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

WP4 Toolchain for heterogeneous multi-node HPC platforms

D4.7 HLS Flow

http://textarossa.eu

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

TEXTAROSSA

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

Grant Agreement No.: 956831

Deliverable: D4.7 HLS Flow

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

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

Deliverable No	D4.7
WP No:	WP4
WP Leader:	INRIA
Due date:	M30
Delivery date:	30/11/2023

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

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

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	02/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP4 Toolchain for heterogeneous multi-node HPC platforms

Deliverable number:	D4.7					
Deliverable title:	HLS Flow					
Due date:	M30					
Actual submission date:	02/12/2023					
Editor:	Paolo Palazzari					
Authors:	Giovanni Agosta, JuanMiguel de Haro, Antonio Filgueras, Carlos Alvarez, Daniel Jiménez, Marco Danelutto, Alessandro Lonardo, Cristian Rossi, Michele Martinelli, Paolo Palazzari, Daniele Cattaneo, Bruno Morelli					
Work package:	WP4					
Dissemination Level:	Public					
No. pages:	95					
Authorized (date):						
Responsible person:	Paolo Palazzari					
Status:	Plan	Draft	Working	Final	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	25/10/2023	P.Palazzari	Draft structure
0.2	03/11/2023	JM. De Haro, C. Alvarez, A. Filgueras, D.Jimenez	Section HLS in Task-based model
0.3	06/11/2023	A. Lonardo, C. Rossi, M. Martinelli	Section “The APEIRON framework for High Level Programming of Streaming Applications on Multi-FPGA Systems"
0.4	15/11/2023	G. Agosta, D. Cattaneo, B. Morelli	Section “TAFFO in the HLS flow”
1.0	20/11/2023	P. Palazzari	Inserted introducing and concluding paragraphs, global check on the doc structure

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Project Technical Committee	28 nov. 2023
Checked by Task Leader	G. Agosta, P. Palazzari	29 nov. 2023

Checked by WP Leader	Berenger Bramas	30 nov. 2023
Checked by Project Coordinator	Massimo Celino	1 dec. 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

Partner Report Activity 8

List of Authors 8

List of Acronyms 8

Executive Summary 11

1 Introduction 12

2 HLS in the Streaming Model 13

2.1 Introduction 13

2.3 **FFfpga** implementation 17

2.4 **FFfpga** usage 20

2.4.1 Single shot applications 20

2.4.2 Single kernel in streaming application 21

2.4.3 Multiple kernels on the same FPGA 22

2.4.4 Exploiting multiple FPGAs 23

2.5 Kernels 24

2.6 Applications 25

2.7 References 26

3 HLS in the Task-based model 27

3.1 Introduction 27

3.1.1 Relationship with the project objectives and strategic goals 28

3.2 Compilation process 29

3.2.1 OmpSs@FPGA Clang/LLVM compiler 30

3.2.2 Accelerator Integration Tool (AIT) 41

3.3 Custom IPs for the FPGA Hardware Runtime 43

3.4 Accelerator Placement, Interleave and Priorities 44

3.5 HBM memory access 45

3.6 FPGA Instrumentation Support 45

3.7 Custom IPs for inter-communication in FPGA-based clusters 47

3.7.1 OMPIF runtime architecture 47

3.7.2 CloudFPGA cluster 48

3.7.3 MEEP cluster 48

3.8 Power Sampling Support 50

3.9 References 51

4 The APEIRON framework for High Level Programming of Streaming Applications on Multi-FPGA Systems 53

4.1 Introduction 53
4.2 APEIRON Building Blocks 56
4.2.1 Communication IP 56
4.2.2 Software Stack 57
4.3 Latency and Bandwidth of Communications between HLS Tasks in the APEIRON Framework 59
4.4 Example of APEIRON Application 60
4.5 References 66
5 TAFFO in the HLS flow 68
5.1 Architecture of TAFFO 68
5.2 Structure of the Software 69
5.3 Integration with AMD Vitis 70
5.4 Using TAFFO with Vitis 71
5.5 Modifying an application 72
5.6 References 73
6 Efficient use of the HLS flow in the streaming model 75
6.1 Fine-grain parallelism in the streaming model 75
6.1.1 Output depends only on input data 75
6.1.2 Output depends on previous outputs data 76
6.2 Vitis implementation of the fine-grain parallelism 77
6.2.1 Output depending only on input data 77
6.2.2 Output depending on previous output data 80
6.3 The medium-grain parallelism 81
6.4 Vitis HLS implementation of the medium-grain pipeline 83
6.5 The coarse-grain pipeline 85
6.6 Vitis HLS implementation of the coarse-grain pipeline 88
6.7 Efficient burst access to DDR banks 92
6.8 Using HDL simulation to look inside FPGA parallelism 93
7 Conclusions 95

Partner Report Activity

Tasks 4.1, 4.2, 4.3, 4.4	Adapting and exploiting the Vitis HLS flow in the various SW technologies developed in TEXTAROSSA
Github address	The software developed and the benchmarks carried out during the activity are downloadable at Github at the address: https://gitlab-tex.enea.it
Technology	FastFlow, OmpSs, TAFFO, APEIRON, Vitis HLS
Technical development	The technical development has been performed by BSC, ENEA, INFN, POLIMI, UNIPI

List of Authors

ENEA	Paolo Palazzari
POLIMI	Giovanni Agosta
BSC	Carlos Alvarez
UNIPI	Marco Danelutto
INFN	Alessandro Lonardo

List of Acronyms

AaaS	Accelerator as Service
ABI	Application Binary Interfaces
ACP	Acceleration Coherency Port
ADC	Analog Digital Converter
AFS	Andrew File System
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMG	Algebraic MultiGrid
AMS	Analog Mixed Signal
API	Application Program Interface
ASIC	Application Specific Integrating Circuit
AXI	Advanced eXtensible Interface (Xilinx IP)
BMC	Baseboard Management Controller
C/R	Checkpointing/Restart
CAPI	Common Application Programmer's Interface
CCXI	Cache Coherent Interconnect for Accelerators
CDU	Cooling Distribution Unit
CLB	Configurable Logic Block
CNN	Convolution Neural Network
CP	Common Platform
CPU	Central Processing Unit
CRDB	Co-design Recommended Daughter Board
CU	Compute Unit
CXL	Compute Express Link
DAG	Data-flow Graphs
DC	Direct Cooling
DCL	Data Control Language
DDR	Double Data Rate memory
DIMMs	Dual In-line Memory Modules
DL	Deep Learning
DLC	Direct Liquid Cooling
DPSNN	Distributed Polychronous Spiking Neural
DSL	Domain Specific Language
DSP	Digital Signal Processing
DTPC	Direct Two-Phase Cooling
ECC	Elliptic Curve Cryptography
ECC	Error correction code memory

EDP	Energy Delay Product
ED2P	Energy Delay Square Product
EDS	Embedded Design Suite
EFLOPS	Exa Floating Point Operations per Second
EPAC	EPI Accelerator
EPI	European Processor Initiative
ETS	Energy-To-Solution
FMM	Fast Multipole Method
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSB	Front Side Bus
FT	Fault Tolerance
FTI	Fault Tolerance Interface
GCD	Graphics Compute Die
GIC	Generic Interrupt Controller
gpm	Gallons per minute
GPU	Graphics Processing Unit
GRM	Global Resource Manager
HBA	Host Bus Adapter
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HEP	High Energy Physics
HLL	High-Level Language
HLS	High Level Synthesis
HPC	High Performance Computing
HPDA	High Performance Data Analytics
HPL	High Performance Linpack
HPS	Hard Processor System
HSS	High Speed Serial
HTC	High Throughput Computing
IoT	Internet of Things
IOB	Input/Output block
IP	Intellectual Property
IPMI	Intelligent Platform Management Interface
IR	Iterative Refinement
KPN	Kahn Process Network
L2HN	L2 cache Coherence Home Node
LCM	Last Common Multiple
LE	Logic Element
LRM	Local Resource Manager
MCM	Muti-Chip-Module
MD	Molecular Dynamic
MDS/T	Metadata Server/Target
ML	Machine Learning
MMU	Memory Management Unit
MPI	Message Passing Interface
MPPA	Multi-Purpose Processing Array
MPSoC	Multi-Processor System on Chip
NFIR	Non-linear Finite Impulse Response
NN	Neural Network
NoC	Network on Chip
NVIC	Nested Vectored Interrupt Controller
NVMe	Non-Volatile Memory
OAM	OCP Accelerator Module
OCP	Open Compute Project
QPI	Quick Path Interconnect
OSS/T	Object Storage Servers /Target
PCG	Preconditioned Conjugate Gradient
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PFLOPS	Peta Floating Point Operations per Second
PGMRES	Preconditioned Generalized Minimal Residual
PoC	Proof of Concept
PSU	Power Suppy Unit
PU	Processing Unit

PUE	Power Usage Effectiveness
QCD	Quantum ChromoDynamic
QFDB	Quad FPGA Daughter Board
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
RMS	Resources Management Systems
RoCE	RDMA over Converged Ethernet
RTC	Real Time Clock
RTRM	Runtime Resource Manager
SAS/SATA	Serial Attached SCSI/Serial ATA
SLR	Super Logic Region of FPGA
SoC	System on Chip
SpMM	Sparse Matrix-sparse Matrix
SpMP	Sparse Matrix Power
SpMV	Sparse Matrix-Vector
SSD	Solid State Drive
STX	Stencil/Tensor accelerator
TCO	Total Cost of Ownership
TDAQ	Trigger & Data Acquisition
TDP	Thermal Design Power
TOPS	Tera Operations per Second
TTS	Time-To-Solution
ULFM	User-level Fault Mitigation
ULL	Ultra-Low Latency
UVM	Unified Virtual Memory
VCS	Virtual Compute Server
VM	Virtual Machine
VPU	Vector Processing Unit
VRP	Variable Precision co-processor

Executive Summary

This document details the utilization and integration of the Vitis High-Level Synthesis (HLS) flow within tools developed in Work Package 4, specifically focusing on the Toolchain for heterogeneous multi-node HPC platforms.

Within the context of streaming models, Vitis HLS kernels have been encapsulated as specialized FastFlow nodes and seamlessly integrated into the FastFlow streaming parallel programming framework. Section 2 of this document provides comprehensive insights into the extensions made to FastFlow and the methodology for interfacing with FPGA kernels.

Similarly, addressing task models, the ompSS@FPGA programming framework has undergone enhancements in this project. New Intellectual Properties (IPs) have been defined to support Fast Task Scheduling, optimize memory accesses, facilitate inter-FPGA communication, and manage kernel execution and power monitoring. Section 3 elucidates the proper instantiation, utilization, and interaction with these IPs.

The APEIRON hardware/software framework supporting the execution of distributed HLS applications has been successfully developed. Section 4 of this document, following a thorough review of the architecture of the APEIRON communication infrastructure, outlines the recommended instantiation and utilization of this framework. The document specifically addresses the API developed for managing both intra and inter-FPGA communications.

TAFFO is a precision tuning framework for automatically exploiting the performance/accuracy trade-off. To avoid expensive dynamic analyses, TAFFO leverages programmer annotations which encapsulate domain knowledge about the conditions under which the software being optimized will run. As a result, TAFFO is easy to use and provides state-of-the-art optimization efficacy in a variety of hardware configurations and application domains. Since TAFFO is based on the LLVM framework, it is easily extensible to HLS workflows. We successfully integrated TAFFO with the Vitis HLS tool, allowing mixed precision workflows that also include a FPGA-based kernel component. This is described in section 5.

Given the pivotal role of High-Level Synthesis (HLS) coding in achieving high performance with the software (SW) tools outlined in this document, a dedicated section, section 6, has been incorporated. This section aims to elucidate guidelines identified during the project, providing insights into the efficient implementation of various classes of algorithms onto FPGA. The focus is on leveraging different granularities of available parallelism.

The software tools developed in the TEXTAROSSA project are highlighted for their significant contributions towards achieving the project's objectives. Specifically:

- Section 2.1 outlines how the streaming models have contributed to realizing the project's objectives.
- In section 3.1.1, a detailed account is provided regarding the contributions made by the Task-based models.
- Section 4.1 sheds light on the contributions stemming from the APEIRON fast communication HW/SW environment.

1 Introduction

After undergoing significant development, High-Level Synthesis (HLS) for FPGA devices has now evolved into a well-established methodology, owing to the advancements achieved by leading HLS tools developed by prominent FPGA hardware and software manufacturers. Notable among these industrial solutions are Vitis by AMD, OneAPI by Intel, and CatapultC by Menthor Graphics.

This deliverable delineates the integration of the Vitis HLS flow, identified as the most prevalent and mature HLS solution, within the programming frameworks developed as part of TEXTAROSSA and tailored for the IDV-E High-Performance Computing (HPC) prototype node. It elucidates how Vitis HLS has been harnessed within the project - configuring it to support novel IPs developed in TEXTAROSSA (such as the Task Scheduler, FPGA monitoring, intra- and inter-FPGA communication), augmenting existing parallel programming frameworks (FastFlow and OmpSS), and incorporating methodologies to facilitate variable precision in computations.

The AMD U280 FPGA board serves as the reference accelerator board selected for integration into the IDV-E node prototype.

The document is structured as follows:

- Section 2 elaborates on the extensions made to the FastFlow streaming framework.
- Section 3 details the extension of ompSS achieved through management of specific new IPs.
- Section 4 presents the APEIRON framework: the communication IP, its software stack, and the associated APIs.
- Section 5 describes the outcomes attained by applying the TAFFO library/tool to programs implemented on the FPGA accelerator.
- Section 6 offers algorithmic templates for leveraging HLS programming effectively within the streaming model.

Throughout the description of the developed software tools, their contributions to realizing the objectives of the TEXTAROSSA project are underscored.

.

2 HLS in the Streaming Model

FastFlow is a structured parallel programming framework developed to target shared memory architectures [Aldinucci et al, 2017], possibly equipped with (GPU) accelerators, and recently extended to target also distributed architectures (COW/NOW) [Tonci et al, 2023]. In the past, FastFlow has been extended to provide minimal support to manage execution of simple accelerator kernels on FPGA leveraging an implementation like the one used for GPUs and targeting FPGAs exploiting a specialized run time developed at the University of Darmstad [Korinth et al, 2025], in the framework of the EU Funded F7 project REPARA. Within TextaRossa, a full integration of FPGA kernels developed through the Vitis toolchain has been developer that supports the orchestration of single and multiple kernels on single or multiple FPGA boards within standard FastFlow streaming applications [Danelutto et al. 2023]. The FastFlow extension manages all the details related to the execution of existing, pre-compiled Vitis Kernels on the Alveo FPGA boards used in the TextaRossa reference nodes, leaving the FastFlow application programmer a minimal burden to specify a distinct template to instantiate the accelerated application components—with respect to the standard one used to instantiate non-accelerated application components—and to declare the list of actual parameters to be supplied/received to/from the FPGA kernel. The FastFlow implementation supporting FPGA Vitis kernel execution (**FFfpga**, from now on) greatly improves the programmability of FPGA accelerated streaming applications and reduces to time-to-production in the accelerated application design and development.

2.1 Introduction

In streaming applications, parallelism is usually exploited by processing different “stages” of the overall computation on different, parallel, execution engines, such that the throughput of the application is improved, with respect to sequential implementation. Typical patterns used in streaming (I.e. stream parallel) applications is the pipeline one. In a pipeline, input data items appearing onto the application streaming input—the tasks—are processed by applying different, independent computations—the stages—such that each stage works on the partial results produced by the previous stage and directs results to the next stage. Eventually, the last pipeline stage directs its results to the streaming application output stream. Assuming uniform stages (that is stages that all consume a comparable amount of time to execute) the throughput of the pipeline achieves a speedup proportional to the number of stages with respect to the sequential execution. In the case of “slow” stages, these stages may be replicated to achieve the aforementioned speedup with some problems to be faced in case the slow stage maintains some kind of internal state. This to avoid the slow stage *de facto* limits the overall performance of the streaming application, that cannot exceed a throughput bound to the inverse of the time taken to execute the slower pipeline component.

Hardware acceleration is of paramount importance in streaming applications as it can be exploited to speed up the execution of particular—slow—stages thus contributing to improvement of the overall throughput.

