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

WP2 New accelerator designs exploiting mixed precision

D2.1 Consolidated specs of accelerators IPs

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

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	D2.1
WP No:	WP2
WP Leader:	CINI-UNIPI
Due date:	M12 (March 31, 2022)
Delivery date:	31/05/2022

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)	

Grant Agreement No.: 956831

Deliverable: D2.1 Consolidated specs of accelerators IPs

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

WP2 New accelerator designs exploiting mixed precision

Deliverable number:	D2.1					
Deliverable title:	Consolidated specs of accelerators IPs					
Due date:	M12					
Actual submission date:	M14					
Editor:	Sergio Saponara					
Authors:	S. Saponara, C. Alvarez, D. Jimenez, A. Lonardo, F. Lo Cicero, P. Cretaro, S. Di Matteo, F. Rossi, M. Cococcioni, M. Lo Gerfo					
Work package:	WP2					
Dissemination Level:	Public					
No. pages:	30					
Authorized (date):	31/05/2022					
Responsible person:	Sergio Saponara					
Status:	Plan	Draft	Working	**Final**	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	2022-05-10	S. Saponara	Draft structure
0.2	2022-05-27	S. Saponara, C. Alvarez, D. Jimenez, A. Lonardo, F. Lo Cicero, P. Cretaro, S. Di Matteo, F. Rossi, M. Cococcioni, M. Lo Gerfo	Added contribution of CINI; INFN and BSC
0.3	2022-05-28	S. Saponara	Revised draft
0.4	2022-05-30	S. Saponara	Final

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	F. Magugliani	May 30th, 2022
Checked by Tasks Leaders	S. Saponara, A. Lonardo, C. Alvarez	May 31st, 2022
Checked by WP Leader	Sergio Saponara	May 31st, 2022
Checked by Project Coordinator	Massimo Celino	May 31st, 2022

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 Acronyms

Acronym	Definition
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CINI	Consorzio Interuniversitario Nazionale per l'Informatica
CPU	Central Processing Unit
DNN	Deep Neural Network
FP32	Floating Point 32 bit
FPGA	Field Programmable Gate Array
FTS	Fast Task Scheduler
HE	Homomorphic Encryption
HW	Hardware
HPC	High-Performance-Computing
INFN	Istituto Nazionale di Fisica Nucleare
IP	Intellectual Property
IPR	Intellectual Property Rights
XOF	eXtendable Output Function
PMB	Project Management Board
PPU	Posit Processing Unit
PQC	Post Quantum Cryptography
RISC	Reduced Instruction Set Computer
SEAL	Simple Encrypted Arithmetic Library
SW	Software
RLWE	Ring Learning With Errors
UART	Universal Asynchronous Receiver Transmitter interface
VHDL	VHSIC Hardware Description Language

VPU	Vector Processor Unit

Executive Summary

This document reports the activities done by Textarossa partners CINI (UNIPISA), INFN and BSC with reference to consolidated specifications of accelerator IPs in WP2 and preliminary design and synthesis results, manly in FPGA technology.

1. Introduction

This document reports the activities done by Textarossa partners CINI (UNIPISA), INFN and BSC with reference to consolidated specifications of accelerator IPs in WP2 and preliminary design and synthesis results, manly in FPGA technology.

Particularly, Section 2 deals with specification and preliminary design results of accelerators with mixed-precision for data compression and for efficient computation of DNN (Deep Neural Network).

To this am, posits are used. Growing constraints on memory utilization, power consumption, and I/O throughput have increasingly become limiting factors to the advancement of high performance computing (HPC) and edge computing applications. IEEE-754 floating-point types have been the de facto standard for floating-point number systems for decades, but the drawbacks of this numerical representation leave much to be desired. Alternative representations are gaining traction, both in HPC and machine learning environments. Posits have recently been proposed as a drop-in replacement for the IEEE-754 floating-point representation. The current literature supports posits as a promising alternative to traditional floating-point systems, both as a stand-alone replacement and in a mixed-precision environment. Development and standardization of the posit type is ongoing, and much research remains to explore the application of posits in different domains, how to best implement them in hardware, and where they fit with other numerical representations.

Section 3 shows the design of accelerators for innovative security services based on Post Quantum Cryptographic (PQC) techniques, useful also for homomorphic encryption.

Sections 2 and 3 show how the proposed accelerators can be integrated with RISC-V computing core like the RSC-V 64B Ariane IP and the RISC-V with support of the Vector extension.

Section 4 refers to IP cores for low-latency inter-core and intra-core communications.

Section 5 refers to an IP for hardware-assistance task scheduling in high performance computing systems.

Conclusions are drawn in Section 6.

2. Accelerators with mixed-precision for AI computing and data compression

The goal of this work is twofold

- the design of an IP core for PPU (Posit Processing Unit) to be connected to a 64b RISC-V processor.
- to build a Posit Processing Unit in the form of a co-processor to be attached to a RISC-V processor by extending its Instruction Set Architecture.

We initially focus on the compression abilities of posits by providing a co-processor with only conversions in mind, called light PPU, see Figure 2.1 below:

- IEEE 32-bit float (a.k.a binary32) to posit16/8 and viceversa
- Fixed point (various sizes according to posit) to posit16/8 and viceversa

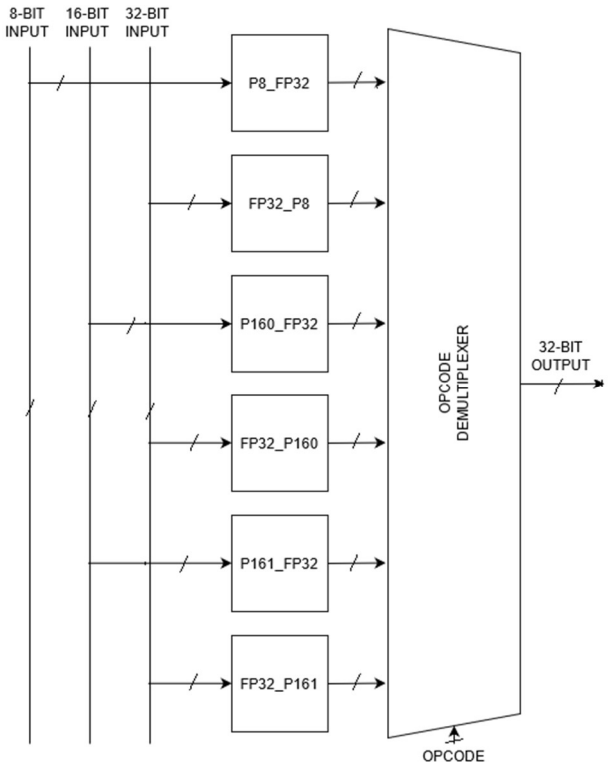


Figure 2.1: Light PPU co-processor with only conversions FP32 to Posit8/16 and viceversa

As reported in Figure 2.2, this co-processor can be paired with a RISCV-V core that already has a floating-point unit (e.g. Ariane 64b RISC-V), without disrupting the already present pipeline. On the other hand, we can equip a RISCV-V core that does not have any floating-point support with this unit to enable ALU computation of posit numbers with the posit-to-fixed conversion modules. We analyzed the first use-case by equipping a CVA6 core with our PPU co-processor and synthesizing it targeting a Xilinx Genesys 2 FPGA obtaining a functional RISC-V core that could run a general-purpose Linux distribution.

