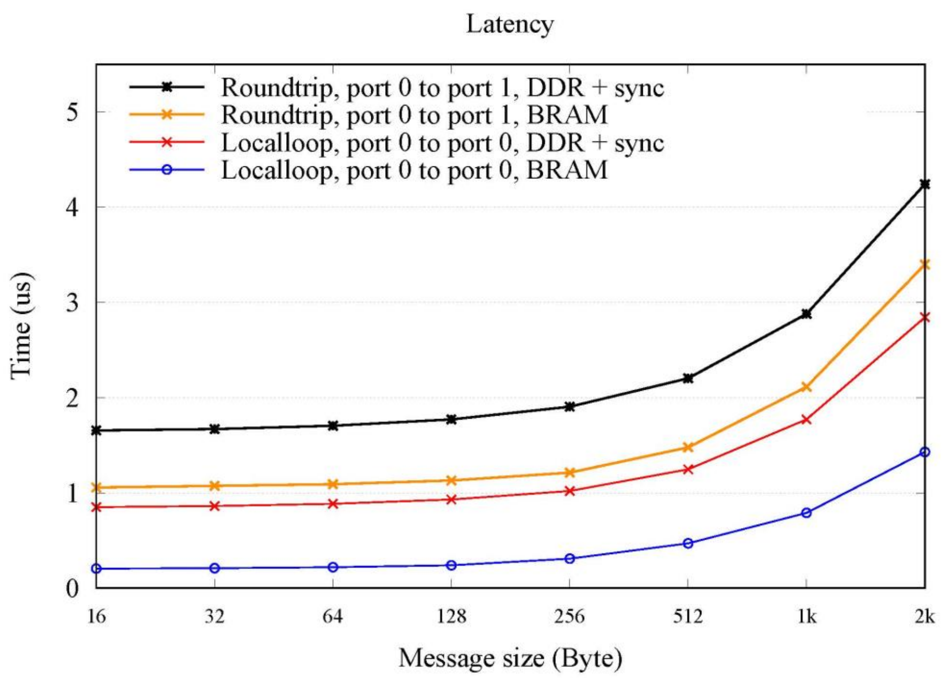

Figure 5.8 Comparison of measured latency between HLS Kernels for an intra-node (loopback) communication and inter-node (roundtrip) communication using BRAM and DDR to allocate send/receive buffers

6 State of the art

Nowadays, FPGAs represent one of the main architectures for HPC applications, considering the impending end of Moore’s Law and Dennard scaling. In addition to this, this type of accelerators is well suited to develop customized algorithms, combining the scalable parallel processing capability of an Application Specific Integrated Circuit (ASIC) with the reprogrammability typical of such type of devices.

In modern development, and in dedicated networks, multiple FPGAs clusters are emplaced to map large HPC kernels by exploiting the low-latency communication capability of these accelerators. However, despite FPGAs high-speed transceiver links, a certain network flexibility with very large clusters could be required in order to map applications’ workloads and to strategically maximize resource utilization and performance. In this direction, many solutions of scalable switched FPGA cluster have been developed, where, for example, the transceiver links are physically connected to ports of high-speed Ethernet switches [8], in an indirect network setup, as in the Virtual Circuit-Switching Network (VCSN) [5] or by implementing FPGAs as Network Interface Cards (NICs), as, for example, in the EasyNet open source networking stack [6] and in the Corundrum open-source network interface [7].

Corundrum and EasyNet are based on a direct network for FPGA inter-communication using a 100 Gbps TCP/IP stack, even if they present a difference in what concern their implementation: in fact, Corundrum is implemented in Verilog HDL, rather than being based on the HLS Xilinx Vitis platform as in the EasyNet case. This difference makes EasyNet much like the APEIRON framework, considering the possibility for the user to implement in the setup a custom kernel connected to the network in an optimized way via communication primitives callable as functions in an HLS library (as it is with the send() and receive() APIs in the HAPECOM library). However, unlike EasyNet with TCP/IP stack, the INFN Communication IP used in the APEIRON framework is more similar to the custom protocols developed in some project at the state of art for which no inter-FPGA backpressure is needed, such as the E40G setup presented in paper [5], or the network infrastructure underlying the Reconfiguration over Network (REoN) protocol [9], which can transport partial bit files via network resource management APIs to a FPGA empowered network node, using standard 10 Gbps Ethernet.