The approach taken in the design of **FFfpga** is based on pragmatic separation of concerns:

- FPGA programmers are left as the only and main responsible for the efficient implementation of the kernels to be executed on the accelerator board. The kind and level of knowledge necessary to implement efficient hardware accelerated kernels is completely different from the one usually in the background of the streaming application programmers.
- All the offloading activities related to the usage of pre-designed FPGA kernels within a FastFlow application have been left in the responsibility of the FastFlow (**FFfpga**) runtime designers, such that streaming application programmers are only asked to identify the kernel suitable to support acceleration in the streaming application and to use the FPGA node FastFlow templates provided by **FFfpga** to instantiate the kernel(s) in the streaming computation code. The **FFfpga** runtime will take care of all the activities related to kernel execution, such as kernel configuration, data memory transfers to/from the FPGA, synchronization, double buffering, etc.

Consequently, the main contribution of **FFfpga** may be summarized as:

- Full integration of existing kernels in C++ FastFlow streaming applications, provided the kernel activation parameters and the kernel results datatypes and sizes are known.
- Minimal effort is required for the application programmer to integrate one or more kernel accelerated computation stages.
- Freedom in the orchestration of accelerated stages within complex streaming applications, by supporting seamless kernel chaining, replicas and placement on different boards connected to the same node.

In the TextaRossa project perspective, **FFfpga** contributes to different objectives:

- It improves **energy efficiency**: FPGA accelerated stages may compute the very same results of functionally equivalent CPU bound stages using smaller amounts of time. Considering the energy needed to keep the FPGA board alive is usually lower than the one needed to keep the CPU alive, this results in a double power saving: smaller device power consumption *and* shorter execution time. By spending shorter execution times than CPU and with the efficient orchestration of accelerated stages within the general, larger streaming application, **FFfpga** also contributes to achieve **sustained application performance**. (project objective 1 and 3)
- As **FFfpga** relieves application programmers from the burden of managing—host side—the execution of the FPGA kernels, **FFfpga** contributes to the **seamless integration of reconfigurable accelerators**. (project objective 4)
- **FFfpga** runs on either Intel based or Arm based CPUs, equipped with Alveo FPGA boards with the associated Vitis development environment, thus fully complying the **integrated development platform** objective. (project objective 6)
- Last but not least, **FFfpga** targets a domain (streaming applications) traditionally targeting different hw/sw combinations (big data, distributed programming frameworks) thus opening perspectives to the usage of more classical HPC frameworks for this kind of applications. (project strategic goal 3, **opening of new usage domains**).

2.2 **FFfpga** in a nutshell

FFfpga comes as a software package to be used along with the current version of FastFlow (https://github.com/fastflow/fastflow.git). FastFlow applications are built of stages subclassing a system provided **ff_node_t<Tin,Tout>** template. The template basically requires the programmer to implement a "service" method **Tout * svc(Tin * task)** implementing the computation to be provided by the stage. FastFlow stages may be orchestrated in pipelines—by using the **ff_Pipe<Tin, Tout>** template—stages may be replicated—by using the **ff_Farm<Tin, Tout>** template—and in addition several other "parallel orchestrator patters" are provided supporting other kind of parallel computations, including a "parallel for" and a "divide and conquer" pattern. Whatever provided as a parallel pattern in FastFlow is an **ff_node_t**, thus supporting full compositionality of patterns. In FastFlow, we can program pipelines with parallel for data parallel stages, farms with divide and conquer worker, etc. Sequential code is always wrapped in plain **ff_node_t** stages. The whole framework is implemented using C++ and supports (only) C++ sequential code, although with minimal effort programmers may use precompiled modules developed with other programming languages by properly wrapping them into minimal C++ **ff_node_t** nodes.

The key concept in **FFfpga** is the *wrapping node.* **FFfpga** provides a different template—the **FNodeTask**—that subclasses the **ff_node_t** and can therefore be used in any place where a normal **ff_node_t** stage may be used. The **FNodeTask** constructor accepts as parameters the device to be used (any of the FPGAs in the node where the **ff_node_t** wrapping node is run, with the associated bitstream file name where the existing, precompiled kernel must be taken) and the kernel name. By using the **FNodeTask** stage in a pipeline, the **FFfpga** run time takes care of all the steps needed to manage the offloading of **FNodeTask** input stream tasks to the FPGA and redirection of the FPGA kernel computed results to the **FNodeTask** output stream. All these activities are completely transparent to the streaming application programmer and include:

- Loading the bitstream on the selected FPGA (among the ones available at the node) if it has not already been loaded.
- Declaring the host side aligned buffers necessary to host kernel input parameters and output results.
- Managing the memory buffer movement in between host and device memory as needed, possibly exploiting double buffering to improve overall streaming throughput.
- Synchronizing execution of the different input tasks on the FPGA kernel(s).

Just to introduce the kind of coding required to exploit **FFfpga**, the effort required to the streaming application programmer to include an accelerated stage in a pipeline requires to specify the pipeline as depicted in Fig. 2.1.

Somewhere, programmer must detail the name of the bitstream implementing the kernel(s) he wants to use. In this declaration of the **FDevice** object, a second parameter with the id (0,1,2,…) of the board to be used may be exploited to declare which one of the FPGA attached to the node has to be used. In principle, the FPGA could also be different models, provided they support Vitis and provided the bitstream named in the first parameter has been correctly compiled for the specified FPGA. All experiments have been performed using two identical FPGA boards per node, so far.

In addition to the code in Fig. 2.1, the streaming application programmer must consider that the parameters provided to the offloading nodes must be organized as a **FTask**. The **FTask** is basically a container hosting different vectors with the addresses and sizes of the parameters processed by the FPGA kernels. Fig. 2.2 shows how an **FTask** may be declared and filled to be used with a kernel that consumes two vectors and produces one vector as a result. In the code, the first parameter of the add_input/add_output calls represents the address of the data structure, the second parameter represents the size of the data structure and the optional third parameter represents the bank to be used to host the data structure in the FPGA board HBM.

```
// declare a logical FFfpga device, by specifying the bitstream filename used and
// (optional, the id of the board to be used (if multiple boards are present)
FDevice device(bitstream, id);
…
// declare the pipeline
ff_pipeline p;

// add stages: stream generator, FPGA offloader, stream consumer stages
p.add_stage(new generator(n, m));
p.add_stage(new FNodeTask(device, kernel_names[0], chain));
p.add_stage(new drain());

…
// execute the pipeline
p.run_and_wait_end();
```

Fig. 2.1: Code snippet with an FPGA kernel offloading stage in FastFlow exploiting FFfpga.

```
FTask * task = new FTask();
task->add_input(a,  size_in_bytes, bank1);
task->add_input(b,  size_in_bytes, bank2;
task->add_output(c, size_in_bytes, bank3);
task->add_scalar(s, sizeof(int));
```

Fig. 2.2: Code snippet declaring an FPGA offloader node task

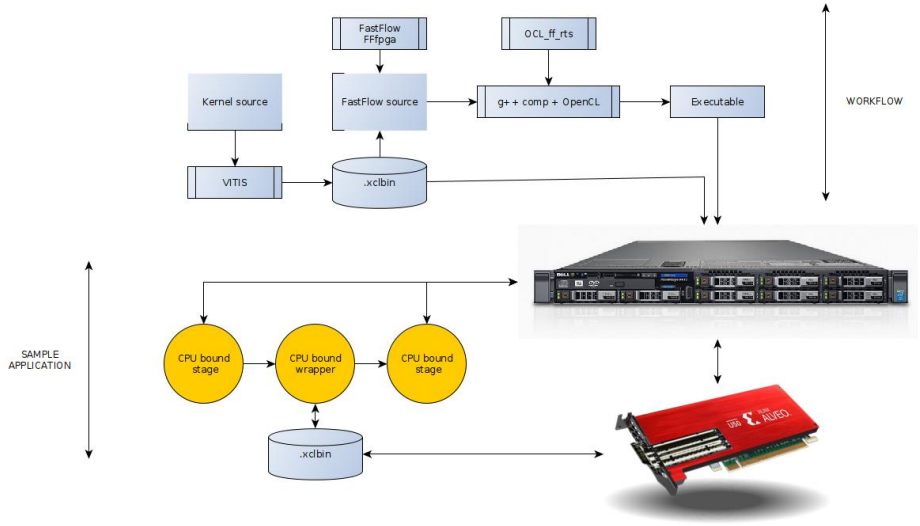


Fig. 2.3: FFfpga application development workflow

The workflow needed to compile the streaming application of Fig. 2.1 is very simple (see Fig. 2.3):

- The application must be compiled using a standard C++ compiler implementing C++17, at least.
- Being FastFlow implemented as a header only library and due to the heavy usage of advanced meta programming techniques, when compiling the –O3 flag is mandatory. If not used, performance of FastFlow application execution may suffer a notable inefficiency (up to 10x slowdown w.r.t. the –O3 version)
- The application can be run from command line, once the bitstream of the kernel(s) used is located in the exact place used to specify the bitstream in the **FDevice** call.

All the interactions of **FFfpga** with the FPGA are managed through OpenCL, and therefore, mandatory prerequisites consist of the availability of OpenCL toolchain and of the XILINX XRT.

2.3 **FFfpga** implementation

FFfpga implementation consists of different small components that are used to build and operate the special **ff_node_t FNodeTask**s. To detail the structure of the **FFfgpa** implementation we need first to dissect the basics of an ff_node_t.

An **ff_node_t** is an object encapsulating a parallel activity in a FastFlow program. The base node just encapsulates a thread computing some kind of sequential (business logic) code. Proper template classes of FastFlow provide more complex, internally parallel, parallel activities such as a pipeline or a parallel for as an f**f_node_t**, thus fully supporting complex parallel compositions in the development of an application.

An **ff_node_t<Tin,Tout>** provides three fundamental methods:

- A service method **svc** with signature **Tout * svc(Tin * t)** implements the actual computation of the parallel activity modelled by the **ff_node_t**. In the base case, it wraps the business logic code, possibly reused from existing application(s), in complex patterns, such as the pipe, orchestrates the parallel execution of the component **svc** methods according to the parallel pattern implemented.
- An initialization method **svc_init** with signature **int svc_init(void)** is executed when the parallel activity implemented by the node is actually started.
- A finalization method **svc_end** with signature **void svc_end(void)** that is executed at the termination of the **ff_node_t** orchestrated by the termination of the topmost **ff_node_t** in the application code.

This said, a **FNodeTask** wraps the execution of an existing, pre-compiled FPGA Vitis kernel by exploiting the **ff_node_t** structure:

- The **svc_init** method is used to host all the OpenCL code needed to set up the management of the FPGA (kernels). We look for the platform, for the device and set up all the handlers needed to manage the execution of a kernel on the FGPA. Finally, the bitstream hosting the kernel is loaded on the FPGA so that, at the end of this method, the device is ready to start kernel executions. It is worth pointing out that in case the setup of the OpenCL management structures has been already performed (e.g. being one of the multiple **FNodeTask** nodes offloading to the same FPGA), this work is performed just once at the first one of the **FNodeTask**s.
- The **svc** service method takes care of the execution of the stream of offloading targeting the FGPA kernel(s). To achieve proper overlapping of the data offloading with the computation of the kernels, this method actually spawns two service threads, such that the data buffering may actually be exploited. From outside, however, the method looks like as a “normal” **ff_node_t svc** method, that is as a synchronous call starting the business logic code execution (the FPGA kernel, in this case) and returning the results of the computation immediately after the computation has ended.
- The **svc_end** method is used to finalize the interaction of the OpenCL subsystem with the FPGA, to clean up the data structures allocated for the management of the FPGA, etc.

In addition to the implementation of the **ff_node_t** subclass **FNodeTask**, we included in the **FFfpga** implementation some service code. Part of the code dedicated to the execution of the OpenCL calls is embedded in dedicated files, and the code needed to represent the tasks directed to the FPGA kernels is included in a specific class the **FTask** class.

The **FTask** class provides methods to declare the structure and sizes of the FPGA kernels parameters (see Fig. 2.4). There are methods to declare input and output parameters (size, location and HBM memory bank allocation), methods to declare scalars to be passed as input parameters to the kernel (output scalars must be represented as vectors with length equal to 1). There are also a couple of methods used to enqueue the tasks (that is finalize buffer copies host to device, if necessary and starting the kernel execution) and to await the termination of the computation of the task. These latter methods are used in the two threads exploited in the **FNodeTask** implementation to support communication/computation overlapping through double buffering: the first thread **eqeue**s the task and the second one a**wait**s termination of the task computation.

```
class FTask
{
private:
        ...
public:
   std::vector<FTaskElement> inputs;
   std::vector<FTaskElement> outputs;
   std::vector<FTaskElement> scalars;

   events_t write_events;
   events_t kernel_events;
   events_t read_events;
   FTask() {}
     void add_input(size_t size) {…}
     void add_input(size_t size, int bank_id) {…}
     void add_input(void * ptr, size_t size, int bank_id) {…}
     void add_output(size_t size) {…}
     void add_output(size_t size, int bank_id) {…}
     void add_output(void * ptr, size_t size, int bank_id) {…}
     void add_scalar(void * ptr, size_t size) {…}
     void wait() {…}
     void enqueue(FDevice & device, cl::Kernel & kernel,
                  cl::CommandQueue & queue, FTask * previousTask = nullptr,
                  bool flush = false) {…}
```

Fig. 2.4: **FTask** structure

The **FNodeTask** class is minimal (see Fig. 2.5): it just provides a constructor creating two threads, and **enqueu**er and a **wait**er thread in pipeline to start kernel execution on the task and await task termination.

```
class FNodeTask : public ff::ff_pipeline {

public:

   FNodeTask(FDevice & device, std::string kernel_name, bool chain_tasks = false)

   {

      add_stage(new FNodeTaskInternals::FNodeEnqueuer(device, kernel_name, chain_tasks));

      add_stage(new FNodeTaskInternals::FNodeReader());

      cleanup_nodes();

   }

};
```

Fig. 2.5: **FNodeTask** structure.

2.4 **FFfpga** usage

FFfpga may be exploited to implement different kinds of offloading strategies:

- Exploit a single kernel in a single shot application (no streaming)
- Exploit a single kernel in a streaming application (repeated execution of the very same kernel)
- Exploit multiple kernels on the same FPGA to sustain throughput of a single streaming node (pipeline stage replicas)
- Exploit multiple FPGAs or single FPGA on the same node with any application setting requiring to exploit multiple kernels in the same or in different computation stages.

We are currently still working to finalize the possibility to manage different kernels on different FPGA attached to different nodes with the kernels communicating either using the on-board Ethernet IP or the interconnection support provided by INFN within the TextaRossa project.

2.4.1 Single shot applications

In this case, we assume that FastFLow is used to run some data parallel application that does not require the execution of streaming tasks. Therefore, we assume that FPGA kernels are used only to accelerate a single computation, possibly using different replicas of the kernels to operate on different partitions of the input data set. In this case, any technique aimed at overlapping computation to communication may result useless. A different, simpler implementation of the **FNodeTask** is available that simply offloads task data to the FPGA, starts kernel, awaits kernel termination, and moves back the computed results to the host memory, sequentially. Therefore, the cost paid for the offloading sums communication to computation times and the only improvement is in the faster computation of the function in the FPGA kernel with respect to the slower computation of the very same function on the host CPU cores.

2.4.2 Single kernel in streaming application

When stream parallelism is exploited, stages in the stream parallel computation may be accelerated offloading to the FPGA kernel a stream of task computations. In order to accelerate a stage computation offloading the task to an FPGA kernel, the user has simply to include an **FNodeTask** instead of a plain (or composition of) **ff_node_t**. In this case, the multithreaded implementation of **FNodeTask** deals with the proper management of the overlapping between parameter copies host to device—as well as result copies device to host—with the streaming computation of the FPGA kernels.

On the application program side, the application programmer has to provide an **FTask** hosting all the data needed to compute the kernel, declare the **FDevice** data structure and then use the **FNodeTask** in the proper pipeline (or farm) declaration, as depicted in Fig. 2.6.

```
FDevice device(bitstream);
  ...
  FTask * task = new FTask(); // sample 2 vec in, 1 vec out kernel
  task->add_input(a,  size_in_bytes, 0); // parameters in different HBM banks
  task->add_input(b,  size_in_bytes, 1); // to support parallel read
  task->add_output(c, size_in_bytes, 2); // and write
  task->add_scalar(s, sizeof(int));
  …
   ff_pipeline p; // declaration of the pipeline
   ...
   p.add_stage(new FNodeTask(device, "kernel_name",  chain)); // offloader stage
   …
  p.cleanup_nodes();
  p.run_and_wait_end(); // pipeline execution (separate declaration and execution in Fastlow)
```

Fig. 2.6: Offloading in streaming applications: excerpt of pipeline application code

The code snippet in the picture, *de facto* uses distinct host threads to offload kernel parameters and start execution of the kernel, and to await kernel termination. Therefore, all the steps needed to execute multiple tasks from the stage input stream happen to be executed in pipeline on host *and* on the FPGA.