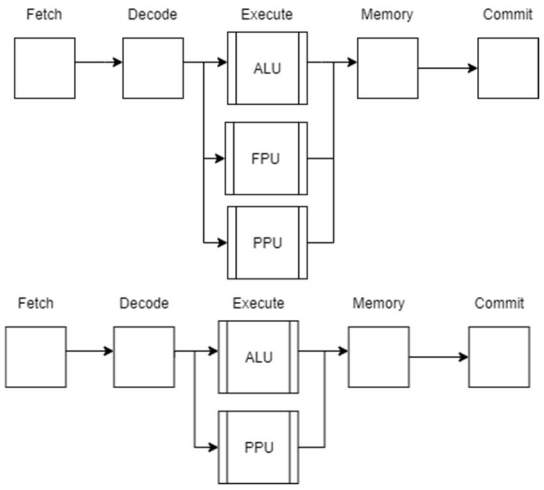


Figure 2.2: PPU possible integration modes within the RISC-V instruction set (with/without the FPU)

We therefore tested compression times of the overall system with the weights of a small LeNet-5 neural network, obtaining the result shown in the table 2.1.

	Time (s)	DNN size (bytes)	Compression
IEEE FP32	2,1	224894	-
posit⟨16, 0⟩	11.6	112874	**1.99**
posit⟨8, 0⟩	5.4	56864	**3.95**

Table 2.1: Compression performance of Posit vs FP32 for a DNN

Another activity has been then performed to use posit also to optimize computing efficiency. A typical workflow when computing linear algebra kernels in a vectorized environment (such that of RISC-V with vector extension such that developed in the European Processor Initiative by BSC):

1. Load operands inside Vector Processor Unit (VPU) registers from the memory hierarchy
2. Perform as much computations as needed exploiting vector instructions
3. Store the results back into the memory

Steps 1. and 3. typically involve the transfer of big chunks of data (e.g. a 512x512 binary32 matrix that need to be processed). These steps may take a significant amount of time in the overall process. The load and store operations are heavily influenced by the numerical format we adopt for the representation of the information. The core idea is to exploit one of the several numerical compression formats (e.g. Bfloat16, Posit16) instead of FP32 to reduce the amount of data transferred to the VPU registers and then decompress it directly inside the vector registers.

The process is the following:

We keep data stored in a compressed format. Data is loaded into the VPU registers without decompressing it beforehand. When needed, the data is decompressed on-demand exploiting VPU registers, see Figures 2.3 and 2.4.

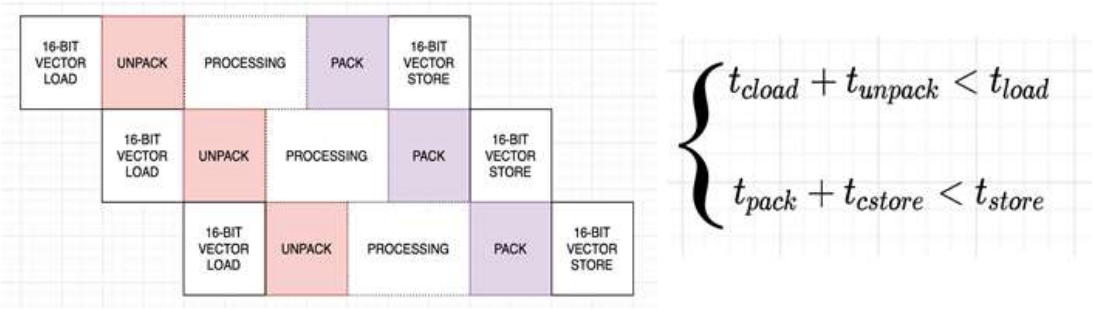


Figure 2.3: Data (un)packing and processing in a RISC-V with vector unit

This approach can greatly help with the overlapping of instructions in an out-of-order VPU by interleaving data-access with processing and de/compression phases.
However, in order to guarantee that the unpack and pack operations do not introduce a high latency in the overall computation, we need to make sure that the equations on the right hold.
We were able to satisfy such conditions using the Bfloat16 format thanks to this property: a right shift of 16 bits on binary32 number, gives us the correspondent Bfloat16 number (and vice versa). Being a single shift handled with very low latency by the VPU ALUs we were able to exploit this approach when using very large vector registers.

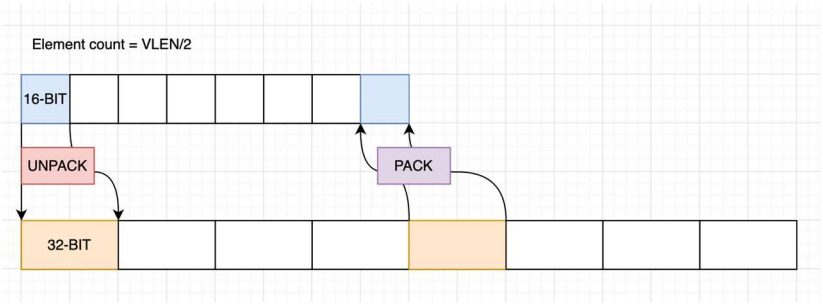


Figure 2.4: 16-bit compressed format

3. eXtreme Secure Crypto IP

This chapter focuses on the following cutting-edge cryptography functions and services:

- Homomorphic Encryption (HE) in the Internet of Things (IoT) context;
- eXtendable Output Functions (XOF) SHAKE 128/256 in Post-Quantum Cryptography (PQC).

These two topics will be discussed respectively in Sections **Errore. L'origine riferimento non è stata trovata.** and **Errore. L'origine riferimento non è stata trovata.**. This chapter presents the benchmark campaign carried out, and the achieved results on different CPU architectures of the aforementioned cryptographic functions. This analysis aims to define the main limits and bottlenecks of such algorithms and to start defining possible HW/SW strategies and specifications to improve performance in terms of computation time and energy efficiency.

3.1. Homomorphic Encryption: SEAL-Embedded Library for IoT devices

Homomorphic Encryption (HE) is a specialized type of encryption that allows specific computations on the encrypted data and generates a cyphertext that, once decrypted, matches the result of operations performed on the plaintext data. HE is nowadays considered a strong privacy-preserving solution that allows users to share data with clouds or any non-secure server. However, HE requires high computational resources and memory consumption, which limits its use in resource-constrained IoT devices. Different HE libraries exist, and the main ones are: Microsoft SEAL [1], PALISADE [2], and HELib [3]. Nevertheless, all of them are not specifically designed for resources-constrained devices. The SEAL-Embedded library [4] is the first HE library targeted for embedded devices that employ several optimizations to perform the encoding and encryption of data, featuring the CKKS HE scheme. An assessment of the SEAL-Embedded library has been carried out to evaluate its performance on different CPUs, and the results will be presented in the next section.