7 Conclusions

In this deliverable we described the Communication IP in detail and showed preliminary synthesis and results of tests developed to validate the design and assess its current performance.

In particular, we used some synthetic tests to measure the bandwidth and the latency in both inter- and intra-node communication. The results are promising and in line with the expected specifications reported in deliverable *D2.1-Consolidated specs of accelerators IPs*.

This document, along with the Communication IP with two intraNode ports packaged as Xilinx object (XO) file for both the U200 and U280 platform, an APEIRON example design, and a demo video showing the performance tests described in section 5.2, are publicly available for download as archive zip file on the deliverable section of the TEXTAROSSA web site (https://textarossa.eu/dissemination/deliverables/).

In the near future, we foresee to increase the internal datapath of the IP to 256 bits and to use the Aurora transceiver with 4 lanes to support applications requiring an increased communication bandwidth. Furthermore, we plan to implement a new channel interface based on the Xilinx® 10G/25G High Speed Ethernet Subsystem in order to enable interoperability with standard switched networks, either to support (e.g. UDP over IP) input and output streams or to implement a switched network topology.

8 References

[1] A New Computer Communication Switching Technique. P. Kermani and L. Kleinrock, Virtual Cut-Through: Comput. Networks 3 (1979) 267.

[2] Deadlock-free message routing in multiprocessor interconnection. Seitz, W. J. Dally, and C. L. 1987. 5, : Computers, IEEE Transactions on, 1987, Vol. C.36, p. 547–553.

[3] APEnet+ 34 Gbps Data Transmission System and Custom Transmission Logic. 2013.

[4] Andrea Biagioni, Paolo Cretaro, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Stanislao Paolucci, Luca Pontisso, Francesco Simula and Piero Vicini, "EuroEXA Custom Switch: an innovative FPGA-based system for extreme scale computing in Europe", EPJ Web of Conferences 245, 09004, 2020.

[5] Antoniette Mondigo, Tomohiro Ueno, Kentaro Sano and Takizawa Hiroyuki, "Comparison of Direct and Indirect Networks for High-Performance FPGA Clusters", International Symposium on Applied Reconfigurable Computing (ARC), 2020.

[6] He, Z.; Korolija, D.; Alonso, G. EasyNet: 100 Gbps Network for HLS. In Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL 2021), Dresden, Germany, 30 August 30–3 September 2021.

[7] Forencich, A.; Snoeren, A.C.; Porter, G.; Papen, G. Corundum: An open-source 100-Gbps NIC. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–5 May 2020; pp. 38–46.

[8] Tomohiro Ueno and Kentaro Sano. 2023. VCSN: Virtual Circuit-Switching Network for Flexible and Simple-to-Operate Communication in HPC FPGA Cluster. ACM Trans. Reconfigurable Technol. Syst. 16, 2, Article 25 (June 2023), 32 pages. https://doi.org/10.1145/3579848

[9] V. Mishra, Q. Chen and G. Zervas, "REoN: A protocol for reliable software-defined FPGA partial reconfiguration over network," 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 2016, pp. 1-7, doi: 10.1109/ReConFig.2016.7857184.

Appendix A. Relevant source codes

Bandwidth test host pseudocode

```
device.load_xclbin(bitstream);
Allocate_recv_buffer(device, buf_size);
Allocate_send_buffer(device, packet_size);
Fill_send_buffer();
Send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
switch.write_register(auto-toggle reset);
kswitch.write_register(local_coord);
(only for localloop test):  //kswitch.write_register(overwrite destination);
kswitch.write_register(threshold);
kswitch.write_register(credit);
If node_sender:
        Run_kernel_receiver(recv_buffer, 1);
gettimeofday(&startTime,NULL);  //start time measurement
        run_kernel_sender (receiver_coord, npackets, packet_size, send_buffer);
        ksender_run.wait();
        kreceiver_run.wait();
gettimeofday(&endTime,NULL); //stoptime measurement
elapsedTime = elapsed(startTime,endTime);
        BW = (npackets*packet_size)/elapsedTime);
 If node_receiver:
Run_kernel_receiver(recv_buffer, npackets);
kreceiver_run.wait();
recv_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
        Run_kernel_sender(sender_coord, 1, 16, send_buffer); //send back 1 packet of size 16B
        ksender_run.wait();
```