Bandwidth of offloading is clearly upper bound by the DDR or HBM memory bandwidth *and* by the PCIe bus bandwidth, as expected.

2.4.3 Multiple kernels on the same FPGA

In case the PCIe and DDR memory bandwidth suit the execution of multiple kernels on the FPGA, different kernels or copies of the same kernel may be targeted in different pipeline stages of farm workers from within the FastFlow streaming application. Provided the proper **FTask** and **FDevice** objects have been declared and initialized, then multiple kernels hosted in the uploaded **.xclbin** file may be used as offloading pipeline stages as show in the excerpt of code in Fig. 2.7.

```
ff_pipeline p;

p.add_stage(new generator(n, m));

p.add_stage(new FNodeTask(device, "kernel_stage1" , chain));

p.add_stage(new middle(n, 1));

p.add_stage(new FNodeTask(device, "kernel_stage2", chain));

p.add_stage(new drain());
```

Fig. 2.7: Multiple kernels to offload different pipeline stage execution on FPGA.

In this case, results of the first offloaded tasks, computed on the FPGA by the **kernel_stage1** kernel, are directed to the second computing stage of the pipeline and then the results of this CPU hosted staged are used as input to the **kernel_stage2** kernel.

In this case, FFfpga manages to overlap the computations of **kernel_stage1** and **kernel_stage2** relative to different tasks on the FPGA, as expected. Fig. 2.8 shows the timings achieved in the execution of a pipeline such as the one in Fig. 2.7.

Fig. 2.8: Execution of two pipeline stages offloaded to distinct kernel on the same FPGA.

To further increase pipeline throughput, we can use replicas of the slower stages that in FastFlow are modelled as farm nodes. A three-stage pipeline with a “bottleneck” second stage may be turned into a pipeline of farm composition by “farming out” the stage, as outlined in Fig. 2.9.

```
ff_pipeline p;              // version with no replicas on second stage
p.add_stage(stage1);
p.add_stage(stage2);
p.add_stage(stage3);
p.run_and_wait_end()
ff_Farm ParStage2(stage2,n);    // version with n replicas on 2nd stage
ff_pipeline p;
p.add_stage(stage1);
p.add_stage(ParStage2);
p.add_stage(stage3);
p.run_and_wait_end()
```

Fig. 2.9: Replicas in FastFlow pipeline bottleneck stage

In case the bottleneck stage may be offloaded to an FPGA kernel, and provided that multiple kernels may be actually hosted on the FPGA and have eventually been included in the **.xclbin**, the parallelization of the pipeline stage as outlined in Fig. 2.9 can be applied by simply using **FNodeTask** stage2 components. Each one of the instances will target a different kernel instance (e.g. some **kernel_stage2:n**) and therefore the declaration of the FastFlow farm object is slightly "wordier". First a **ff_farm<> f;** has to be declared, then a **vector<ff_node_t*> w;** has to be used to host as many **FNodeTask *** as the number of replicas required, by issuing multiple **w.push_back(new FNodeTask(device, "kernel_stage2:0"));** and then setting the farm workers with the vector with some **f.add_workers(w);**.

The kind of behaviour achieved is the one depicted in Fig. 2.10, outlining the overlapping of execution of the four (in this particular case) kernel workers of the farm.

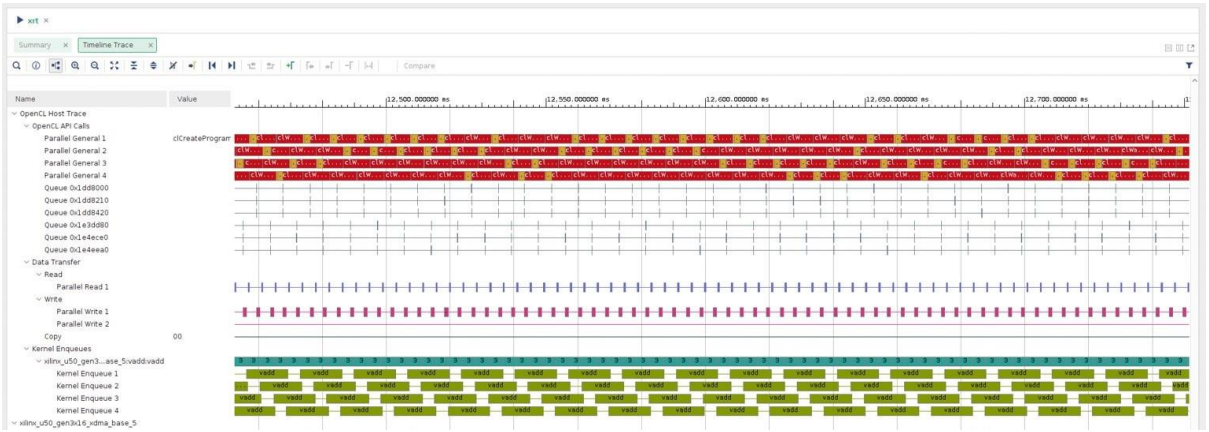


Fig. 2.10: Replica execution in farmed pipeline stage.

2.4.4 Exploiting multiple FPGAs

In case a node supports more than a single FPGA (e.g. in TextaRossa reference node A and B, where we have two U280 Alveo cards), any of the solutions outlined in the previous sections may be adopted simply changing the **FDevice** declarations to direct kernel management to the first or the second FPGA. Taking again as an example the pipeline application of Fig. 2.7, what we need to do to use the first kernel on one FPGA and the second kernel stage on the second FPGA is to declare two **FDevice** objects and the use the two objects to instantiate the two **FNodeTask** first parameters. In particular, the **FDevice** constructor takes a second, optional parameter, which is the index of the device in the device list "discovered" using the OpenCL calls. As such, by declaring **FDevice dev1(bitstream1,0); FDevice dev2(bitstream2,1);** and subsequently using **p.add_stage(new FNodeTask(dev1, "kernel_stage1")); p.add_stage(new FNodeTask(dev2, "kernel_stage2"));** the execution of the two offloading stages will be managed on the two boards available.

2.5 Kernels

In principle, kernels supported by the current FFfpga implementation are those taking any number of parameters from FPGA global memory (HBM) and delivering results in buffers in the very same memory using buffers initialized and managed from within the OpenCL interface. The typical structure of such kernels is a **#pragma HLS dataflow** unit with a module fetching input data from the OpenCL allocated HBM buffers and delivering them to an **hls_stream**, a module actually computing results from the input **hls_stream** and delivering such results to another **hls_stream**, and finally a module storing in the OpenCL allocated output HBM output buffers the items appearing on the computing module output stream (see Fig. 2.11, outlining the typical kernel code structure).

```
void krnl_stage1(uint32_t* in1, uint32_t* in2, uint32_t* out, int size) {

#pragma HLS INTERFACE m_axi port = in1 bundle = gmem0

#pragma HLS INTERFACE m_axi port = in2 bundle = gmem1

#pragma HLS INTERFACE m_axi port = out bundle = gmem0

   static hls::stream<uint32_t> in1_stream("input_stream_1");

   static hls::stream<uint32_t> in2_stream("input_stream_2");

   static hls::stream<uint32_t> out_stream("output_stream");

#pragma HLS dataflow

  // dataflow pragma instruct compiler to run following three APIs in parallel

   load_input(in1, in1_stream, size);

   load_input(in2, in2_stream, size);

   compute_add(in1_stream, in2_stream, out_stream, size);

   store_result(out, out_stream, size);

}
```

Fig. 2.11: Typical structure of an **FFfpga** kernel.

Nothing prevents a more complex internal structure of the kernels used with FFfpga. Indeed, the **FTask** implementation supports the declaration of buffers only used FPGA side, such that these buffers may be used to have different modules implemented on the same FPGA (kernel) to share partial results besides using **hls_stream**s as outlined in the kernel structure of Fig. 2.11.

2.6 Applications

So far, we have developed different toy examples and applications exploiting the FFfpga implementation. UNITO (third party of the CINI beneficiary in the TextaRossa project) developed a CNN compute kernel targeting Alveo FPGAs implementing a classifier for MNIST. The model was developed and trained using TensorFlow, then converted to C via HLS4ML tool and finally compiled with Vitis. An FFfpga application was then used to offload tasks to the kernel. Synthesis of the kernel achieved 100Mhz on an Alveo U50 and the application demonstrated the functionality of the full FFfpga implementation. Fig. 2.12 outlines the internal structure of the CNN kernel exploited via FFfpga.

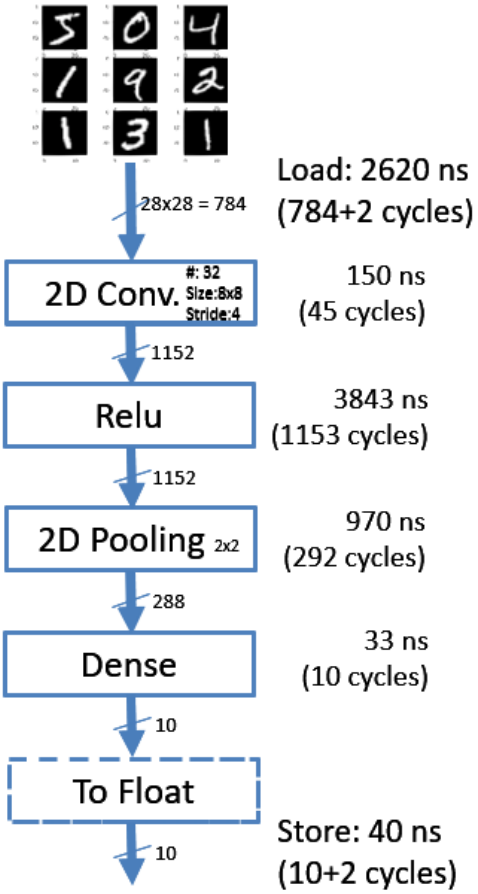


Fig. 2.12: structure of the CNN kernel

UNIPI (third party of the CINI beneficiary of the TextaRossa project) in cooperation with ENEA developed several simple streaming applications exploiting kernels processing images and compressing data in different shapes and flavours, that have been used to exercise, debug, and tune the **FFfpga** implementation. As an example, the kernel used to implement data compressor consists of more than 600 lines of code, but eventually has the very same structure as Fig. 2.11. The kernel is built of two distinct modules (see Fig. 2.13) in pipeline with the inter module communications managed by using an **hls_stream.**

```
extern "C" {
void DeflateKernel(hls::stream<io_stream_16B>  &strm_in,
                   hls::stream<io_stream_32B> &strm_out, int input_size)
{
#pragma HLS INTERFACE axis port=strm_out
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE s_axilite port=input_size
#pragma HLS INTERFACE s_axilite port=return
   static hls::stream<io_stream_48B_tuser> LZ77Enc2Huffman_stream("LZ77Enc2Huffman_stream");
#pragma HLS dataflow
   LZ77_Encoder(strm_in, LZ77Enc2Huffman_stream, input_size);
   huffman_encoder(LZ77Enc2Huffman_stream,strm_out);
}
}
```

Fig. 2.13: Compressor kernel module

2.7 References

[Aldinucci et al, 2017] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati “FastFlow: High-Level and Efficient Streaming on Multicores” in Programming Multicore and Many-Core Computing Systems book Chapter 13, pp 261-280, 2017. doi: 10.1002/9781119332015.ch13

[Tonci et al, 2023] N. Tonci, M. Torquati, G. Mencagli, M. Danelutto. “Distributed-memory FastFlow Building Blocks”, International Journal of Parallel Programming (IJPP), Springer, HLPP 2022 Special Issue, 2023, DOI: 10.1007/s10766-022-00750-5

[Danelutto et al. 2023] M. Danelutto, G. mencagli, A. Ottimo, F. Iannone, P. Palazzari, FastFlow targeting FPGAs. Euromicro Conference on Parallel Distributed and Network processing (PDP), IEEE Press, pp. 104-108, 2023

[Korinth et al, 2025] Korinth, J., Chevallerie, D. de la, and Koch, A. ThreadPoolComposer – An Open-Source FPGA Toolchain for Software Developers. In *Second International Workshop on FPGAs Software Programmers (FSP)*. 2015

3 HLS in the Task-based model

OmpSs@FPGA as a programming model, is an extension to the OmpSs-2 task-based programming model [OmpSs-2 2023] for C/C++ High Performing Computing applications [Haro 2021]. In this document, we refer to the programming model just as OmpSs. Every OmpSs program starts with one thread executing the main function, and function calls annotated with task pragmas are transformed by the compiler to asynchronous calls to the Nanos6 runtime. This runtime analyses dependencies (specified in the pragma) and schedules tasks that are ready for execution (i.e. no memory collisions). With OmpSs@FPGA, these tasks can be scheduled to hardware accelerators. To do that, the FPGA task code is processed by a High-Level Synthesis Tool (HLS) and transformed to a hardware IP implemented in the bitstream. OmpSs@FPGA is also a framework, a collection of software tools and hardware IPs that automatically create a bitstream and a software binary from the original source.

3.1 Introduction

OmpSs@FPGA framework support aims to create a free, open fra mework that makes programming Xilinx FPGAs easy. The framework leverages Vitis HLS and LLVM compilers to allow a single-source C/C++ program annotated with pragmas to be compiled and executed on a heterogeneous system composed of a CPU and an AMD FPGA. Task-based programming provides great code portability across different FPGAs and, at the same time, provides good performance results. It also allows FPGA tracing to improve code behaviour analysis. Finally, new features have been added to OmpSs@FPGA programming model to allow FPGA-based cluster execution of OmpSs programs.

The main contributions under the point of view of implementation are the following:

- LLVM/Clang compilation support to extract each of the programmer tasks to be accelerated (FPGA tasks) and creates and builds its corresponding top function (wrapper) fully compatible with new versions of Vivado and Vitis HLS. On the other hand, the same support generates the CPU binary. Each wrapper has the necessary ports and connections to synchronize the FPGA task execution with the hardware runtime, perform the communication between the host memory and the FPGA tasks memory, and execute the FPGA task of the programmer. The set of wrappers generated (one per FPGA task) is automatically passed to the Accelerator Integration Tool (also develop in this project) to generate the bitstream, transparently to the user.
- Accelerator Integration Tool (AIT) allows to create the Vitis/Vivado projects and the bitstream, transparently to the programmer, by just annotating functions of a sequential program with OmpSs@FPGA pragmas. AIT calls Vitis/Vivado tools to generate the HLS and Design projects, while dealing with several optimizations specified at user level both at compile time and configuration time, including low level information about the SLR placement of the accelerators.
- Custom IPs for the FPGA Hardware Runtime to allow all the synchronizations between FPGA and Software Runtimes and management of the FPGA and SMP tasks executions.
- Accelerator Placement techniques, high level guided by the programmer.
- Interleaving and Priority Custom IPs to avoid memory conflicts among FPGA tasks.
- FPGA instrumentation support.
- Software runtime support of FPGA tasks.
- Power sampling support.

- Support and Custom IPs for inter-communication in a FPGA-based clusters.

In Deliverable 4.6 we explain how the programming model works and how FPGA tasks are scheduled, including nested FPGA tasks. In this deliverable we detail the compilation process and custom IPs that allow inter-communication, integrating and making compliant the OmpSs@FPGA model with the Vitis HLS flow.

3.1.1 Relationship with the project objectives and strategic goals

Task based programming models such as OmpSs provide a good opportunity to abstract the underlying hardware complexity, so implementation effort is kept low while maintaining good levels of performance. The contributions described in this section are related to the project objectives:

- Objective 1 - Energy efficiency. Executing in FPGA has been demonstrated to be competitive with other computing platforms in terms of energy efficiency. In addition of providing the support to executing on the IDV-E platform, the OmpSs task-based model is integrated with power measurement tools included in the HLS flow to be able to further control and improve the energy spent when executing in the platform.
- Objective 2 - Sustained application performance. As explained in the next sections, we aim to improve the performance obtained when executing applications over the IDV-E platform both by improving the framework through an improved HLS flow and by improving the task scheduling using the Fast Task Scheduler developed in Task 2.5.
- Objective 3 - Fine-tuned thermal policies integrated with an innovative cooling technology. As explained in Objective 1, the power measurement tools integrated in the OmpSs HLS flow for IDV-E provide the basis for integrating fine-tuned thermal policies developed in Task 4.5.
- Objective 4 - Seamless integration of reconfigurable accelerators. The OmpSs runtime allows for seamless integration of reconfigurable accelerators as can be seen in their respective sections. The HLS flow developed is necessary to allow this seamless integration of new HLS reconfigurable accelerators developed.
- Objective 5 - Development of new IPs. The Fast Task Scheduler IP is a key part of the OmpSs@FPGA framework. OmpSs@FPGA contributes to the IP development as a primary tool to test the IP functionality. It also provides design requisites that must be incorporated in the IP for the whole framework to work as expected. Also, new accelerators can be more easily developed in HLS thanks to the presented work.
- Objective 6 - Integrated Development Platform. Task based runtimes will be used in applications executing on the project platforms. It is important to highlight that IDV-E features a host CPU (ARM based) that has never been used before to drive computation in a PCIe attached FPGA. Developing the system in a way that is compatible with different new CPUs helps to ensure new host CPUs (like EPI CPUs) will be able to drive this kind of computation in the future.