Benchmark on RISC-V CPUs

The source code of the SEAL-Embedded library can be found in [5]. Two different RISC-V processors have been selected for the benchmark campaign, and two different environments have been implemented on the FPGA Board Zynq UltraScale+ MPSoC ZCU106 equipped with the System-on-Chip (SoC) XCZU7EV-2FFVC1156. Figure 3-1 shows the proposed hardware systems running the benchmark. The selected RISC-V processors are:

- The 32-bit RISC-V RISCY, whose HDL code can be downloaded in [6]. The left side of Figure 3-1 shows the complete system implemented in the target FPGA which encompasses the RISCY CPU, 256KB of on-chip memory, and AXI4 peripherals (i.e. JTAG and serial UART interface).
- The 64-bit RISC-V CVA6, whose HDL code can be downloaded in [7]. The right side of Figure 3-1 shows the complete system implemented in the target FPGA which includes the CVA6 CPU, 512MB of memory (i.e. onboard DDR4), and AXI4 peripherals (i.e. JTAG and serial UART interface).

Both systems run at 100 MHz of frequency on the target FPGA.

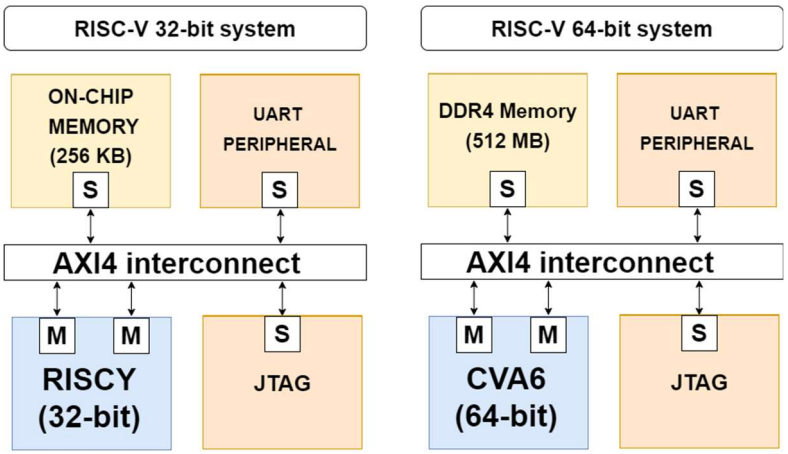


Figure 3-1: RISCV-based systems for benchmarking the SEAL-Embedded library.

The proposed systems allowed to find out the main bottleneck of the SEAL-Embedded library, which relies on the encryption function based on Ring Learning With Errors (RLWE) computation. Table 3-1 shows the benchmark results. The first column indicates the selected polynomial degree for the RLWE encryption. This parameter impacts both the message size (i.e. the size of the message blocks that must be provided to the encryption function, reported in the Msg size column) and the security strength of the RLWE encryption. Further details about the SEAL-Embedded library can be found in [4]. Despite the SEAL-Embedded being targeted for resource-constrained devices, it cannot be successfully executed on the RISCY CPU for Poly-Degree higher than 4096 (256KB of memory are not enough). In addition, the latency for the encryption process is extremely high: around 3 seconds are required to encrypt 8 KB with 4096 Poly-Degree.

Poly-Degree	Msg size	CVA6 (64-bit)	RISCY (32-bit)
1024	2048 B	17.19 ms	207.10 ms
2048	4096 B	37.09 ms	444.22 ms
4096	8192 B	273.80 ms	2806.43 ms
8192	16384 B	1184.19 ms	--
16384	32768 B	5861.02 ms	--

Table 3-1: Benchmark results for the encryption function of the SEAL-Embedded Library. Column 1 indicates the selected polynomial degree for the RLWE encryption, column 2 indicates the message size in Bytes, column 3 shows the results for the CVA6 processor and column 4 the results for the RISCY processor. Both CPUs run at 100 MHz.

Hardware accelerator specifications definition

Table 3-2 summarizes desired specifications of the hardware accelerator for the SEAL-Embedded library.

Target	Specification	Condition	Comments
Communication Interface	AXI4 - Memory Mapped (MM)	Must Have	Standard AXI4 Slave MM interface for the communication with CPU.
	AXI4 – Direct Memory Access (DMA)	Nice to Have	DMA for High-Throughput data exchange from/to memory. Depending on the needs could be implemented.
Supported parameters	Poly-degree for the RLWE symmetric encryption. Hardware support for all the parameters (i.e. from 1024 to 16384).	Must Have	
Latency for encryption	Desired latency for encryption of 8KB (with the polynomial degree of 4096) could be hundreds of milliseconds.	Must Have	Depending on the needs the performance can be improved respect to the defined specification.

Table 3-2: Specification definition of the hardware accelerator for the SEAL-Embedded library.

3.2. eXtendable Output Functions (XOF) SHAKE-128/256

An eXtendable Output Function (XOF) is a variable-length HASH function in which the length of the output can be chosen to meet the requirements of individual applications. The XOFs can be specialized to hash functions or used in a variety of other applications. The reference standard for the XOF is the NIST FIPS 202 [8], where two XOFs are specified: SHAKE-128 and SHAKE-256. Several NIST Post-Quantum finalists for both Key Encapsulation Mechanism (KEM) and Digital Signature (DS) adopt the XOF functions SHAKE 128/256: CRYSTALS-Kyber (KEM) and Dilithium (DS), Classic McEliece (KEM), NTRU (KEM), Saber (KEM) and Falcon (DS). In particular, in DS algorithms the hardware acceleration of XOFs becomes crucial since they are employed to HASH messages of any size. Some IoT applications, for instance Over-The-Air update, requires verifying the DS of large messages (e.g. up to Gigabytes) with low latency. Next section will show the benchmark results of the DS algorithms CRYSTALS-Dilithium and Falcon running on both RISC-V CVA6 and ARM-A53 CPUs, aiming to identify how the message size affects the computation time.

Performance evaluation in Post-Quantum Digital Signature Algorithms

The source code of the Crystals-Dilithium and Falcon algorithms can be downloaded at the NIST official page for the PQC competition:https://pq-crystals.org/, https://falcon-sign.info/. In this case, we selected the CPUs RISC-V CVA6 and ARM-A53 because they can be reasonably used for IoT applications. Two different environments have been implemented on the FPGA Board UltraScale+ MPSoC ZCU106 equipped with the System-on-Chip (SoC) XCZU7EV-2FFVC1156:

- A RISC-V CVA6-based system, the one reported on the right side of Figure 3-1. In this case, the entire system is implemented on the target FPGA at 100 MHz of frequency.
- An ARM-A53-based hard-core system running at 1.2 GHz of frequency. The processor is connected to 2 GB of DDR4 memory.

Table 3-3 reports the results for the DS verification function of CRYSTALS-Dilithium and Falcon algorithms with different message lengths (i.e. from 10KB to 100 MB) on the RISC-V CVA6 CPU, while Table 3-4 reports similar results on the ARM-A53 CPU (in this case the message length varies from 10KB to 1 GB).