Bandwith test "kernel sender" pseudocode (example for DDR test)

```
int nword = packet_size / sizeof(word_t);
Foreach (packet){
        Header = Fill_header;
        Hdr_fifo_out.write(Header);
        foreach (word) {
        data_fifo_out.write(data_word);
}
        Footer = fill_footer()
Hdr_fifo_out.write(footer);
}
```

Bandwith test "kernel receiver" pseudocode (example for DDR test)

```
Foreach (packet){
hdr_fifo_in.read(hdr);
len = hdr.packet_size;
```

```
N_words = len/sizeof(word)
        Foreach(word in N_words){
                word[j] = data_fifo_in.read();
}
        header_fifo_in.read(footer)
}
```

Latency test host pseudocode

```
device.load_xclbin(bitstream);
If !bram_usage:
        Allocate_recv_buffer(device, buf_size);
        Allocate_send_buffer(device, packet_size);
        Fill_send_buffer();
        Send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
switch.write_register(auto-toggle reset);
kswitch.write_register(local_coord);
kswitch.write_register(threshold);
kswitch.write_register(credit);
If initiator FPGA:
gettimeofday(&startTime,NULL);  //start time measurement
        run_kernel_sender_receiver (destination_coord, npackets, packet_size, send_buffer, recv_buffer,
                                    bram_usage);
        ksender_receiver_run.wait();
        If !bram_usage:
                recv_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

gettimeofday(&endTime,NULL); //stoptime measurement
elapsedTime = elapsed(startTime,endTime);
Latency = (elapsedTime/2)/npackets;
```

Latency test "kernel sender_receiver" (krnl_sr) pseudocode

```
Foreach (packet){
        If bram_usage:
                memory_in = local_BRAM_buffer_in;
                memory_out = local_BRAM_buffer_out;
        send(memory_in, packet_size, coord, task_id, ch_id, data_fifo_out);  //Communication Library
        receive(ch_id, memory_out, data_fifo_in);
}
```

Latency test "kernel pipe" (krnl_pipe) pseudocode

```
Foreach (packet){
        receive(ch_id, local_memory, data_fifo_in); //Communication Library APIs
        send(local_memory, packet_size, coord, task_id, ch_id, data_fifo_out);
}
```

Appendix B. Integration of Communication IP in Vitis environment

Pre-requisites

- Xilinx Alveo U200/U280 card

- Xilinx Vitis 2021.1

(https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2021-1.html)

- Xilinx runtime (XRT), XDMA Deployment Target Platform, and XDMA Development Target Platform

(https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#gettingStarted)

Environment

```
> source /opt/Xilinx/Vitis/2021.1/settings64.sh
> source /opt/xilinx/xrt/setup.sh
```

(Dependent on your local installation paths).

Example kernel

The only requisite for an HLS kernel to connected to one of the Communication IP, is to be compliant with the following defintion:

```
void krnl_example(
            <optional parameters>,
            header_stream_t message_hdr_in[N_INPUT_CHANNELS],
            message_stream_t message_data_in[N_INPUT_CHANNELS],
            header_stream_t message_hdr_out[N_OUTPUT_CHANNELS],
            message_stream_t message_data_out[N_OUTPUT_CHANNELS]
)
```

So that is has N_INPUT_CHANNELS and N_OUTPUT_CHANNELS to receive/send incoming/outgoing messages through the Communication IP.
The `header_stream_t and message_stream_t` types are defined as:

```
typedef hls::stream<uint128_t> message_stream_t;
typedef hls::stream<apenet_header_t> header_stream_t;
```

And the `apenet_header_t`, representing an apenet protocol header shown in figure 2.3, is defined as:

```
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[4];
        uint64_t u[2];
} apenet_header_t;
```

Please refer to directory D2.8/APEIRON_example_design/include/ for further information.

Build steps

After the kernel code is written, you can build the application, generating the FPGA binary file (.xclbin).

First step is to write the vpp_linker.cfg Vitis project configuration file, that specifies the operational clock frequency and the interconnections between the components' ports.