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

- Strategic Goal #1: Alignment with the European Processor Initiative (EPI). As shown in this deliverable the OmpSs@FPGA task-based programming model provides a system that can use an EPI processor to drive computations in a cluster of FPGA PCIe attached accelerators. Also, as described deliverable 2.11, the programming model allows to manage a manycore RISC-V processor with significant performance improvement over other state-of-the-art approaches.
- Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The OmpSs@FPGA framework and its

HLS flow is developed following the open-source model and is freely available in its GitHub repository.

- Strategic Goal #3: Opening of new usage domains. The task-based frameworks address the problem of simplifying the task of executing applications over FPGA-based computing platforms. In this sense, we expect that through the improvement of the tool, it will open the possibility of executing efficiently new applications on the objective platforms.

3.2 Compilation process

Figure 3.1 shows the compilation process in the OmpSs@FPGA framework developed. A C/C++ source file is read by the LLVM compiler where a frontend phase splits the code into two different flows: SMP and FPGA. As outline tasks are not supported, this distinction is done through C/C++ function annotation with task declaration pragmas. In OmpSs@FPGA, tasks or kernels can target both SMP or FPGA devices. The SMP part of the code, i.e. main code and tasks that do not have an FPGA target, is separated and its compiler directives are replaced by Nanos6 API calls. The Nanos6 runtime has a dedicated API for FPGA tasks, which uses internally the xTasks library, containing the low-level code to communicate with the FPGA. It is separated from the main runtime because each hardware platform uses different communication protocols, depending on the board memory model (e.g. shared like SoCs or distributed like PCIe attached FPGAs).

The FPGA code is also separated and integrated with a wrapper code, which communicates with a hardware runtime inside the FPGA, accesses main memory to load/store local memories and starts the actual hardware task engine. This wrapper is in fact C++ code with Vitis HLS pragmas. Since generally FPGAs count with on-chip RAM (e.g. BRAMs), the kernels can exploit this feature by storing data in this local memory. Depending on the memory model, main DRAM can be shared with the CPU or featured separately in the FPGA board. In both cases accessing local memory is faster, and OmpSs@FPGA allows to declare arrays stored in local memory and use them inside a task. The difference with other approaches like CUDA or OpenCL is that the local copies are automated in the wrapper and thus transparent to the user, who otherwise must code them explicitly in the kernel. Once the code is transformed by LLVM, it is passed to the Accelerator Integration Tool (AIT). This tool feeds all the high-level codes to the vendor-provided tools and integrates them inserting the proper connections with the hardware runtime and the FPGA I/O pins to generate the final bitstream.

This integrated compilation process has some useful features such as compilation of the whole system (bitstream and executable file) from a single command and automatic connection and integration of the hardware design, reducing the complexity of an otherwise error-prone process.

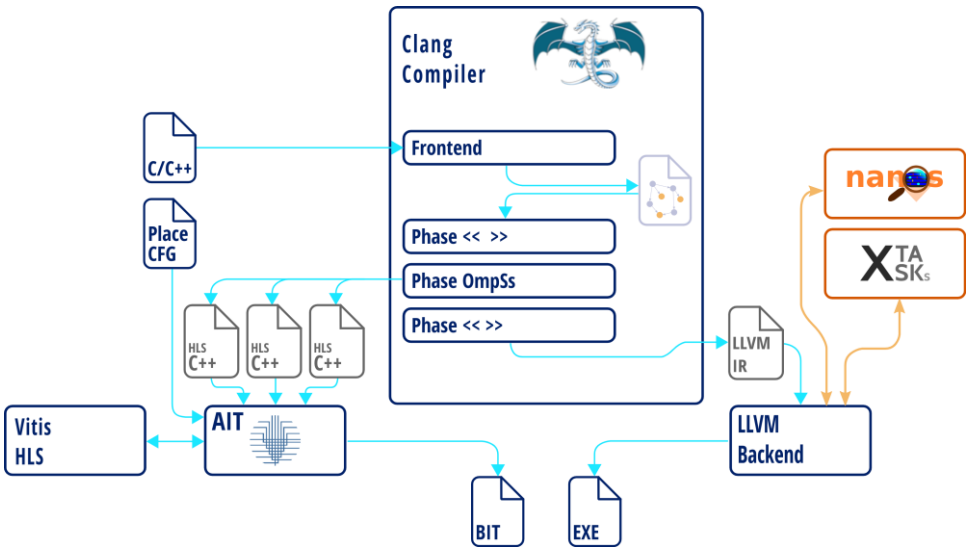


Figure 3.1: OmpSs@FPGA LLVM compiler process

3.2.1 OmpSs@FPGA Clang/LLVM compiler

In this work we have introduced changes in the frontend and backend of Clang to provide support to OmpSs@FPGA programming model for Vitis HLS. Figures 3.2 and 3.3 show the schemes of the frontend and backend support for FPGA.

On one hand, frontend support performs the following main tasks:

- Preprocessing and lexer to identify the different tokens in the code.
- Parsing and semantic analysis to accept HLS pragmas and analyze any dependence introduced by their clauses.
- Parsing and semantic analysis to accept HLS types (i.e. half precision).
- Parsing and semantic analysis to accept OmpSs-2 pragmas and clauses, analysing any dependence introduced by their clauses.

On the other hand, backend support has two main paths:

- Host path: Generates the intermediate code for LLVM to obtain the host binary.
- FPGA path: Performs the wrapper generation for AIT. In this case we have had to implement and add new AST nodes to be able to modify the programmer code, create a wrapper with all the necessary ports and support task creation and synchronization.

During the backend process it is necessary to modify the original code to create the appropriate wrapper. On the other hand, it is not possible to modify the AST generated by Clang. Therefore, we have implemented a replacement mechanism to modify the source code at the printer step.

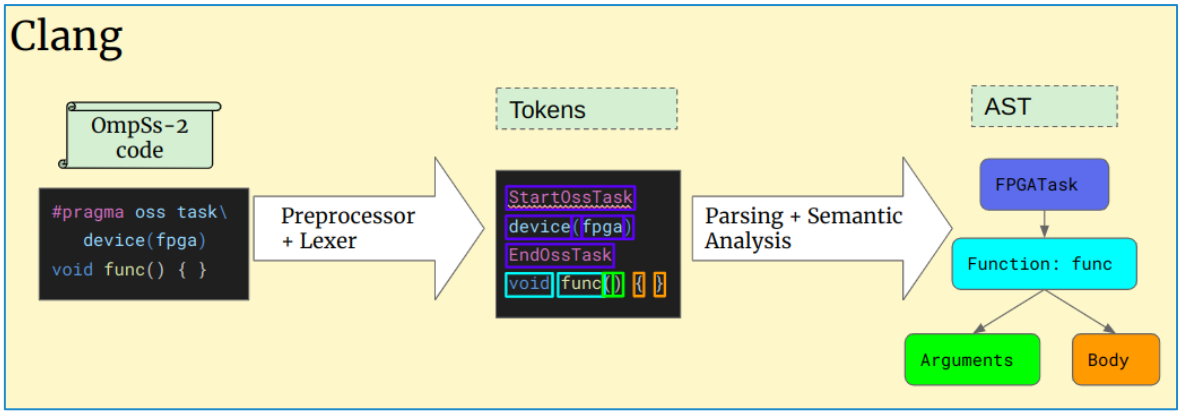


Figure 3.2: Clang frontend support for OmpSs@FPGA

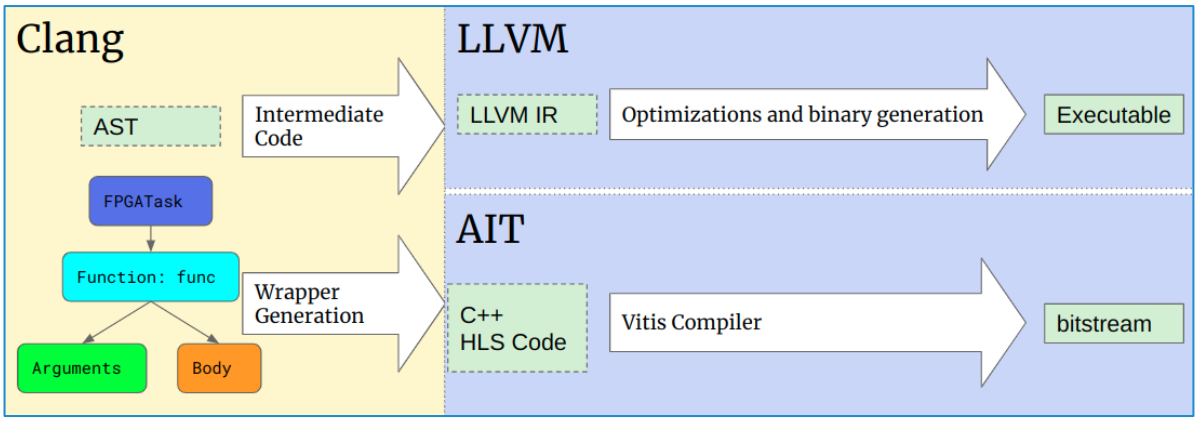


Figure 3.3: Clang backend support for OmpSs@FPGA

Listing 3.4 shows an excerpt of a OmpSs@FPGA program, showing a FPGA task annotation.

```
#pragma oss task device(fpga) in([CONST_BSIZE*CONST_BSIZE]a,[CONST_BSIZE*CONST_BSIZE]b)\
                              inout([CONST_BSIZE*CONST_BSIZE]c) copy_deps num_instances(2)
void matmulBlock_hw(elem_t a[CONST_BSIZE*CONST_BSIZE], elem_t b[CONST_BSIZE*CONST_BSIZE],
                    elem_t c[CONST_BSIZE*CONST_BSIZE]) {
        int i,j,k;
#pragma HLS ARRAY_PARTITION variable=a cyclic factor=CONST_BSIZE/2 dim=1
#pragma HLS ARRAY_PARTITION variable=b block factor=CONST_BSIZE/2 dim=1
 loop_i_matmul:
   for (i = 0; i < BSIZE; i++) {
loop_j_matmul:
      for (j = 0; j < BSIZE; j++) {
#pragma HLS PIPELINE II=2
         elem_t sum = c[i*BSIZE + j];
loop_k_matmul:
         for (k = 0; k < BSIZE; k++) {
            sum += a[i*BSIZE + k] * b[k*BSIZE + j];
         }
         c[i*BSIZE + j] = sum;
      }
   }
}
```

Listing 3.4: OmpSs Program Example FPGA task

Our framework extracts this kernel and generates a Vitis HLS compatible code. We generate the top function (called wrapper) to include the communication ports between the CPU and the FPGA. This

wrapper is charge of the task management protocol with the FPGA hardware runtime, in addition to perform the declaration of the local BRAM variables and copies, if needed.

3.2.1.1 Top wrapper function

Function parameters and Vitis HLS directives:

Listing 3.5 shows an excerpt of the transformation done of the code, to be integrated in a Vitis HLS design to obtain the bitstream. By default, the original parameters of the function-task are kept with a prefix "mcxx_". This prefix is automatically inserted. Those parameters will be automatically transformed to AXI ports by Vitis HLS to access the real parameters. In addition, two specific HLS stream ports: "mcxx_inPort" and "mcxx_outPort" are added to allow the FPGA hardware runtime management of the accelerators.

```
Void matmulBlock_hw_wrapper(hls::stream<ap_uint<64> >& mcxx_inPort,
                            hls::stream<mcxx_outaxis>& mcxx_outPort,
                            elem_t * mcxx_a, elem_t * mcxx_b, elem_t * mcxx_c) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=mcxx_inPort
#pragma HLS interface axis port=mcxx_outPort
#pragma HLS interface m_axi port=mcxx_a
#pragma HLS interface m_axi port=mcxx_b
#pragma HLS interface m_axi port=mcxx_c
```

Listing 3.5: Wrapper header example of a FPGA task

Local variables (BRAM) and accelerator copies

By default, local variables are automatically declared by our framework with the original names of the parameters and with the static size specified at the programmer definition. This allows us to keep the exact original code of the task-function. Listing 3.6 shows an excerpt of the code of the wrapper with the declaration of the three original parameters as local variables: a, b, and c. Those variables are arrays of 1 dimension of 4096 elem_t elements. Those variables are synthesized to BRAM memories that will be partitioned following the programmer Vitis HLS annotations of the original code.

```
void matmulBlock_hw_wrapper(hls::stream<ap_uint<64> >& mcxx_inPort,
                            hls::stream<mcxx_outaxis>& mcxx_outPort,
                            elem_t * mcxx_a, elem_t * mcxx_b, elem_t * mcxx_c) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=mcxx_inPort
#pragma HLS interface axis port=mcxx_outPort
#pragma HLS interface m_axi port=mcxx_a
#pragma HLS interface m_axi port=mcxx_b
#pragma HLS interface m_axi port=mcxx_c
static elem_t c[4096];
static elem_t a[4096];
static elem_t b[4096];
...
```

Listing 3.6: Wrapper local variables example

In addition, wrapper code contains the necessary copies of the data using Vitis HLS memcpy . Listing 3.7 shows the copies of a, b, and c parameters. Those copies are done if, and only if, the FPGA hardware runtime activates it through the flag variables (see Deliverable 2.10 to see the circumstances where copies are avoided due to runtime optimizations).

```
…
  if (mcxx_flags_2[4]) {
    memcpy(c, mcxx_c + mcxx_offset_2/sizeof(elem_t), (16384));
  }
  if (mcxx_flags_0[4]) {
    memcpy(a, mcxx_a + mcxx_offset_0/sizeof(elem_t), (16384));
  }
  if (mcxx_flags_1[4]) {
    memcpy(b, mcxx_b + mcxx_offset_1/sizeof(elem_t), (16384));
  }
  matmulBlock_hw_moved(a, b, c);
  if (mcxx_flags_2[5]) {
    memcpy(mcxx_c + mcxx_offset_2/sizeof(elem_t), c, (16384));
  }
...
```

Listing 3.7 Wrapper memory copies and original kernel invocation

Then, the original kernel (with suffix moved) is called with the local variables, and finally, the output result is copied through the mcxx_c port, if needed.

The number of elements to be copied from/to are determined by mcxx_offset variables. Those values are provided by the FPGA hardware runtime.

Runtime control management

Wrapper code reads control information through the mcxx_inPort port to manage the copies and execution of the kernel. For each parameter it reads two data info to know, for instance, if it has to copy the data or not, and the size of the data it has to copy. In addition, it read information regarding the identification of the task and its parent task. Listing 3.8 shows the declaration of the local variables associated to each parameter and the task information.

```
  …
  mcxx_inPort.read(); //command word
  __mcxx_taskId = mcxx_inPort.read();
  ap_uint<64> __mcxx_parent_taskId = mcxx_inPort.read();
  ap_uint<8> mcxx_flags_0;
  ap_uint<64> mcxx_offset_0;
  ap_uint<8> mcxx_flags_1;
  ap_uint<64> mcxx_offset_1;
  ap_uint<8> mcxx_flags_2;
  ap_uint<64> mcxx_offset_2;
  {
  #pragma HLS protocol fixed
    {
      mcxx_flags_0 = mcxx_inPort.read()(7,0);
      ap_wait();
      mcxx_offset_0 = mcxx_inPort.read();
    }
    ap_wait();
    {
      mcxx_flags_1 = mcxx_inPort.read()(7,0);
      ap_wait();
      mcxx_offset_1 = mcxx_inPort.read();
    }
    ap_wait();
    {
```

```
      mcxx_flags_2 = mcxx_inPort.read()(7,0);
      ap_wait();
      mcxx_offset_2 = mcxx_inPort.read();
    }
    ap_wait();
  }
...
```

Listing 3.8: Excerpt of wrapper code reading control and task information.