Message length[byte]	VERIFICATION FUNCTION – RISC-V CVA6 processor			
	Dilithium-2	Dilithium-5	Falcon - 512	Falcon - 1024
10K	30,27 ms	69,21 ms	14,11 ms	16,91 ms
100K	104,85 ms	143,60 ms	83,99 ms	86,88 ms
1M	865,77 ms	904,37 ms	799,24 ms	802,12 ms
10M	8455,38 ms	8492,71 ms	7.933,16 ms	7.936,13 ms
100M	84351,33 ms	84375,76 ms	79.273,83 ms	79.275,83 ms

Table 3-3: Computation time for the DS verification function of CRYSTALS-Dilithium and Falcon algorithms on the RISC-V CVA6 CPU.

Message length [byte]	VERIFICATION FUNCTION – ARM-A53 processor			
	Dilithium-2	Dilithium-5	Falcon - 512	Falcon - 1024
10K	4,65 ms	10,37 ms	4,6 ms	6,26 ms
100K	13,80 ms	19,52 ms	33,08 ms	34,71 ms
1M	107,77 ms	113,49 ms	325,00 ms	326,63 ms
10M	1.045,22 ms	1.050,89 ms	3.236,75 ms	3.238,38 ms
100M	10.419,43 ms	10.424,82 ms	32.354,11 ms	32.355,73 ms
1G	104.161,53 ms	104.167,26 ms	323.527,44 ms	323.529,06 ms

Table 3-4: Computation time for the DS verification function of CRYSTALS-Dilithium and Falcon algorithms on the ARM-A53 CPU.

Hardware accelerator specifications definition

Table 3-5 summarizes desired specifications of the hardware accelerator for the SHAKE-128/256 functions.

Target	Specification	Condition	Comments
Communication Interface	AXI4 - Memory Mapped (MM)	Must Have	Standard AXI4 Slave MM interface for the communication with CPU.
	AXI4 – Direct Memory Access (DMA)	Nice to Have	DMA for High-Throughput data exchange from/to memory. Depending on the needs could be implemented.
Supported parameters	Support both SHAKE-128 and SHAKE-256 functions	Must Have	
Latency/throughput	To be further investigated	TBD	Depending on the needs, desired latency and throughput shall be identified. Data exchange via DMA can significantly improve performance.

Table 3-5: Specification definition of the hardware accelerator for the SHAKE-128/256 functions.

4. IPs for low-latency intra-node and inter-node communication links

The INFN Communication IPs implement a n-D Torus direct network for FPGA accelerators, allowing low-latency data transfer between processing tasks deployed on the same FPGA (IntraNode communication) and on different FPGAs (InterNode communication).

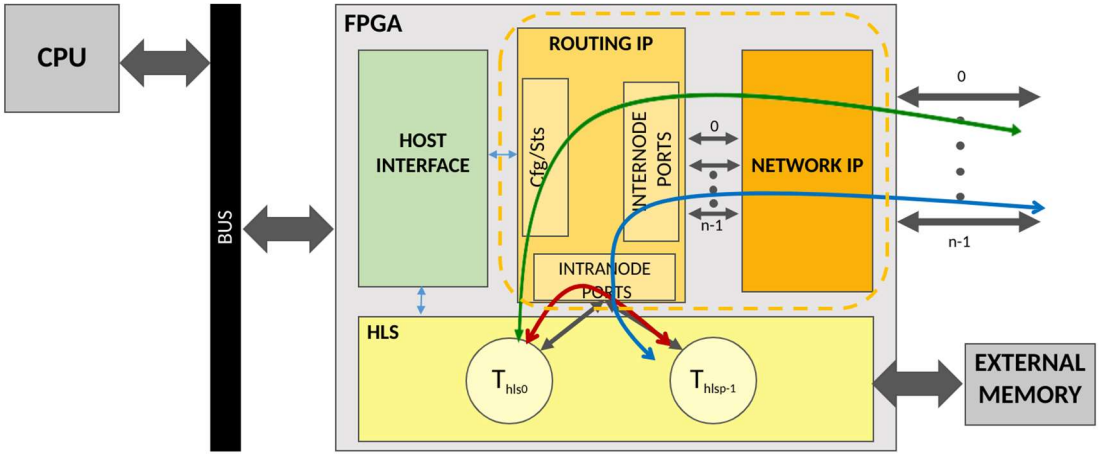


Figure 4.1 : Example of IntraNode (red) and InterNode (green and blue) data transfers between tasks

The hardware block structure, depicted in Figure 4.2, can be split into a **Network_IP** and a **Routing_IP**, described in more detail in the next sections.

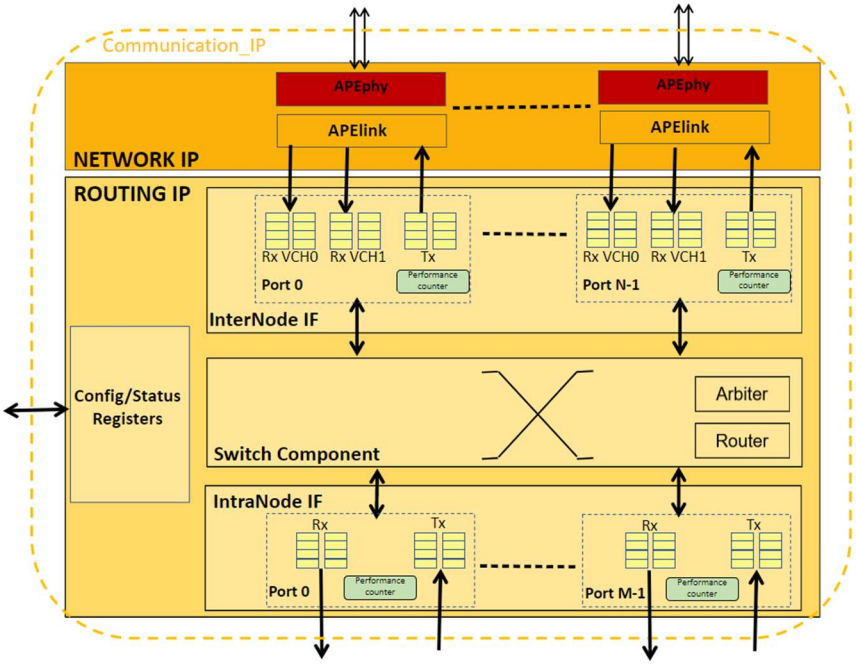


Figure 4.2: Architectural partition of Communication IPs

The INFN Communication IPs, developed in VHDL, will be implement as RTL-kernel in Vitis, a framework which allows to develop, debug, and optimize accelerated applications using standard programming languages for both software and hardware components.

In TEXTAROSSA project, target platforms are both Xilinx Alveo U200 and U280 cards, featuring the Xilinx UltraScale+ technology

Table 4.1 collects Alveo Boards features.