For example, the following vpp_linker.cfg specifies a clock frequency of 100 MHz and connects krnl_example_0 and krnl_example_1 respectively to Intranode_port_0 and intranode_port_1 of the Communication IP.

```
kernel_frequency=0:100|1:1

[connectivity]
stream_connect=TextaRossa_switch_1.dtaxisrx0:krnl_example_0.dt_in
stream_connect=TextaRossa_switch_1.hdaxisrx0:krnl_example_0.hd_in
stream_connect=krnl_example_0.dt_out:TextaRossa_switch_1.dtaxistx0
stream_connect=krnl_example_0.hd_out:TextaRossa_switch_1.hdaxistx0
stream_connect=TextaRossa_switch_1.dtaxisrx1:krnl_example_1.dt_in
stream_connect=TextaRossa_switch_1.hdaxisrx1:krnl_example_1.hd_in
stream_connect=krnl_example_1.dt_out:TextaRossa_switch_1.dtaxistx1
stream_connect=krnl_example_1.hd_out:TextaRossa_switch_1.hdaxistx1
```

After this, starting from the .xo files of the communication IP and of the user kernels, it is possible to launch the build process (this takes a couple of hours at least) for U200 board:

```
> v++ -t hw --platform xilinx_u200_gen3x16_xdma_1_202110_1 -s --
temp_dir _tmp_build --log_dir _tmp_build/logs --report_dir
_tmp_build/reports -I include --link --config vpp_linker.cfg --xp
param:compiler.userPostDebugProfileOverlayTcl=scripts/post_sys_link.tcl
--messageDb _tmp_build/test.xclbin.mdb -o test.xclbin
TextaRossa_switch_2in_2ex_U200.xo krnl_example.xo
```

And for the U280 board:

```
>  v++ -t hw --platform xilinx_u280_xdma_201920_3 -s --temp_dir
_tmp_build --log_dir _tmp_build/logs --report_dir _tmp_build/reports -I
include --link --config vpp_linker.cfg --xp
param:compiler.userPostDebugProfileOverlayTcl=scripts/post_sys_link.tcl
--messageDb _tmp_build/test.xclbin.mdb -o test.xclbin
TextaRossa_switch_2in_2ex_U280.xo krnl_example.xo
```

The generated binary (test.xclbin) can then be used to program the FPGA of the accelerator card.

Appendix C. APEIRON Example Design

Pre-requisites

- Xilinx Alveo U200/U280 card

- Xilinx Vitis 2021.1

(https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2021-1.html)

- Xilinx runtime (XRT), XDMA Deployment Target Platform, and XDMA Development Target Platform

(https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#gettingStarted)

Environment

```
> source /opt/Xilinx/Vitis/2021.1/settings64.sh
```

```
> source /opt/xilinx/xrt/setup.sh
```

(Dependent on your local installation paths).

EXAMPLE: Latency test Design

This example design demonstrates the main functionalities of the APEIRON framework, using a Communication IP configured with 2 intranode ports and 2 internode ports. Each port is bidirectional, and each direction sports a header/data FIFO couple according to the packet protocol described in Section 2.

The source code for this example design can be found in the D2.8 tree: D2.8/APEIRON_example_design.

Referring to the testbed in Figure 5.7, the two replicas of krnl_sr() communicating through the switch, and defined as:

```
void krnl_sr(){
<optional parameters>,
           message_stream_t message_data_in[N_INPUT_CHANNELS],
           message_stream_t message_data_out[N_OUTPUT_CHANNELS]
)
```

have N_INPUT_CHANNELS and N_OUTPUT_CHANNELS to receive/send incoming/outgoing messages.

As described in Section 5.2.2, Ports 0 and 1 of the router are connected to the *krnl_sr()* HLS kernels, through the autogenerated *dispatcher_0/1()* and *aggregator_0/1()*.

So the *dispatcher...()* and *aggregator...()* kernels work as adaptors from and toward the single bidir channel of the router port.

The host application orchestrates the execution of the test, initializing the send/receive buffers in the device global memory and launching the HLS kernels.

In the latency_test, a packet is sent from the node_0:port_0 to node_0:port_<destination port> where it is received and then sent back. In the example design included in the deliverable archive file, we used a simplified configuration with a single node, where communication happens between the two interNode ports of the same router connected to each other (Localloop configuration).

The developer has to write a YAML configuration file (*config.yaml*) describing the attributes of each HLS kernel, namely the number of its input and output channels and the IntraNode port of the Communication IP to which it is connected, along with the number of router internode ports of the Communication IP (*links*) and the target operating frequency of the overall design in MHz (*freq),* taking in consideration that the current validated operating frequency for the Communication IP is 100 MHz.