Once the code has been run and copy out the data, if needed, the wrapper has to inform the hardware runtime about the end of the task. This is done using the mcxx_outPort, as shown in Listing 3.9.

```
...
 {
  #pragma HLS protocol fixed
    ap_uint<64> header = 0x03;
    ap_wait();
    mcxx_write_out_port(header, 0, 0, mcxx_outPort);
    ap_wait();
    mcxx_write_out_port(__mcxx_taskId, 0, 0, mcxx_outPort);
    ap_wait();
    mcxx_write_out_port(__mcxx_parent_taskId, 0, 1, mcxx_outPort);
    ap_wait();
  }
```

Listing 3.9: Excerpt of wrapper code notifying the end of the task and the task information.

Wide and Unified Memory Port

Each accelerator needs to access memory to get data, whether it is stored in a local memory or directly accessed. Vitis HLS tool uses an AXI4 interface to handle memory reads and writes. The most straightforward solution to convert pointer accesses in C to AXI transactions is through the creation of an independent memory port per each pointer argument. Vitis HLS syntax to read and write from an AXI interface is the same as in C to access a pointer or array.

However, there are two main problems with this implementation. First, using a different interface for each pointer or array argument can easily outnumber memory access ports. The number of real memory ports on an FPGA system is quite low (except for devices with HBM), for instance, discrete boards like the Xilinx Alveo U200 have only one per DDR module, whereas SoCs like the Xilinx Zynq UltraScale+ have up to six. Reducing several ports to one is indeed possible with an AXI interconnect, but it increases resource usage and hinders design routability. Moreover, performance-wise it is better not to have more than one memory port in a single Vitis HLS module. Our efforts to make the tool use more than one port in parallel have been unsuccessful, since Vitis HLS seems to always respect the access order between all external interfaces.

The second and most important handicap of the mentioned implementation is the bandwidth. The default behavior is to use a data bus with the same bit width as the data type. However, the FPGA memory controller may allow a wider data bus. Therefore, to exploit the memory bandwidth of the system, the AXI data bus used must be as wide as supported by the memory controller, if the frequency is lower or equal than the memory side of the AXI interface. This way, each cycle the accelerator can read multiple data elements.

To conclude, to remove redundant resources and improve performance, we added the possibility to use a single memory port with a configurable data width. Specified as a compiler flag at compile time, the user can provide the bit width of the data bus. This port is shared across all array arguments, thus, limiting the total required AXI interfaces to one per accelerator. Using this feature is only possible for array arguments

that are stored in local memory. If the task directly accesses memory, the shared memory port is not supported. We have not yet found any use case that would benefit from this feature without the use of local memory.

Listing 3.10 shows a portion of the wrapper HLS code generated by OmpSs@FPGA LLVM from FPGA task mastrixBlock_hw seen above. The resulted wrapper contains a single memory interface, mcxx_memport with a 128-bit data bus and two local memories. The compiler also generates the necessary code to copy the data from main memory, mainly loops enhanced with Vitis HLS pragmas to maximize bandwidth.

HLS pipeline pragmas are used to pipeline the memory load with the store to local memory. The inner loop is fully unrolled automatically by the tool, and the Initiation Interval (II) depends on the partition of the local memory. To generate the unified memory interface with a specified width, the argument "-fompss-fpga-memory-port-width <width>" must be provided to the compiler in the invocation command.

There are some restrictions to consider when using the unified port. The HLS generated code, for each parameter, uses an interface declared as a pointer to an unsigned integer type with the specified width. Therefore, all accesses must be aligned to that type. In the example shown in listing 3.10, the memory port is used to copy from a 128 bit unsigned integer pointer to a local array of 4096 floats. Hence the address stored in param[0] must be aligned to 16 bytes or the lower bits will be discarded in the division of the condition of innermost loop "_ _ j". Accessed type width must be multiple of the memory port width. There is a compiler flag to enable support for copies that are not multiple of this width, at the cost of using more resources and a more complex logic. Moreover, the union used to do the casting between types, mainly to avoid float to integer conversion, uses a 64-bit unsigned integer type. As a result, the casted type cannot have more than 64 bits, and due to union restrictions, it cannot have a non-trivial constructor.

Another important concept to have in mind is that to get the maximum bandwidth (II=1 in the copy loop), the local array must be able to be written or read at the same rate as the memory. I.e. it needs enough ports to read or write a memory word in one cycle. For instance, in the example the a, b, and c memories should have at least 4 ports. Translated into Vitis HLS terms, they should have a cyclic partition of factor 2 if implemented as BRAMs, since each one has two ports. This happens for a and c, but not for b, which has been partitioned using blocks of elements. Here there is a tradeoff between an extra cost during the copy vs the extra cost during the matrix computation.

```
void matmulBlock_hw_wrapper(hls::stream<ap_uint<64> >& mcxx_inPort,
                            hls::stream<mcxx_outaxis>& mcxx_outPort,
                            ap_uint<128>* mcxx_memport) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=mcxx_inPort
#pragma HLS interface axis port=mcxx_outPort
#pragma HLS interface m_axi port=mcxx_memport
  static elem_t a[4096];
  static elem_t c[4096];
  static elem_t b[4096];
  ...

  // Read task parameters, address of a param[0]
  if (mcxx_flags_0[4]) {
    for (int __i = 0; __i < ((16384) - 1)/sizeof(ap_uint<128>)+1; ++__i) {
    #pragma HLS pipeline II=1
      ap_uint<128> __tmpBuffer;
      __tmpBuffer = *(mcxx_memport + mcxx_offset_0/sizeof(ap_uint<128>) + __i);
      for (int __j=0; __j <(sizeof(ap_uint<128>)/4); __j++) {
        __mcxx_cast<elem_t> cast_tmp;
        cast_tmp.raw = __tmpBuffer((__j+1)*4*8-1,__j*4*8);
        a[__i*(sizeof(ap_uint<128>)/4)+__j] = cast_tmp.typed;
      }
    }
  }
```

```
   ...

}
```

Listing 3.10: Wrapper Example of the FPGA task using wide port 128 bits

Mixed precision for CNN support

In deliverable 4.8 describes the necessary LLVM modifications that allow to use Vitis HLS specific mixed precision types, very useful for CNNs.

3.2.1.2 Data synchronizations

Programmers may need to perform an exclusive section in the FPGA using critical as shown in foo2 function in Listing 3.11.

```
void foo2(const float *src, float *dst) {
   #pragma oss critical
   *dst = *src;
}
#pragma oss task device(fpga) in([1] src) out([1] dst)
void foo(const float *src, float *dst) {
   foo2(src, dst);
}
```

Listing 3.11: Excerpt of a code using FPGA tasks with a critical section.

To implement a critical, the FPGA must execute a function at the beginning and end of the sentence, activating and disabling a mutual exclusion zone; mcxx_set_lock to activate it or wait, and mcxx_unset_lock, to deactivate it. To be able to perform locks they require the in and out ports. So, the first one transformation that should be done is to add these ports as parameters to both the function that uses them as the entire chain of dependencies that make use of it (in this case, “foo2” and “foo”). The next one transformation involves replacing the node with the critical one by a BlockStmt (a code block of the style “{stmts;*}”) that contains a function call, the child of the critical (that is, the statement that was marking as exclusive, in this case *dst = *src) and another function call. The resulting code of these two transformations is shown in Listing 3.12.

```
void mcxx_set_lock(hls::stream<ap_uint<64> >& mcxx_inPort,
                   hls::stream<mcxx_outaxis>& mcxx_outPort);
void mcxx_unset_lock(hls::stream<mcxx_outaxis>& mcxx_outPort);

void foo2_moved(const float *src, float *dst,
                hls::stream<ap_uint<64> > &mcxx_inPort,
                hls::stream<mcxx_outaxis> &mcxx_outPort) {
    {
        mcxx_set_lock(mcxx_inPort, mcxx_outPort);
        *dst = *src;
        mcxx_unset_lock(mcxx_outPort);
    }
}
void foo_moved(const float *src, float *dst,
               hls::stream<ap_uint<64> > &mcxx_inPort,
               hls::stream<mcxx_outaxis> &mcxx_outPort) {
    foo2_moved(src, dst, mcxx_inPort, mcxx_outPort);
}

...

void mcxx_set_lock(hls::stream<ap_uint<64> >& mcxx_inPort,
                   hls::stream<mcxx_outaxis>& mcxx_outPort) {
#pragma HLS inline
  ap_uint<64> tmp = 0x4;
  ap_uint<8> ack;
```

```
  do {
    ap_wait();
    mcxx_write_out_port(tmp, 1, 1, mcxx_outPort);
    ap_wait();
    ack = mcxx_inPort.read();
    ap_wait();
  } while (ack == 0);
}

void mcxx_unset_lock(hls::stream<mcxx_outaxis>& mcxx_outPort) {
#pragma HLS inline
  ap_uint<64> tmp = 0x6;
  mcxx_write_out_port(tmp, 1, 1, mcxx_outPort);
}
```

Listing 3.12: Excerpt of the code to support data sharing synchronization.

The code of mcxx_set/unset_locks is also included in the wrapper code (shown in Listing 3.12) so that the Vitis HLS toolchain can implement the necessary hardware. We use the in/out memory port to implement the locks.

3.2.1.3 Task creation and synchronization support

One important feature of OmpSs@FPGA is the creation of tasks from inside FPGA tasks. Listing 3.13 shows an example of FPGA task foo invoking FPGA task foo2. In addition, a taskwait is added to wait for them.

```
#pragma oss task device(fpga) in([1] src) out([1] dst)
void foo2(float *src, float *dst) {
   *dst = *src;
}
#pragma oss task device(fpga) inout([1] src) inout([1] dst)
void foo(float *src, float *dst) {

   if (src[0] > 10)
      foo2(src, dst);
   else
      foo2(dst, src);

#pragma oss taskwait
}
```

Listing 3.13: Excerpt of a OmpSs code with FPGA sibling tasks.

When a task has to generate other tasks, it may have to pass through arguments of the parent task as show in the example. Note, however, that pointers received by the top wrapper function are translated to AXI ports in order to be able to access external data. Because it must preserve the position relative to the port, the pointers are replaced by a 64-bit number that simply is updated with the current memory position, but this number cannot then be used for a read from memory as shown in Listing 3.14.

```
template <typename T>
struct __mcxx_ptr_t {
  T *ptr;
  unsigned long long int val;
  __mcxx_ptr_t(T *ptr, unsigned long long int val) : ptr(ptr), val(val) {}
  __mcxx_ptr_t() {}
  inline operator __mcxx_ptr_t<const T>() const {
    return __mcxx_ptr_t<const T>(ptr, val);
  }
  template <typename V> inline __mcxx_ptr_t<T> operator+(V const val) const {
    return __mcxx_ptr_t<T>(ptr, this->val + val * sizeof(T));
  }
  template <typename V> inline __mcxx_ptr_t<T> operator-(V const val) const {
```

```
    return __mcxx_ptr_t<T>(ptr, this->val - val * sizeof(T));
  }
  template <typename V> inline operator V() const { return (V)val; }
  T& operator[](long long int i) { return ptr[val/sizeof(T)+i]; }
};
```

Listing 3.14: Port replacement structure

An example of the use of this class would be the one shown in Listing 3.15, where we observe the transformations made in user code for the task case "foo" in Listing 3.13. FPGA task foo2 does not appear in the code shown. Two calls to invocations to mcxx_task_create are seen instead. This function receives the arguments and dependencies initialized in the parent task foo, and the identification of the foo2 FPGA task (4775314383). "mcxx_task_create" function is in charge of passing the task hash to be created, its dependencies and arguments to the Hardware Runtime, as it was created in the host.

```
void foo_moved(__mcxx_ptr_t<float > src, __mcxx_ptr_t<float > dst,
               hls::stream<ap_uint<8> > &mcxx_spawnInPort,
               hls::stream<mcxx_outaxis> &mcxx_outPort) {
    if (src[0] > 10)
        {
            unsigned long long __mcxx_args[2];
            unsigned long long __mcxx_deps[2];
            __mcxx_ptr_t<float > __mcxx_arg_0;
            __mcxx_arg_0 = src;
            __mcxx_args[0U] = __mcxx_arg_0.val;
            __mcxx_ptr_t<float > __mcxx_dep_0;
            __mcxx_dep_0 = src;
            __mcxx_deps[0U] = 1UL << 58UL | __mcxx_dep_0.val;
            __mcxx_ptr_t<float > __mcxx_arg_1;
            __mcxx_arg_1 = dst;
            __mcxx_args[1U] = __mcxx_arg_1.val;
            __mcxx_ptr_t<float > __mcxx_dep_1;
            __mcxx_dep_1 = dst;
            __mcxx_deps[1U] = 2UL << 58UL | __mcxx_dep_1.val;
            mcxx_task_create(4775314383UL, 255U, 2U, __mcxx_args,
                             2U, __mcxx_deps, 0U, 0U, mcxx_outPort);
        }
    else
        {
            unsigned long long __mcxx_args[2];
            unsigned long long __mcxx_deps[2];
            __mcxx_ptr_t<float > __mcxx_arg_0;
            __mcxx_arg_0 = dst;
            __mcxx_args[0U] = __mcxx_arg_0.val;
            __mcxx_ptr_t<float > __mcxx_dep_0;
            __mcxx_dep_0 = dst;
            __mcxx_deps[0U] = 1UL << 58UL | __mcxx_dep_0.val;
            __mcxx_ptr_t<float > __mcxx_arg_1;
            __mcxx_arg_1 = src;
            __mcxx_args[1U] = __mcxx_arg_1.val;
            __mcxx_ptr_t<float > __mcxx_dep_1;
            __mcxx_dep_1 = src;
            __mcxx_deps[1U] = 2UL << 58UL | __mcxx_dep_1.val;
            mcxx_task_create(4775314383UL, 255U, 2U, __mcxx_args,
                             2U, __mcxx_deps, 0U, 0U, mcxx_outPort);
        }
    mcxx_taskwait(mcxx_spawnInPort, mcxx_outPort);
}
```

Listing 3.15: Excerpt of code showing the task creation of foo2 tasks.

Finally, "mcxx_taskwait" is used to wait for the child tasks. Listing 3.16 shows the code of mcxx_taskwait and mcxx_task_create functions, automatically generated by our framework in the foo wrapper code.

```
void mcxx_task_create(const ap_uint<64> type, const ap_uint<8> instanceNum,
                      const ap_uint<8> numArgs, const unsigned long long int args[],
                      const ap_uint<8> numDeps, const unsigned long long int deps[],
                      const ap_uint<8> numCopies,
                      const __fpga_copyinfo_t copies[],
                      hls::stream<mcxx_outaxis>& mcxx_outPort) {
```

```
#pragma HLS inline
  const ap_uint<2> destId = 2;
  ap_uint<64> tmp;
  tmp(15,8)  = numArgs;
  tmp(23,16) = numDeps;
  tmp(31,24) = numCopies;
  mcxx_write_out_port(tmp, destId, 0, mcxx_outPort);
  mcxx_write_out_port(__mcxx_taskId, destId, 0, mcxx_outPort);
  tmp(47,40) = instanceNum;
  tmp(33,0)  = type(33,0);
  mcxx_write_out_port(tmp, destId, 0, mcxx_outPort);
  for (ap_uint<4> i = 0; i < numDeps(3,0); ++i) {
    mcxx_write_out_port(deps[i], destId, numArgs == 0 && numCopies == 0 && i == numDeps-1,
mcxx_outPort);
  }
  for (ap_uint<4> i = 0; i < numCopies(3,0); ++i) {
    mcxx_write_out_port(copies[i].copy_address, destId, 0, mcxx_outPort);
    tmp(7,0) = copies[i].flags;
    tmp(15,8) = copies[i].arg_idx;
    tmp(63,32) = copies[i].size;
    mcxx_write_out_port(tmp, destId, numArgs == 0 && i == numCopies-1, mcxx_outPort);
  }
  for (ap_uint<4> i = 0; i < numArgs(3,0); ++i) {
    mcxx_write_out_port(args[i], destId, i == numArgs-1, mcxx_outPort);
  }
}

void mcxx_taskwait(hls::stream<ap_uint<8> >& mcxx_spawnInPort,
                   hls::stream<mcxx_outaxis>& mcxx_outPort) {
#pragma HLS inline
  ap_wait();
  mcxx_write_out_port(__mcxx_taskId, 3, 1, mcxx_outPort);
  ap_wait();
  mcxx_spawnInPort.read();
  ap_wait();
}
```

Listing 3.16: mcxx_task_create and mcxx_taskwait functions.

3.2.1.4 FPGA Instrumentation support

Programmer can activate the instrumentation code. This instrumentation allows, by default, to obtain timing information of data copies done by the wrapper code and the execution of the kernel. In addition, if there are critical sections it allows you to obtain information about the synchronization overheads.