Card Component	U200	U250	U280
FPGA	UltraScale+ XCU200-2FSGD2104E	UltraScale+ XCU250-2LFIGD2104E	UltraScale+ XCU280-L2FSVH2892E
DDR4	64 gigabyte (GB) 4x DDR4 16 GB		32 gigabyte (GB) 2x DDR4 16 GB
	2400 mega-transfers per second (MT/s), on 64-bit ECC DIMM		
HBM	–	–	8 GB, 32-pseudo channels
PCIe	16-lane PCI Express		
	PCIe Integrated Endpoint block connectivity		
	Gen1, 2, or 3 up to x16		Gen1, 2, or 3 up to x16 Gen4 x8
Network Interface	2x QSFP28	2x QSFP28	2x QSFP28
I2C Bus	✓	✓	✓
Status LEDs	✓	✓	✓
Power Management	Power management with system management bus (SMBus) voltage, current, and temperature monitoring		
Electrical Design Power	65W PCIe slot functional with PCIe slot power only		
	140W PCIe slot functional with 110A max V_{CCINT} current PCIe slot power and 6-pin PCIe AUX power cable connected		
	215W PCIe slot functional with 160A max V_{CCINT} current PCIe slot power and 8-pin PCIe AUX power cable connected[1]		
Configuration Options	1 gigabit (Gb) Quad Serial Peripheral Interface (SPI) flash memory		
	Micro-USB JTAG configuration port		
UART	UART access through the USB port		

Table 4.1 – Features of the Alveo family boards

These cards provide a PCI Express interface to allow communication between the host processor and the network and are equipped with two 4-lane QSFP28 ports capable of 100 Gbps each.
QSFP+ ports available allow the connection, using point-to-point, bi-directional, full-duplex communication channels, of each board with its two neighbors in a 1-D torus network topology (a ring).

4.1. Routing IP

The **Routing_IP** defines the switching technique and routing algorithm, dynamically interconnecting all IP's ports and solving contentions for shared resources.
The transmission is packet-based, in the sense that Communication IP sends, receives, and routes packets with header (shown in figure 4.3), a variable size payload and a footer.

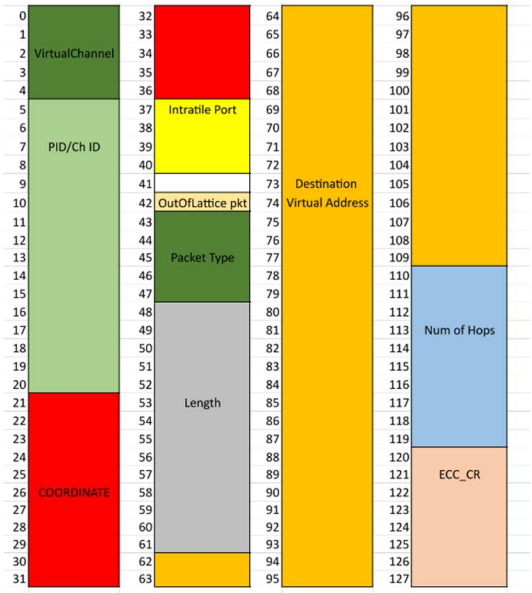


Figure 4.3: Packet's header format

The two sets of interfaces exposed, i.e. **IntraNode** and **InterNode**, are composed by a number of ports (M and N) that 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 specific FIFO.
For interNode communications the routing policy applied is the dimension-order one (DOR): it consists in reducing the coordinates offset between current and destination node to zero while routing the packet, considering one dimension at time in an inverse lexicographic order (e.g. ZYX).
The deadlock-avoidance of DOR routing is guaranteed by the implementation in the I**nterNode IF** of two virtual channels for each physical channel [9].
The employed switching technique — i.e., when and how messages are transferred — is Virtual Cut-Through (VCT) [10]: the router starts forwarding the packet as soon as the routing algorithm has picked a direction and the receiving buffer has enough space to store the full packet.

4.1.1 HLS Communication Adaptor

Task-side, input/output channels' interface is decoupled from the Routing IP one, so that the user doesn't have to care about the network protocol. A task should only implement a generic stream interface for each communication channel, based on the AXI4-Stream protocol, as follows:

```
void example_task(
        message_stream_t message_data_in[N_INPUT_CHANNELS],
        message_stream_t message_data_out[N_OUTPUT_CHANNELS]
        )
```

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 IPs: Aggregator and Dispatcher. 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.

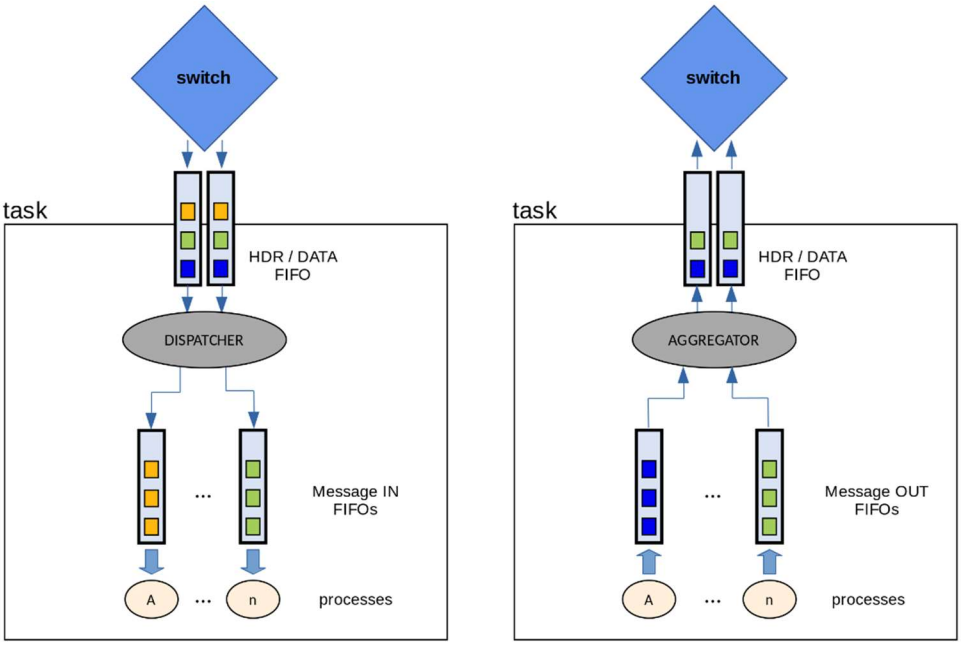


Figure 4.4: Schematic view of incoming and outgoing communication flow

4.2. Network IP

The **Network_IP** block, is in charge of managing data flow over the serial links between FPGAs.
In the first Communication IPs release, to transfer data between each node with its neighbors we will use Xilinx Aurora 64B/66B cores for the serialization of the messages over the cable, and INFN APElink IP [11] to guarantee reliable communication, performing error detection and correction.
In the final version we will implement also 10/25 Gbps and 100 Gbps Ethernet.