The APEIRON configuration file for this example design is:

```yaml
kernels:

        - name: krnl_sr_1

          input_channels: 4

          output_channels: 4

          switch_port: 0


        - name: krnl_sr_2

          input_channels: 4

          output_channels: 4

          switch_port: 1


config:

  freq: 100

  links: 2
```

Having this file as input, the APEIRON framework links the Communication IP and the HLS kernels that are connected to it and generates the bitstream for the overall design, according to the following steps.

Build steps

Make sure that the .xo file of the Communication IP matches the execution platform, checking the symbolic link contained in the D2.8/APEIRON_example_design/ip_repo directory:

```
> ls -la
lrwxrwxrwx. 1 lonardo users      39 Apr 22 18:36 TextaRossa_switch_2in_2ex.xo ->
../../TextaRossa_switch_2in_2ex_U200.xo
```

In this case the configuration is set to generate a firmware for the U200 platform, in case one wishes to generate firmware for the U280, the following commands must be issued:

```
> cd D2.8/APEIRON_example_design/ip_repo/
> ln -s ../../TextaRossa_switch_2in_2ex_U280.xo TextaRossa_switch_2in_2ex.xo
```

First step is to generate the vpp_linker.cfg Vitis project configuration file, the operational clock frequency and the interconnections between the components, using the config.yaml as input:

```
> ./generate.py
```

Select the correct platform (U200 or U280) in the Makefile by setting the PLATFORM variable.

After this step, it is possible to launch the build process (this takes a couple of hours at least, refer to make.log file to inspect a successfully build process).

> *make*

```
mkdir -p _tmp_build

v++ -t hw --platform xilinx_u200_gen3x16_xdma_1_202110_1 -s --temp_dir _tmp_build -
-log_dir _tmp_build/logs --report_dir _tmp_build/reports  -I include --config
hw_hls/krnl_sr.cfg --messageDb _tmp_build/krnl_sr.xo.mdb -o hw_hls/krnl_sr.xo
hw_hls/krnl_sr.cpp

Option Map File Used: '/opt/Xilinx/Vitis/2021.1/data/vitis/vpp/optMap.xml'


****** v++ v2021.1.1 (64-bit)

  **** SW Build 3278995 on 2021-07-20-20:33:48

    ** Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.


INFO: [v++ 60-1306] Additional information associated with this v++ compile can be
found at:

        Reports:
/apotto/home1/homedirs/lonardo/D2.8/APEIRON_example_design/_tmp_build/reports/krnl_
sr

        Log files:
/apotto/home1/homedirs/lonardo/D2.8/APEIRON_example_design/_tmp_build/logs/krnl_sr

Running Dispatch Server on port: 37243

INFO: [v++ 60-1548] Creating build summary session with primary output
/apotto/home1/homedirs/lonardo/D2.8/APEIRON_example_design/hw_hls/krnl_sr.xo.compil
e_summary, at Sat Apr 22 19:26:58 2023

INFO: [v++ 60-1316] Initiating connection to rulecheck server, at Sat Apr 22
19:26:58 2023

Running Rule Check Server on port:41135

INFO: [v++ 60-1315] Creating rulecheck session with output
'/apotto/home1/homedirs/lonardo/D2.8/APEIRON_example_design_TODELETE/_tmp_build/rep
orts/krnl_sr/krnl_sr_guidance.html', at Sat Apr 22 19:27:00 2023

INFO: [v++ 60-895]   Target platform:
/opt/xilinx/platforms/xilinx_u200_gen3x16_xdma_1_202110_1/xilinx_u200_gen3x16_xdma_
1_202110_1.xpfm

...
```

Examine design reports

Use the vitis_analyzer tool to visualize and navigate the relevant reports for the design. Run the following command:

```
> vitis_analyzer D2.8/APEIRON_example_design/test.xclbin.link_summary
```