This is possible including a new port as an argument of the wrapper code and instrumenting the wrapper code with calls to mcxx_instrument_event. Listing 3.17 shows an excerpt of the wrapper code of the mamulBlock_hw FPGA task where the new argument and part of the instrumentation are done.

```
void matmulBlock_hw_wrapper(hls::stream<ap_uint<64> >& mcxx_inPort,
                            hls::stream<mcxx_outaxis>& mcxx_outPort,
                            ap_uint<128>* mcxx_memport,
                            hls::stream<__mcxx_instrData_t>& mcxx_instr) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=mcxx_inPort
#pragma HLS interface axis port=mcxx_outPort
#pragma HLS interface m_axi port=mcxx_memport
#pragma HLS interface ap_hs port=mcxx_instr

...

mcxx_instrument_event(78, __mcxx_taskId, mcxx_instr);
  if (mcxx_flags_1[4]) {
    for (int __i = 0; __i < ((16384) - 1)/sizeof(ap_uint<128>)+1; ++__i) {
    #pragma HLS pipeline II=1
      ap_uint<128> __tmpBuffer;
      __tmpBuffer = *(mcxx_memport + mcxx_offset_1/sizeof(ap_uint<128>) + __i);
```

```
      for (int __j=0; __j <(sizeof(ap_uint<128>)/4); __j++) {
        __mcxx_cast<elem_t> cast_tmp;
        cast_tmp.raw = __tmpBuffer((__j+1)*4*8-1,__j*4*8);
        b[__i*(sizeof(ap_uint<128>)/4)+__j] = cast_tmp.typed;
      }
    }
  }

  ...

mcxx_instrument_event(206, __mcxx_taskId, mcxx_instr);
mcxx_instrument_event(80, __mcxx_taskId, mcxx_instr);
  matmulBlock_hw_moved(a, b, c, mcxx_instr);
mcxx_instrument_event(208, __mcxx_taskId, mcxx_instr);
mcxx_instrument_event(79, __mcxx_taskId, mcxx_instr);
  if (mcxx_flags_2[5]) {
    for (int __i = 0; __i < ((16384) - 1)/sizeof(ap_uint<128>)+1; ++__i) {
    #pragma HLS pipeline II=1
      ap_uint<128> __tmpBuffer;
      for (int __j=0; __j <(sizeof(ap_uint<128>)/4); __j++) {
        __mcxx_cast<elem_t> cast_tmp;
        cast_tmp.typed = c[__i*(sizeof(ap_uint<128>)/4)+__j];
        __tmpBuffer((__j+1)*4*8-1,__j*4*8) = cast_tmp.raw;
      }
      *(mcxx_memport + mcxx_offset_2/sizeof(ap_uint<128>)+ __i) = __tmpBuffer;
    }
  }
mcxx_instrument_event(207, __mcxx_taskId, mcxx_instr);
  ...
```

Listing 3.17: Wrapper Example of a FPGA task with instrumentation

Note that each event has its pair: <start, end> event points. Any start event instrumentation value is smaller than “200” while end event instrumentation is start event value + 128.

Listing 3.18 shows the mcxx_instrumentation_event function, which writes the event identification, the task id to the special instrumentation port.

```
void mcxx_instrument_event(unsigned char event,
                           unsigned long long payload,
                           hls::stream<__mcxx_instrData_t>& mcxx_instr) {
#pragma HLS inline
  __mcxx_instrData_t tmp;
  tmp.range(63, 0) = payload;
  tmp.range(95, 64) = event & 0x7F;
  tmp.range(103, 96) = event >> 7;
  tmp.bit(104) = 1;
  ap_wait();
  mcxx_instr.write(tmp);
  ap_wait();
}
```

Listing 3.18: mcxx_instrument_event instrumentation function

3.2.1.5 Json Wrapper Information

A particularity of AIT is that part of the compilation information is received through a Json file. This document contains, for each wrapper file generated, the information regarding the location of the file, the identifier of the wrapper, and information about the characteristics it uses. Given that this Json file is composed of the information of the various compilation stages we are in combining, it has been chosen to generate an "extracted.json.part" file in the same directory as the wrappers with the relevant information. Once in AIT's JobAction, it reads these parts and combines them in a single Json file. The content of this file would be like Listing 3.19.

```
{
{
    "full_path" : "<path>/task_hls_automatic_clang.cpp",
    "filename" : "task_hls_automatic_clang.cpp",
    "name" : "task",
    "type" : 5998371774,
    "num_instances" : 1,
    "task_creation" : false,
    "instrumentation" : false,
    "periodic" : false,
    "lock" : false,
    "deps" : false,
    "ompif" : false
}
}
```

Listing 3.19: Example content of the "extracted.json" file

3.2.2 Accelerator Integration Tool (AIT)

AIT automatically builds the block design with all the accelerators requested by the programmer, creating the necessary interconnections between them and custom and Xilinx IPs, allowing the management, scheduling and memory accesses of the tasks and the communication with the host application, if needed. Among all the IPs used, the main custom IPs are the hardware runtime (detailed in deliverable 2.11 and 2.10), communication accelerators to support OMPIF (partially explained in deliverable 4.6) and the custom interleaving and priority memory access (explained in deliverable 4.6). The runtime basically receives commands from the software runtime and forwards them to the related accelerator. In the case of having more than one accelerator available for any given task, a simple round-robin scheduler dispatches the task to one of the accelerators. After the execution finishes, it informs the hardware runtime which forwards the information to the software control. The communication accelerators allow decoupling processing and point-to-point communication between processing element devices; for instance, FPGA accelerators. Finally, the custom interleaving and priority IPs reduce memory conflicts and increase the memory bandwidth achieved. The framework can keep the intermediate and final design projects so that the programmer can open either the HLS project or the final hardware design in case of having any issue with, for instance, a negative WNS.

Figures 3.20 and 3.21 show the N-body overall view and zoom full screen, respectively, of the final block design with target Alveo u200. This block design includes the programmer accelerators and the FST hardware runtime.

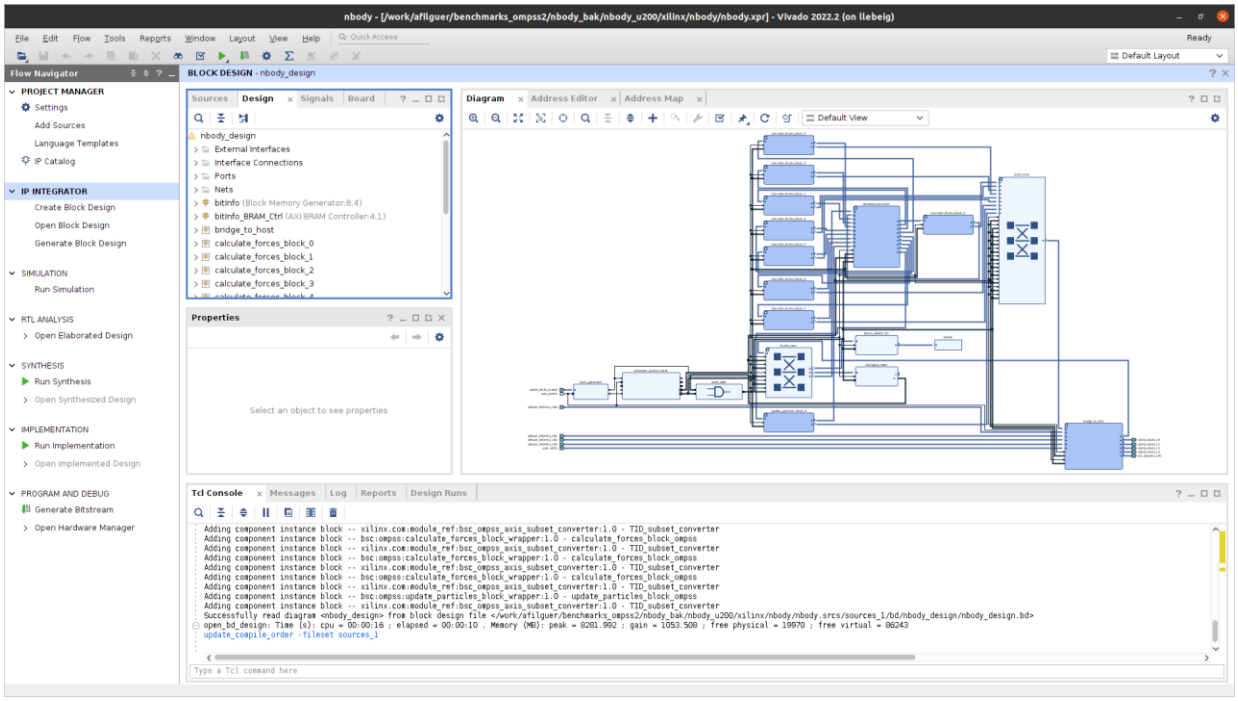


Figure 3.20: Overall view of the final hardware block design of N-body application generated by AIT with 7 kernel accelerators (FPGA tasks).

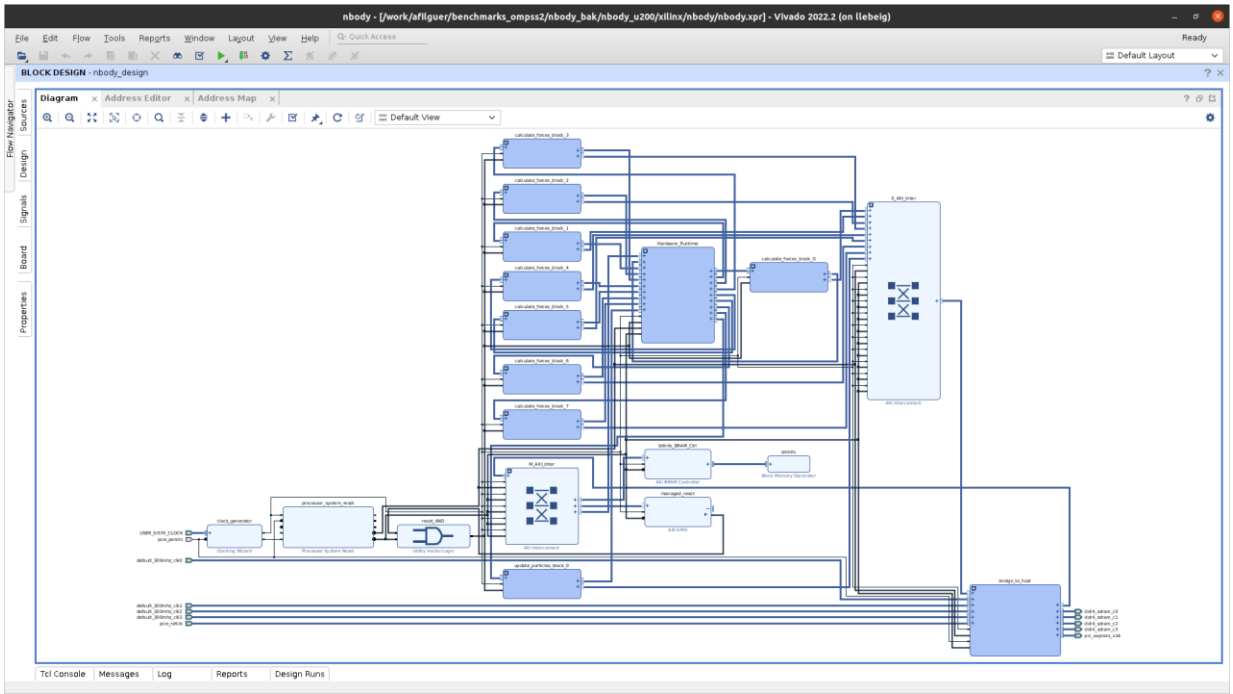


Figure 3.21: Fullscreen of the final hardware block design of N-body application generated by AIT.

Figure 3.22 shows a zoom in of a Vivado/Vitis design for Matrix Multiply application with two instances of a matmulBlock_hw kernel seen above. The design is connected to the FPGA DDR RAM memory and PCIe to Host; providing support for discrete FPGAs connected via PCIe and Ethernet.

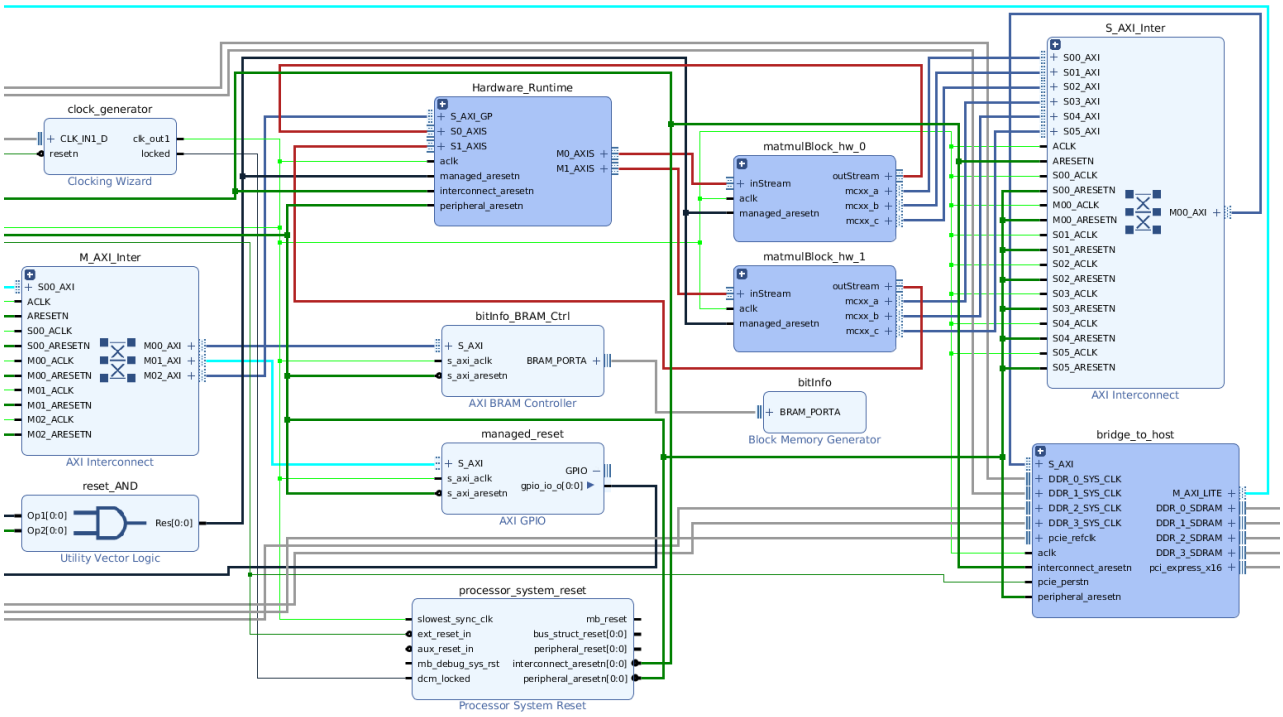


Figure 3.22: Matrix Multiply Hardware design with two accelerators instances of matmulBlock_hw FPGA task.

3.3 Custom IPs for the FPGA Hardware Runtime

As mentioned, AIT inserts a small hardware runtime, as shown in Figure 3.6 (Fast Task Scheduler described in deliverable 2.10), in the final hardware design. This runtime (Vivado block design shown in Figure 3.3.1) basically receives commands from the software runtime and forwards them to the related accelerator. In the case of having more than one accelerator available for any given task, a simple round-robin scheduler dispatches the task to one of the accelerators. After the execution finishes, it informs the hardware runtime which forwards the information to the software control. The characteristics of this FPGA Hardware Runtime are described in deliverables 2.10 and 2.11.

Figure 3.23 shows the block design of the Hardware Runtime IP in Figure 3.6. There you can see the input and output command queues, and the input and output stream connections. The Fast Task Scheduler oversees the management of all the task executions, copy optimizations, etc.