5. IP for fast task scheduling

The objective of task 2.5 is the development of a HW IP, compatible with a RISC-V core available in BSC, for fast task management compliant at tool chain-level with OmpSs approach that in TEXTAROSSA is then linked to the Vitis HLS tool to ensure that reconfigurable units are integrated in the tool chain. The use of tasks to interface with the accelerators will allow the runtime to integrate both tasks exploiting the "Kahn channel" abstraction and standard OmpSs (OpenMP) tasks which will improve the scope of applications targeted by the project. Preliminary results show that using a HW manager to schedule tasks significantly improves the performance of the application. This IP will be a service IP since it interfaces with the cores and other IPs in the project.

5.1 Basic design and functionality

This document describes the basic design and functionality of the IP for fast task scheduling (from now on FTS or Fast Task Scheduler). A first diagram is shown in Figure 5.1.1.

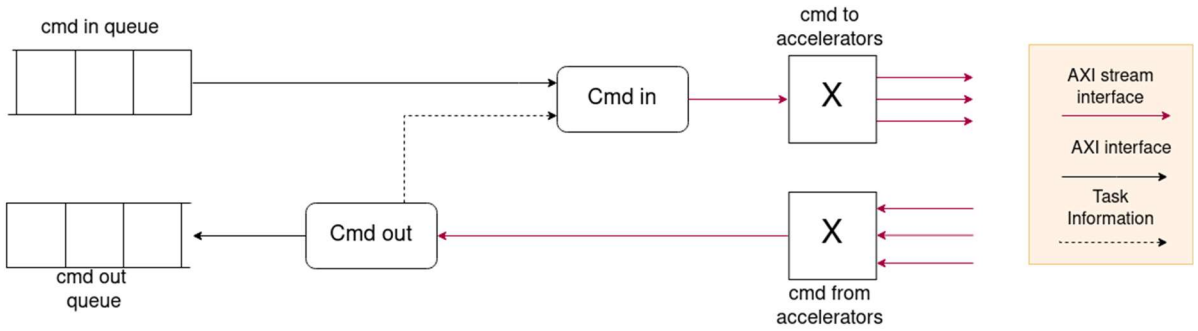


Figure 5.1.1 IP for fast task scheduling diagram (FTS).

As it can be seen in Figure 5.1.1 the system is composed of two command queues, one for input coming from the CPU/exterior of the FPGA (“cmd in queue”) and another going to the CPU/exterior of the FPGA (“cmd out queue”), two control modules (“Cmd in” and “Cmd out”) and two interconnection multiplexers/demultiplexers. Tasks are sent from the host CPU to the fast task scheduler by using commands that are temporarily stored in the “cmd in queue”. These commands are processed in order by the “Cmd in” module and, depending on the accelerators availability, are sent to the appropriate accelerator. Commands are sent through the “cmd to accelerators” demultiplexer through an AXI stream interface, and only when accelerators are available (ready) in order to avoid interface contention and starvation.
Once the task has been processed by the corresponding accelerator, it informs the FTS through an output AXI stream interface that is multiplexed to reach the “Cmd out” module with a “Finished Task” command. “Finished Task” command is expected to be processed in very few cycles (tens of cycles at most). Therefore, although some contention can be expected when several accelerators finish at the same time submitting this command, no significant performance drop is expected in this case.
The “Cmd out” module is in charge of processing the “Finished Task” packet by forwarding it with the adequate format to the “cmd out queue” and to notify the “Cmd in” module about the new ready state of the accelerator in order for the FTS to forward a new task to it.

Tasks and Periodic Tasks

In order to accomplish fast task scheduling, the prototype IP is going to be able to schedule both tasks and periodic tasks [12].

Periodic systems (i.e., recurrent workloads) are a common workload in industrial environments and real-time applications. Those workloads use the task concept to define the different activities that must be executed periodically (after some amount of time). Thereby, task-based parallel programming models are great candidates to support recurrent workloads. We propose extending the current syntax of task-based parallel programming models to define the main recurrent task parameters. Therefore, modeling recurrent workloads can be accomplished efficiently in terms of code lines, and with all parallel capabilities of baseline programming models. Also, we propose using the reconfigurable heterogeneous platforms to efficiently manage these recurrent workloads. These platforms will provide an efficient management of recurrent tasks, keeping the great programmability provided by the parallel programming models.
Periodic tasks are defined as tasks that repeat themselves a number of times. This repetition can be set a number of times (as soon as possible) or at a certain time interval defined by the user (provided that the task is executed in less time than the defined trigger interval). These kinds of tasks have demonstrated to be very powerful to address the problems of industrial environments [5.1]. The support for periodic tasks would also be incorporated in the programming model support in task 4.2. The periodic tasks syntax include two clauses: period(N) and num_repetitions(K). An example for a periodic and a regular task definition is shown in Figure 5.1.2.

```
# pragma omp task inout ([10] array )
num_repetitions(reps) period (1000000)
void periodic_task ( int * array , const int reps );
# pragma omp task inout ([10] array )
void regular_task ( int * array );
int main (...) {
        int array [10];
        regular_task ( array );
        periodic _task ( array , 100);
        regular_task ( array );
        # pragma omp taskwait
}
```

Figure 5.1.2 Periodic and regular task definition example.

The main function calls the regular task, then the periodic task, and finally the regular, creating a chain of three task instances due to its data dependence. The periodic task has the num_repetitions clause, which defines that the task body will be executed reps times (in this case, the argument value is 100), and the period clause, which defines that the task will begin the execution every 1 second (1000000 microseconds). The first regular task becomes ready when created as its data dependences are satisfied. In contrast, the other two are postponed. The recurrent task is postponed until the first regular task finishes, and the second regular task is postponed until the 100 repetitions of the recurrent task have been executed.

5.2 Queues and Commands information

The following section describes the structure of memories used to communicate the Host (usually using libxtasks) with the FTS.

Command in and Command out queues

Each queue has 1024 elements (uint64_t type) and it is divided into 16 subqueues of 64 elements. Each subqueue corresponds to one accelerator, starting from accelerator 0 (positions [0,63]) to accelerator 15 (positions [960,1023]) as shown in Figure 5.2.1.

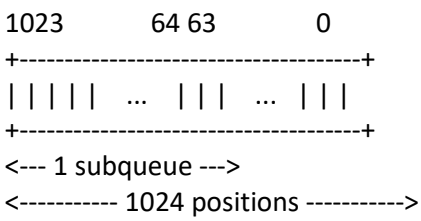


Figure 5.2.1 Command in and Command out queues.

Each command uses a dynamic number of slots in the queue. The number of slots depends on the command. The odd command codes make the accelerator become busy (no further commands will be sent to the accelerator until it returns the command out response) and the even command codes do not. The information is structured as shown in Figure 5.2.2.