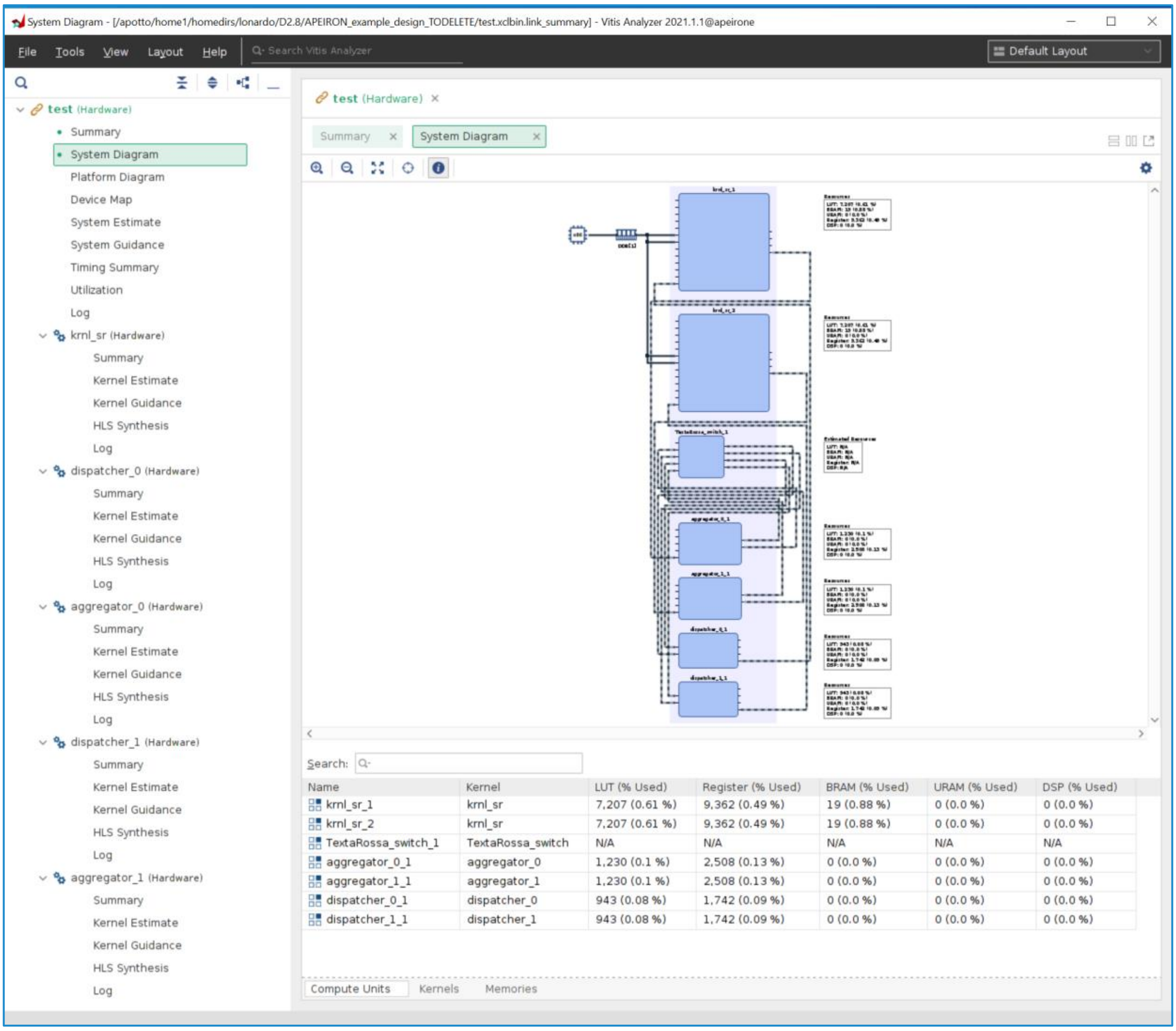

Figure C. 1 Inspection of the design report through the GUI of the Vitis Analyzer

Execution

To program the FPGA and launch the communication latency test between kernels connected to port 0 and to <destination port> of the switch in the same FPGA (for this design configuration), using <number of packets> packets of size <packet size>:

```
>./latency_test -b test.xclbin -l <packet size> –n <number of packets>
-i <destination port>
```

For example, this is the output of the execution when performing the latency test between port 0 (sender) and port 0 (receiver) using one million packets of size 16B, allocating send and receive buffers in BRAM memory:

```
> ./latency_test --bram --quiet -b test.xclbin -l 16  -n 1000000 -i 0

Packet size: 16 B         Latency: 0.20138 us
```