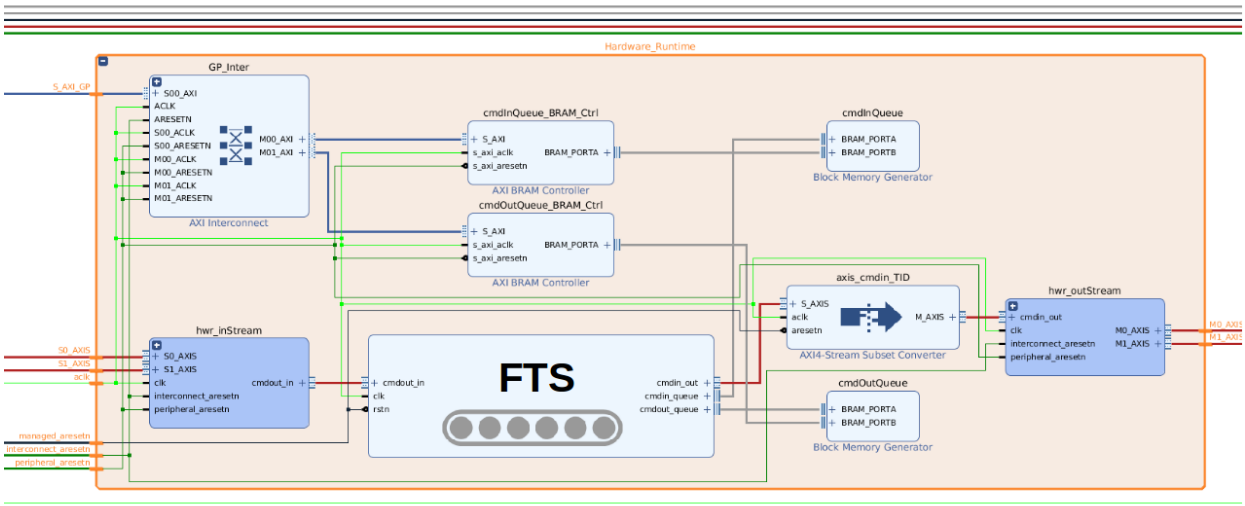


Figure 3.23: Hardware Runtime block design.

3.4 Accelerator Placement, Interleave and Priorities

In modern FPGA devices, place and route has become an increasingly difficult task due to an increase in resources and device complexity. This results in an exponential increase in implementation possibilities. Such a huge search space causes tools to have a hard time providing a good solution. This is even more challenging in chiplet-based devices due to their topology. In the same way, off-chip memory resources have grown both in size and number of modules. These resources are presented to the user as raw memory interfaces requiring the user to manage how accelerator kernels access off-chip memory to make effective use of the available bandwidth. Efficient usage of memory resources becomes a critical challenge as more computational resources are added to a design imposing more pressure on the memory subsystem. To tackle these problems, the OmpSs@FPGA framework has modified not the HLS flow but the hardware runtime that is instantiated in the final design, implementing new IPs that in conjunction with the programming model deliver very good performance results [Filgueras 2022, 2023, 2023b]. These modifications can be summarized as follows:

- Accelerator Placement: Acceleration in multi-SLR FPGAs may result in a below expected resource usage by the model due to the larger costs of propagating signals. To mitigate these larger costs of propagating signals across large regions, specially between different SLR, a new accelerator placement feature allows users to assign specific computation kernels to different SLR.
- Memory Interleaving for DDR channels: We have implemented a general and transparent way to efficiently place application data into separate memory modules. The goal is that accelerator accesses can be scattered across multiple memory interfaces to reduce access conflicts when several accelerators need to access data that otherwise, would be stored in the same memory module. This is implemented by inserting an interleaver module between accelerator memory interface and the memory interconnection and between any other IP that access off-chip memory, such as the PCIe block.
- Memory Priorities: An adverse effect observed in some applications is memory access conflicts. When more than one accelerator tries to access a single memory interface, one transaction is processed while the rest have to wait for the one in progress to finish. By enabling priorities, this latency can be hidden by allowing transactions to be pipelined. In this case a single accelerator

(the one with higher priority) will send multiple transaction requests in a row and data can be transferred at maximum throughput.

All these modifications source code (and the resulting IPs generated) can be found in the OmpSs@FPGA GitHub. A complete description of the features introduced in the context of Textarossa and an evaluation of its performance results is detailed in deliverable 4.6 task-based runtime systems, section OmpSs@FPGA - Task-based runtime for FPGAs.

3.5 HBM memory access

One of the main goals of AIT and OmpSs@FPGA is to abstract all FPGA-related complexity and heterogeneity by providing a set of high-level tools allowing programmers to guide the implementation process. This includes providing an efficient exploitation of available memory resources, without requiring an in-depth knowledge of how it is arranged, or which type of technology is used.

One of the challenges of the goal is supporting the different memory models used by the different FPGA boards available. HBM and DDR memories present different characteristics and consequently need different interfaces in order to be accessed from FPGA accelerators. The IDV-E platform developed in Textarossa features both types of memory. In OmpSs@FPGA support for both memories is developed. As both memories can be accessed at the same time/from different programs we have opted to include the support at the runtime level. The full description of the HBM specific support can be found in deliverable 4.6 task-based runtime systems, section OmpSs@FPGA - Task-based runtime for FPGAs.

3.6 FPGA Instrumentation Support

OmpSs@FPGA programming model framework allows the programmer to generate instrumented FPGA tasks to generate hardware execution traces with details of the execution of FPGA tasks. The programmer can specify at compile time that she/he wants an instrumented FPGA tasks and then, at runtime, it can be executed to generate the execution trace or not. The hardware execution traces generated includes information about the elapsed time of:

- data transfers between host and FPGA memory,
- the execution of the programmer FPGA task,
- taskwait and critical synchronizations
- task creation overhead

To be able to instrument the code, a specific port has been added to the top function (wrapper) at compile time. This port must be passed through each function as a new parameter to be able to write trace events to specific trace execution buffers from any point of the code. Listing 3.24 shows the HLS wrapper generated with the new instrumentation port and an excerpt of the code to figure out if the execution should or not generate an execution trace.

```
void foo_wrapper(... stream<__mcxx_instrData_t>& mcxx_instr) {
...
  if (__command(7,0) == 2) {
    __mcxx_instrData_t tmpSetup;
    tmpSetup(63,0) = mcxx_inPort.read();
    tmpSetup(79,64) = (__command>>8)&0xFFFFFF;
    tmpSetup[104] = 0;
    mcxx_instr.write(tmpSetup);
    return;
  }
  ...
```

```
  foo_moved(..., mcxx_instr);
  ...
}
```

Listing 3.24: Vitis HLS wrapper code generated by OmpSs@FPGA Clang with instrumentation port

Listing 3.25 shows an example of the Clang transformation of a critical pragma to support this critical (set and unset lock) and instrument it to measure its elapsed time (mcxx_instrument_event). The original source code (top) and the HLS transformed code (bottom), done by our framework, are shown.

```
// Input Programmer source code
#pragma oss critical
*dst = *src;
...
// Output of the Clang transformation
{
mcxx_instrument_event(85U, 2UL, mcxx_instr);
mcxx_set_lock(mcxx_inPort, mcxx_outPort);
mcxx_instrument_event(213U, 2UL, mcxx_instr);
}
*dst = *src;
{
mcxx_instrument_event(85U, 3UL, mcxx_instr);
mcxx_unset_lock(mcxx_outPort);
mcxx_instrument_event(213U, 3UL, mcxx_instr);
}
```

Listing 3.25: Vitis HLS wrapper code generated by OmpSs@FPGA Clang with instrumentation port

There is a trace execution buffer per accelerator so that there will not be data race conditions. Each trace event basically consists on the identifier of the task, flag indicating start and end point of the event, type of the event and clock time measured in the FPGA (and that will be processed thanks to a timestamp taken at the beginning of the application execution). Nowadays, the execution trace event follows the Obtuse but Versatile Nanoscale Instrumentation (OVNI) [OVNI 2023]. The execution trace generated of listing 3.25 is shown in figure 3.26. Figure 3.26 shows a Paraver [Paraver 2023] view of the execution trace along a period of time (x axis). Horizontal lines (only one shown) show the state of each of the threads or accelerators executed. Each state is shown with a different color. Figure shows the different steps of the FPGA task execution, and in particular, the execution time of the data transfer from the host memory to the FPGA memory (brown), the FPGA task execution (green), the set lock (white), again green to execute another part of the FPGA task, the unset lock (red), FPGA task execution (green) and finally, the data transfer from the FPGA to the host (violet).

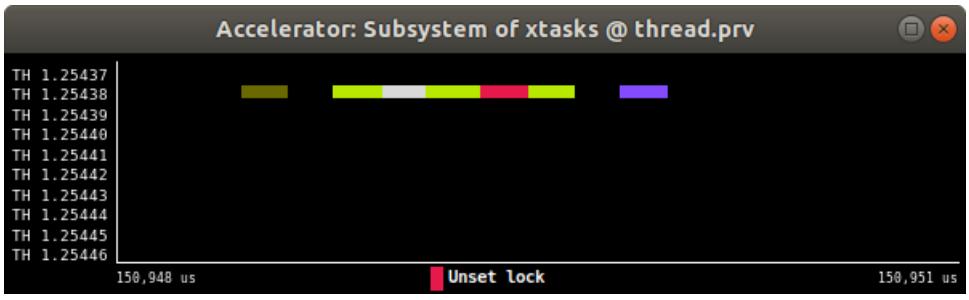


Figure 3.26: Paraver view of an execution trace. Critical example shown in listing 3.25.

Hardware instrumentation also allows the programmer to understand the performance execution of several nested and non-nested FPGA tasks running in parallel.

3.7 Custom IPs for inter-communication in FPGA-based clusters

In this section we explain the IPs and the protocols developed to support multi-FPGA clusters with OmpSs@cloudFPGA [Haro 2022]. This is an extension of the OmpSs@FPGA programming model that adds message passing to FPGAs, like the Message Passing Interface (MPI) in CPU clusters. The programmer can use a similar API in the HLS code, to do simple send/receive operations and a few collectives, like broadcast. This message passing model is called OmpSs MPI for FPGAs (OMPIF). Another contribution of OmpSs@cloudFPGA to classic OmSs@FPGA is a new type of task called distributed. This type of task is created by the CPU, and the software runtime replicates it to every FPGA in the cluster. I.e. with a single call to a distributed task, all nodes start executing the code associated with it, and use the cluster rank and size like in a regular MPI program. The programming model is discussed in detail in deliverable 4.6, where we show some code examples and give performance and power results on different clusters.

3.7.1 OMPIF runtime architecture

The message passing runtime implements a custom protocol based on a stream interface. A message is split in chunks of a fixed size. This is needed to adapt to the actual transport layer, which usually limits the size of a network packet. For example, most ethernet networks limit the packet size to 1500 bytes. Therefore, the message sender has a parameter to control the amount of data sent to the transport layer. Besides that, it appends a custom header with the protocol information. This protocol supports packet reordering and packet loss, because we were working with UDP in cloudFPGA, and ethernet later in MEEP. The specification of the source and destination ranks depends on the FPGA shell. We discuss this topic further in section 4.2.3. Therefore, the message sender expects to receive acknowledgement messages inside a time window, if the ack is not received, the module sends it again.

In the receiving side, there are two components, the packet decoder and message receiver. The packet decoder reads all incoming packets from the FPGA shell and converts them to the expected OmpSs@cloudFPGA format. It also decides where to forward the packet depending on the type. Ack messages go to the message sender, while data messages (produced by the message sender), go to a DMA engine and then to an intermediate buffer in main memory. We need the intermediate buffer because the packet decoder doesn't know the final address. Moreover, source and destination FPGAs are not synchronized, so when the message reaches its destination, the message receiver may not have the address yet. When this one is ready, it reads from the intermediate buffer and moves the message to the user space.

This protocol is illustrated in figure 3.27, where we can see on the left how the message sender of FPGA 0 reads from memory and sends the data through the network. The packet decoder writes to memory through the DMA engine. Then, on the right we see the ack message produced by the packet decoder, and the message receiver moving data to the user-provided address in the OMPIF_Recv call.

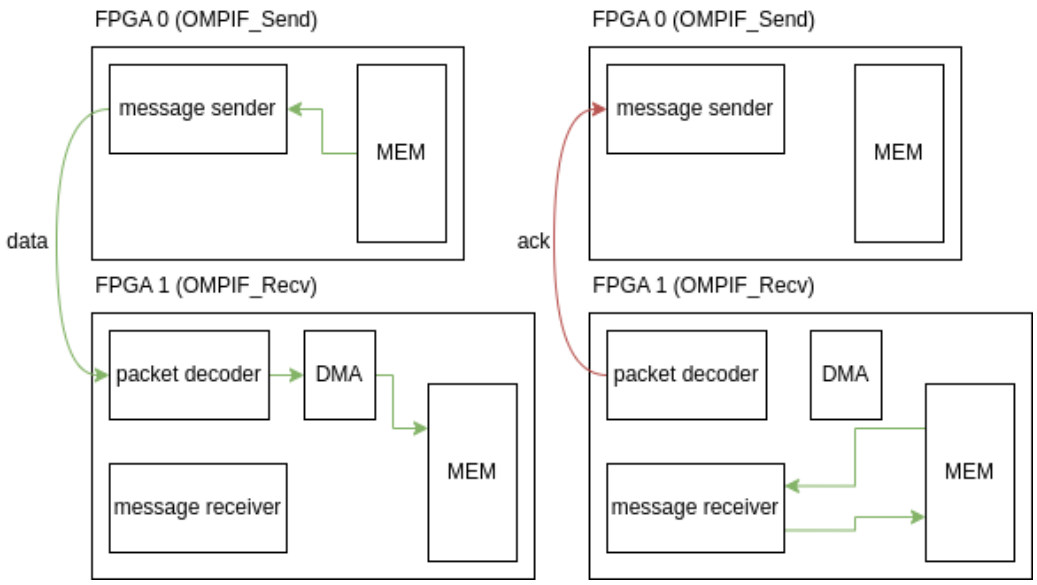


Figure 3.27: OMPIF send and receive process

3.7.2 CloudFPGA cluster

We implemented the first prototype of the OmpSs@cloudFPGA model in the cloudFPGA cluster of IBM research Europe. It has 56 AMD Kintex UltraScale FPGAs, connected in a 10Gb ethernet network. In the data center, FPGAs are installed as standalone nodes, thus there are no CPU hosts attached with PCIe like in most systems. Instead, FPGAs communicate only via TCP/UDP protocols. For that, they created a shell inside the cloudFPGA project. This shell implements the TCP/UDP stack and the layers beneath, like ethernet. It provides a standard interface to the user, based on AXI-stream with meta data including the UDP/TCP ports and source/destination ranks. These ranks identify every FPGA in a cluster, including CPU nodes. Therefore, we reuse these identifiers for OMPIF, however we have to translate them because in our current implementation CPU nodes are not counted.

3.7.3 MEEP cluster

We are currently working on an implementation of this model on our own FPGA cluster at BSC, with 96 Alveo U55C cards. In this cluster we don't have a shell with UDP/TCP support, but we don't need it because the FPGAs are connected to a private 100Gbe network. Therefore, we decided to work directly on the ethernet layer. For that, we have to implement IP bridges between OmpSs@cloudFPGA designs and the ethernet physical layer.

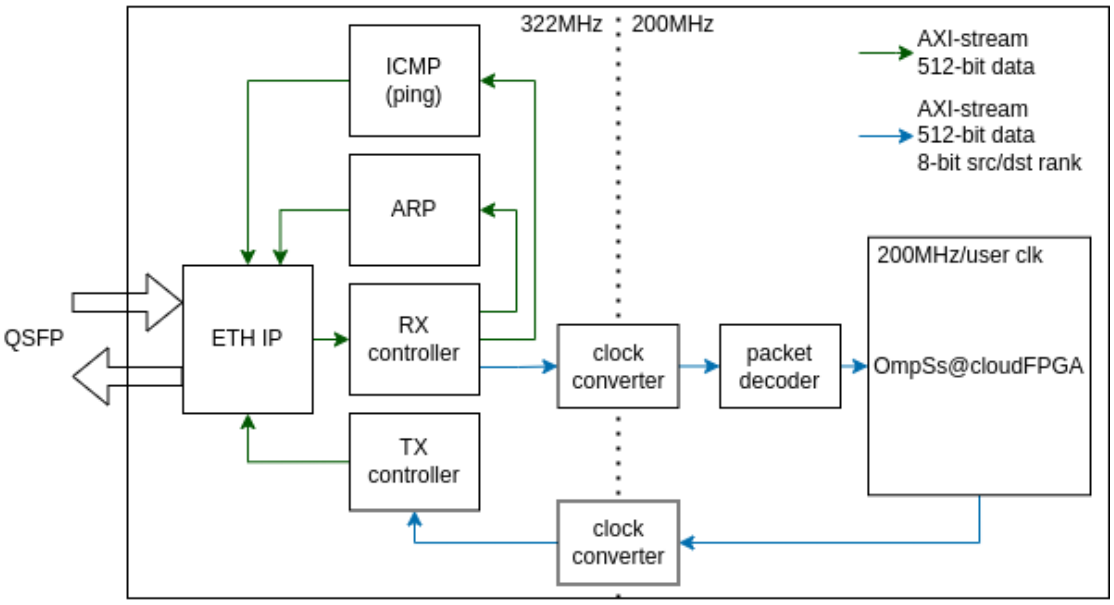


Figure 3.28: U55C ethernet shell

This custom shell communicates with the UltraScale+ Integrated 100Gb ethernet subsystem IP. The IP connects to a QSFP port and provides AXI-stream interfaces to send and receive data. We implemented several modules to adapt our interface with this IP, illustrated in figure 3.28. The ethernet IP implements the CRC generation and checking of an ethernet frame, but the user must add the rest of the header. The RX and TX controllers add and remove this ethernet header. The TX controller uses the destination rank of the input AXI-stream to select the MAC address from a table, which can be modified at runtime by the CPU. The RX controller uses this table to search the source rank from the source MAC address. Another role of the controllers is to ensure the constraints of the ethernet IP streams are satisfied. On the receiving side, RX stream doesn't have a ready signal, so when data is received, the RX controller is always ready to consume it or drop the entire frame. To do that, it has an internal queue that stores full frames, if the queue becomes full before the last word of data, it drops the whole frame. Moreover, this queue also discards frames that are corrupted. This information is forwarded by the ethernet IP, that marks an error bit in the last word when the CRC code doesn't validate. On the TX stream side, although there is a ready signal, the specification states that the valid signal must be always asserted from the beginning to the end of a frame. This means we have to send the packets at full bandwidth. To ensure that, another queue in the TX controller waits for a full frame before sending it to the ethernet IP.