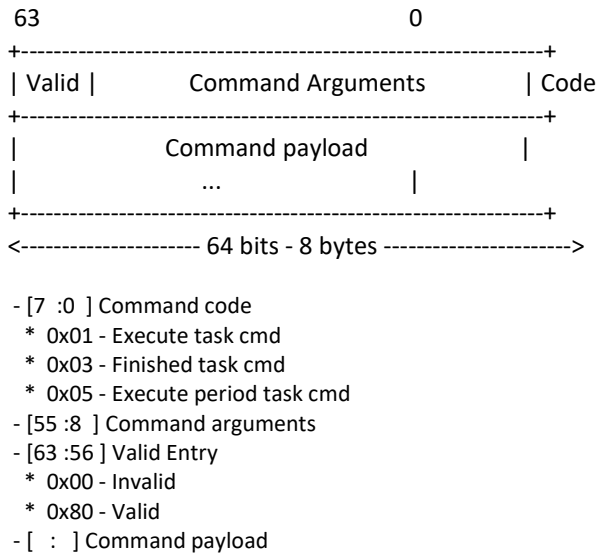
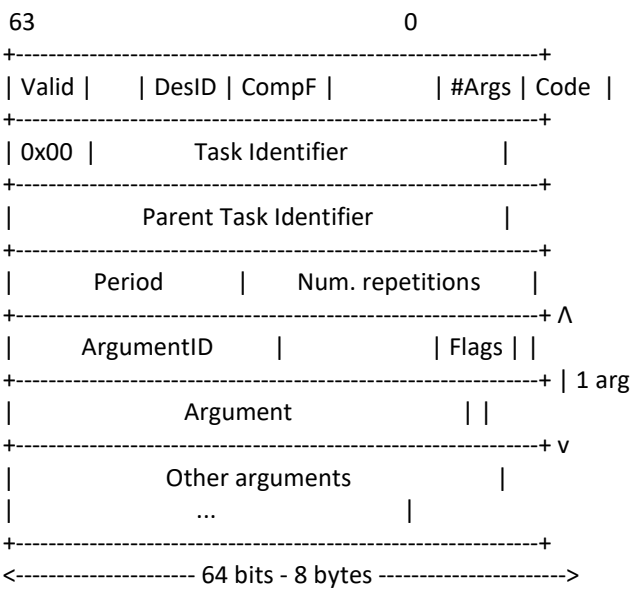


Figure 5.2.2 Commands format.

As it can be seen the commands use the first position to indicate the command and the next positions as a payload (actual command information).

Commands format

We have defined three initial commands in the FTS, a Execute task command, a Finished task command notification and a Execute periodic task command. The Execute task and Execute periodic task commands follow the structure shown in Figure 5.2.3.



- [7 :0] Command code
  * 0x01 - Execute task cmd
  * 0x05 - Execute periodic task cmd
- [15 :8] Number of arguments
- [31 :16]
- [39 :32] Compute flag
  * 0x00 - Compute disabled
  * 0x01 - Compute enabled
- [47 :40] Destination ID where the accelerator will send the 'complete' signal
  * 0x1F - Processing System (PS)
- [55 :48]
- [63 :56] Valid Entry
  * 0x00 - Invalid
  * 0x80 - Valid
- [119:64] Task identifier
- [127:120] 0x00 constant. This field is used to identify task commands created externally
- [191:128] Parent Task identifier. This field is ignored by the FTS and the accelerators, it is maintained to match the format of the internal command queue and the format expected by the accelerators.
- [223:192] Number of times that task body will be executed (Execute periodic task cmd only)
- [255:224] Time (us) between task body launches (Execute periodic task cmd only)

Each argument is:
- [7 :0] Flags
  * 0x00 - BRAM
  * 0x01 - Private
  * 0x02 - Global
  * 0x10 - Enable input copy to wrapper BRAM
  * 0x20 - Enable output copy from wrapper BRAM
  * bit7 is internally used by cmd In module to store whether the input copy has been optimized or not.
- [31 :8]
- [63 :32] Argument ID
- [127:64] Argument value

Figure 5.2.3 Execute task and Execute periodic task commands format.

Finally, Figure 5.2.4 shows the Finished task command format. As it can be seen this command is simpler as it only sends the task identifier information. This information is used by the task creator

(runtime running in the host) to keep track of possible dependencies and could also be used by the FTS to identify the accelerator that has finished.

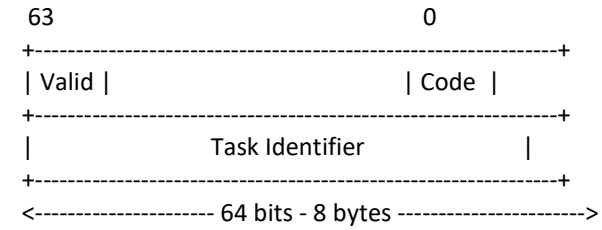


Figure 5.2.4 Finished task command format.

5.3 Data reuse optimizations

In order to reduce the amount of data to be accessed by the accelerators, FTS includes an automatic detection of data reuse among tasks that are waiting in the command in queue. FTS can detect if two consecutive tasks in the command in queue are re-using the same input data. In that case, it can deactivate the copy flag of the argument to be reused of the second task before this task is issued to the accelerator. Therefore, the accelerator will only need to copy data that is not already in its local memory.

Figure 5.3.1 shows two execution traces of an application when data reuse is deactivated or activated. This application has been annotated with FPGA tasks using OmpSs@FPGA and has been cross-compiled for and executed on a Zynq 7000 family board (two Cortex-a9 at 666MHz + FPGA running at 100Mhz) as a proof of concept. This is using two different versions of the FTS mentioned above to coordinate the two accelerators (IPs) and the software running on the two cores in the SMP, showing that FTS IP will be able to interface with cores and other IPs in the project. Horizontal lines in the trace show the states (different colors) on the SMP threads (two lines on the top of each execution trace) and two accelerators (two lines on the bottom of each execution trace), along the time.

Task execution in an accelerator has, with no optimizations, three states (colors): copy in data (first starting with a flag - olive green), kernel execution (second - dark olive green) and the last one copy out data (brown).

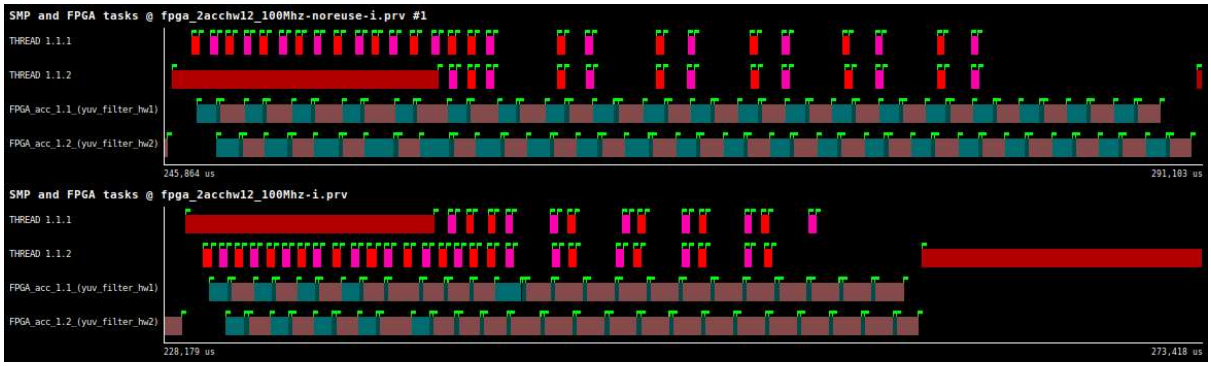


Figure 5.3.1 Execution traces of an application using two accelerators. Execution traces show the same time duration. Top: FTS has data reuse deactivated. Bottom: FTS has data reuse activated for tasks in the Command in queue.

On the execution trace on the top of the figure we can see that there are always three states (different states start and end with flags), which is not ideal. Those tasks are always re-using the same input vector but the accelerator is not conscious about this fact and is copying the input vector all the time. On the other hand, the execution trace on the bottom shows the performance achieved once FTS includes the data reuse feature. In this case FTS can automatically detect data to be reused in an accelerator and help to almost remove all input copies modifying the argument copy flags of the task descriptions.

Note however that there are still tasks in the execution trace on the bottom of Figure 5.3.1 that have three states and no data reuse is detected. This happens because originally data reuse detection among the tasks is only performed among tasks waiting in the Command In queue and no detection is done between a task being executed and tasks that arrive later to the Command In queue. This situation may happen in several applications: a task is submitted (first one) by the runtime, it immediately starts execution in the accelerator, and then, another task is submitted by the runtime. Since the first one has already started, no detection can be done between Command In queues commands. This can be solved by taking care of the task being executed in the accelerator at that moment. FTS has been improved to detect and be able to catch this situation. This can be seen in Figure 5.3.2. The execution trace on the bottom incorporates that feature. Only the first task of all tasks being executed has to copy the data, significantly improving the first FTS version (no data reuse) and allowing first task executing-second task in Command In queue data reuse. The extra-copy seen in the execution trace of the Figure 5.3.2 (bottom) is because the accelerator was completely empty when a new task was submitted. The overall performance improvement with data reuse can be significant as it can be seen in Figure 5.3.2.

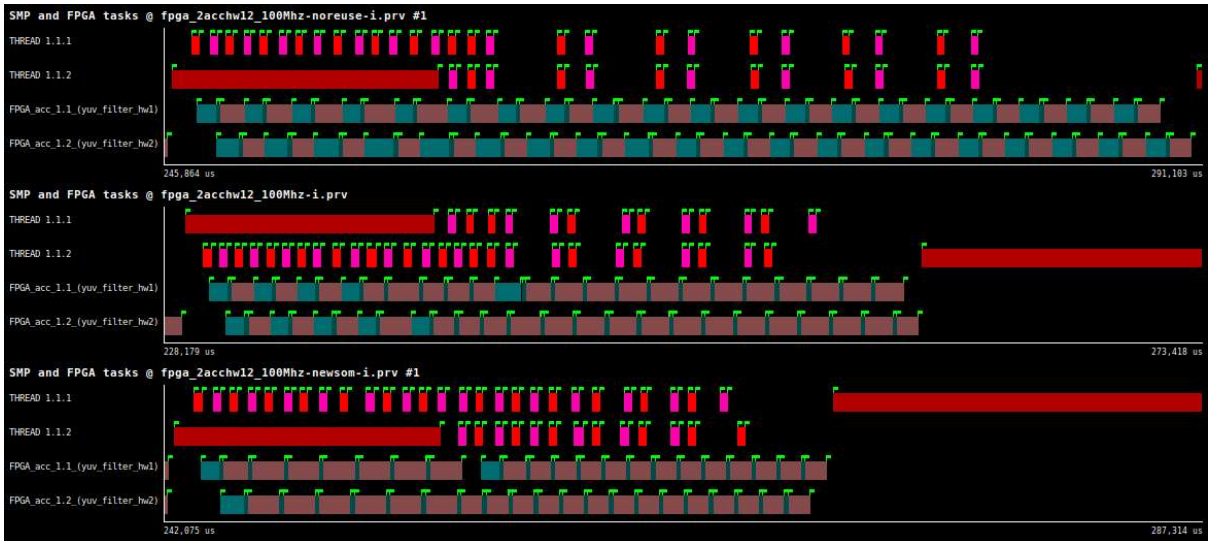


Figure 5.3.2 Execution traces of an application using two accelerators. Execution traces show the same time duration. Top: FTS has data reuse deactivated. Middle: FTS has data reuse activated for tasks in the Command in queue. Bottom: FTS has data reuse activated for tasks in the Command In queue and tasks being executed.

6. Conclusions

This document reports the activities done by Textarossa [13] partners CINI (UNIPISA), INFN and BSC with reference to the consolidated specifications of accelerator IPs in WP2 and preliminary design and synthesis results, manly in FPGA technology.

CINI UNIPI in Section 2 has presented the specification and preliminary design results of accelerators with mixed-precision (using fixed, float and posit formats), for data compression and for efficient computation of DNN (Deep Neural Network).

CINI UNIPI in Section 3 has presented the specification and preliminary design results of accelerators for innovative security services based on Post Quantum Cryptographic (PQC) techniques taking into account the NIST standardization effort. The proposed accelerator will be useful also for homomorphic encryption where SW libraries from Microsoft have been proposed already in the market.

The specifications and preliminary design results for IP cores used in low-latency inter-node and intra-node communications and for fast task scheduling are also presented in Sections 4 and 5.

The proposed IPs are interesting, also in view of synergies between Textarossa and the other initiatives like EPI and the European Pilot, since all the proposed accelerators can be integrated with RISC-V computing cores like the RISC-V 64b Ariane IP and the RISC-V with support of the Vector extension in the EPAC (European Processor Accelerator).

7. References

General

[1] Chen, H., Laine, K., & Player, R. (2017). Simple Encrypted Arithmetic Library - SEAL v2.1. Financial Cryptography Workshops.
[2] PALISADE. https://gitlab.com/palisade. New Jersey Institute of Technology(NJIT).
[3] Halevi, S., & Shoup, V. (2020). HElib design principles. Tech. Rep.
[4] Natarajan, D., & Dai, W. (2021). SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(3), 756–779. https://doi.org/10.46586/tches.v2021.i3.756-779.
[5] https://github.com/microsoft/SEAL-Embedded.
[6] https://github.com/pulp-platform/pulpino.
[7] https://github.com/openhwgroup/cva6.
[8] Dworkin, M. (2015), SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD.
[9] Deadlock-free message routing in multiprocessor interconnection. Seitz, W. J. Dally, and C. L. 1987. 5, s.l. : Computers, IEEE Transactions on, 1987, Vol. C.36, p. 547–553.
[10] A New Computer Communication Switching Technique. P. Kermani and L. Kleinrock, Virtual Cut-Through: Comput. Networks 3 (1979) 267.
[11] APEnet+ 34 Gbps Data Transmission System and Custom Transmission Logic. Ammendola, R et al JINST 8 C12022
[12] Bosch J. Vidal M., Filgueras A., Jiménez-González D., Álvarez C., Martorell X., Ayguadé E.: Task-Based Programming Models for Heterogeneous Recurrent Workloads. ARC 2021: 108-122
[13] E. Commission, “Grant Agreement 671668 - TEXTAROSSA: exploring Manycore Architectures for Next-GeneratiOn HPC systems.” 2021.