Furthermore, the RX controller filters messages that are not recognized by looking at the ethernet type field of the ethernet header. It only allows ICMP, ARP and OMPIF packets. For the ICMP and ARP protocols, there are dedicated servers implemented in SystemVerilog. ICMP is used to respond to ping requests, limited to 64 bytes of payload. For ARP, the server responds to IPv4 requests. We implemented these protocols to check network connectivity and latency from the CPU side, since the servers are also connected to the FPGA network.

In the figure we can also see there is a clock domain crossing between the ETH IP and the OmpSs@cloudFPGA application. The 322MHz frequency is fixed by the IP itself, but to reach 100Gb at 512 bits per cycle, we only need 195MHz. However, the clock generator has a limited resolution and the closest frequency we can get is 200MHz.

3.8 Power Sampling Support

To analyze how OmpSs@FPGA features regarding placement, memory interleave and priorities, etc. affect power consumption of different applications, we have developed an automatic method to query power usage statistics from FPGA board power delivery infrastructure [Filgueras 2023b]. To do so, we integrated a Card Management Subsystem (CMS) into the design static logic instantiated by the tools. A simplified diagram is shown in figure 3.29. The CMS module polls an external microcontroller in the card (µC block in figure 3.29) to get readings from the voltage regulators and sensors in the card. We can also get temperature readings, fan speed, voltage and current for different power supply rails. To compute power consumed by the full card, we sum all card power inputs: PCIe connector power and PCIe external power.

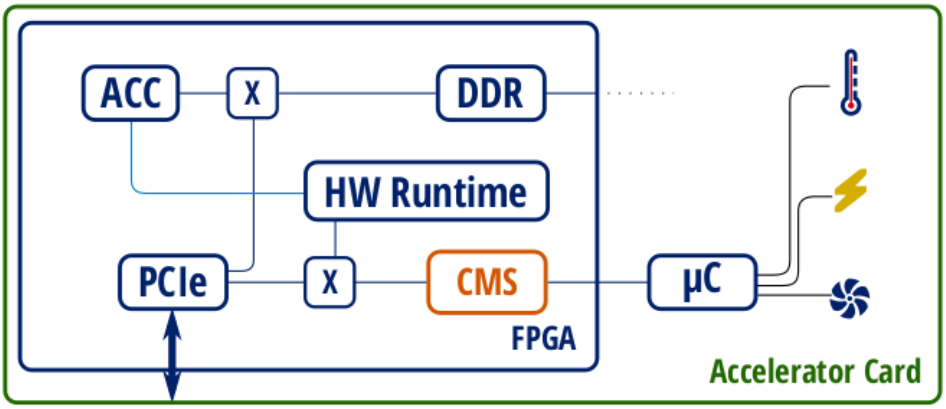


Figure 3.29: Design diagram showing power measurement components in a design

However, instantiating this module into a design that is close to the practical limit of the device resources, often causes implementation to fail due to routing congestion and timing issues. In order to work around this issue, we implement the design at lower frequencies and then extrapolate power consumption from multiple data points. As shown in figure 3.30, gathered data shows that power grows linearly with frequency. This figure shows the relationship between power and frequency for the different evaluated applications. Each data point shows power measured at a certain frequency and trend lines are shown for each of the different evaluated designs. For each application, baseline (solid line) and improved (dotted line) designs are measured separately. Applications correspond with designs described in table IV running at different frequencies.

It is worth highlighting that all data points are very close to their regression line, more precisely, average R^2 is greater than 0.995. Therefore, extrapolated data should match actual measurements with very little error.

Figure 3.30 shows results for four different applications:

- MM-half, single and double: Matrix multiply half, single and double precision, respectively. Matrix multiplication is a well-known embarrassingly parallel application. The application computes C = C + A × B, being A, B and C matrices of size N × N.
- Cholesky: This benchmark performs the Cholesky decomposition of a real Hermian positive definite matrix A into a lower triangular matrix L. Multiplying L by its transpose, results in the

original matrix A = L × L T. In the same fashion as the matrix multiplication kernel, the input matrix A is distributed in square blocks of size BS × BS single-precision elements.

- N-body: The N-body simulation computes how a group of particles with different masses interact with each other due to gravitational forces over a period of time. Algorithm input is a set of particles, each one consisting of an initial position, mass, and initial velocity. Position and velocity are 3-dimensional single precision floating point vectors, while mass is a scalar value. The output of the algorithm is the set of particles with their positions updated due to gravitational interactions after a given amount of time steps.
- Spectra: The Spectra application computes a histogram of electronic weights between particles versus distance for a given set of particles. To do so, it needs to compute the distance between each pair of particles and then add their electronic weight to the histogram. The histogram is afterwards used to compute the X-ray spectrum of the physical material being analysed allowing the determination of the material composition [Gonzalez 2022].

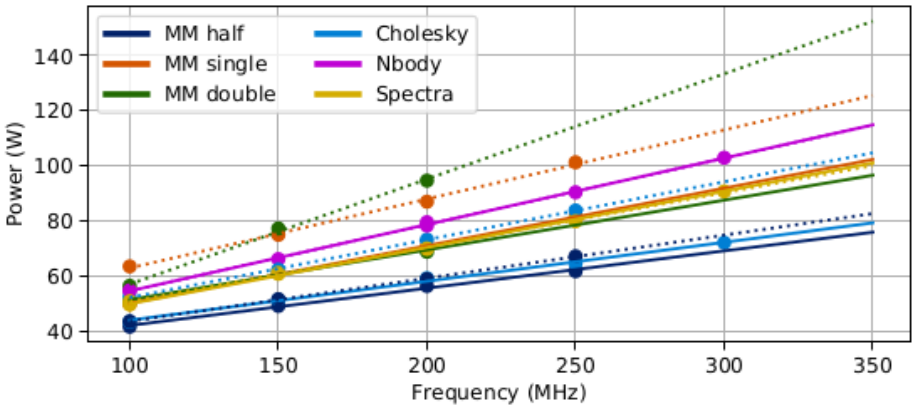


Figure 3.30: Power/frequency plot and trend lines for all evaluated applications

3.9 References

[OmpSs-2 2023] 2023. OmpSs-2. [Online] (last visit Nov 6th 2023) https://pm.bsc.es/ompss-2

[OVNI 2023] 2023. Obtuse but Versatile Nanoscale Instrumentation. [Online] (last visit Nov 6th 2023) https://ovni.readthedocs.io/en/master/

[Paraver 2023] 2023. Paraver: a flexible performance analysis tool. [Online] (last visit Nov 6th 2023) https://tools.bsc.es/paraver

[Gonzalez 2022] C. Gonzalez, S. Balocco, J. Bosch, J. M. de Haro, M. Paolini, A. Filgueras, C. lvarez, and R. Pons, "High performance computing pp-distance algorithms to generate x-ray spectra from 3d models," International Journal of Molecular Sciences, vol. 23, no. 19, 2022. [Online]. Available: https://www.mdpi.com/1422-0067/23/19/11408

[Filgueras 2022] Antonio Filgueras, Daniel Jimenez-González, Carlos Álvarez: Improving resource usage in large FPGA accelerators. 9th BSC Doctoral Symposium Book of Abstracts. 2022

[Filgueras 2023] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell: Improving Performance of HPC Kernels on FPGAs Using High-Level Resource Management. FCCM 2023: 213

[Filgueras 2023b] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell:

FPGA framework improvements for HPC applications. FPT 2023, accepted, to be published in December 2023.

[Haro 2021] Juan Miguel De Haro Ruiz, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta: OmpSs@FPGA Framework for High Performance FPGA Computing. IEEE Trans. Computers 70(12): 2029-2042 (2021)

[Haro 2022] Juan Miguel De Haro Ruiz, Rubén Cano, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, François Abel, Burkhard Ringlein, Beat Weiss: OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing. IPDPS 2022: 828-838

4 The APEIRON framework for High Level Programming of Streaming Applications on Multi-FPGA Systems

The APEIRON framework, developed by the INFN APE Lab, aims at offering hardware and software support for running real-time dataflow applications on a network of interconnected FPGAs.
The main motivation for the design and development of the APEIRON framework is that the currently available HLS tools do not natively support the deployment of applications over multiple FPGA devices, which severely chokes the scalability of problems that this approach could tackle. To overcome this limitation, we envisioned APEIRON as an extension of the Xilinx Vitis framework able to support a network of FPGA devices interconnected by a low-latency direct network as the reference execution platform. Developers can define scalable applications, using a streaming programming model inspired by Kahn Process Networks, that can be efficiently deployed on a multi-FPGAs system: the APEIRON communication IPs allow low-latency communication between processing tasks deployed on FPGAs, even if they are hosted on different computing nodes. Thanks to the use of HLS tools in the workflow, processing tasks are described in C++ as HLS kernels, while communication between tasks is expressed through a lightweight C++ API based on non-blocking *send()* and blocking *receive()* operations.
In this section we provide a general overview of the framework, describing its main components and an example of APEIRON application. For a more detailed description of the communication IPs and of the framework software stack please refer to deliverables D2.9 - IP for low-latency inter-node communication links, part 2 and D4.5 - Inter-FPGA Communication SW Stack.

4.1 Introduction

Using APEIRON developers have the capability of deploying scalable applications on a multi-FPGAs system via a streaming programming model inspired by Kahn processing networks.

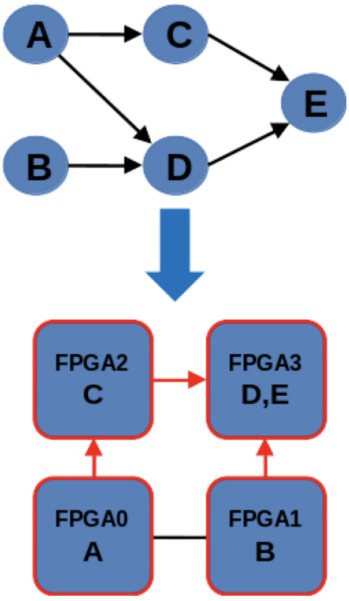


Figure 4.1: Application's dataflow graph mapped on 4 interconnected FPGAs system

This allows the straightforward mapping of application's computational dataflow graph onto the underlying execution platform consisting in a network of FPGAs (Fig. 4.1) via a simple configuration tool that instructs the framework to create all the files required for the FPGA bitstream generation and to automatically build the application interconnection logic (Fig. 4.2).

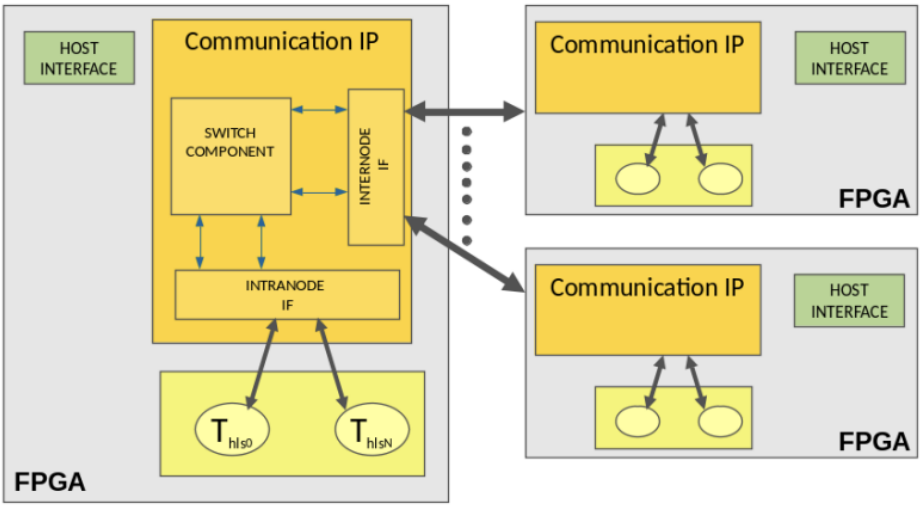


Figure 4.2: APEIRON interconnection logic: Communication IPs managing data streams I/O and communication between HLS computing tasks (represented as yellow ovals).

Processing tasks are implemented in C++ thanks to the use of the Xilinx Vitis High Level Synthesis tool and deployed on the different FPGA nodes. Communication between tasks is expressed through a lightweight C++ API (called HAPECOM) based on non-blocking send() and blocking receive() operations and is implemented in the multi-FPGA execution platform by the APEIRON communication IP (Fig. 4.2).

Our design idea was motivated by the following considerations:

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

These considerations are strictly related to the TEXTAROSSA project objectives:

- Objective 1 - Energy efficiency. APEIRON addresses this objective enabling the complete offload of the streaming processing to FPGA devices [Qasaimeh2019, Nguyen2020, Goz2020]. Furthermore, avoiding the involvement of the CPUs and system bus resources in data transfers improves the energy efficiency of the multi-FPGA execution platform.
- Objective 2 - Sustained application performance. The sustained application performance of distributed streaming applications, such as the RAIDER use case, are strongly affected by the performance of the network system. Implementing a direct FPGA to FPGA interconnect and bypassing the host network stack, allows to keep the communication latency in the sub-microsecond range and to increase the bandwidth for small messages.
- Objective 4 - Seamless integration of reconfigurable accelerators. The APEIRON framework leverages the Vitis HLS workflow, extending it to a multi-FPGA execution platform through a lightweight communication library (HAPECOM) at programming level, and through a simple configuration system for the deployment of the distributed application to the multi-FPGA execution platform.
- Objective 5 - Development of new IPs. The INFN Communication IP is the key enabling technology behind the APEIRON framework, allowing direct low-latency intra/inter FPGA communications between HLS kernels.
- Objective 6 - Integrated Development Platform. The ARMv8 based IDV-E represents the reference execution platform for the APEIRON runtime in the TEXTAROSSA project. Nevertheless, the framework has been developed and extensively tested on a X86_64 based small cluster in our lab.

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

- Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The APEIRON framework software is developed following the open-source model and is freely available in its GitHub repository (https://github.com/APE-group/APEIRON).
- Strategic Goal #3: Opening of new usage domains. The APEIRON frameworks aims at offering hardware and software support for running real-time dataflow applications on a network of interconnected FPGAs, leveraging on the Vitis HLS tool. We believe that it has the potential to ease the development and to support the efficient execution of a wide class of applications suited to be executed on a multi-FPGA platform, such as but not limited to real-time HPDA ones.

4.2 APEIRON Building Blocks

The APEIRON main components are the Communication IP, implementing the low-latency direct network between FPGAs, and the software stack (runtime support, HAPECOM C++ communication library, tools for automatic linking of computational tasks and IPs for bitstream generation). The former is thoroughly described in deliverable D2.9 - IP for low-latency inter-node communication links, part 2 and the latter in deliverable D4.5 - Inter-FPGA Communication SW Stack, here we provide a short overview of both.

4.2.1 Communication IP

The Communication IP represents the main enabling component of the APEIRON framework and is based on the HPC direct network designs previously developed by our research group, like APEnet [1] and ExaNet [2].
The Communication IP implements within the framework a direct network which allows low-latency data transfer between processing tasks deployed on the same FPGA (intra-node communication) and on different FPGAs (inter-node communication). These processing tasks are implemented as HLS kernels; the details of their definition in the framework and of their interface with the IP are presented in Sec. 3.2.1.

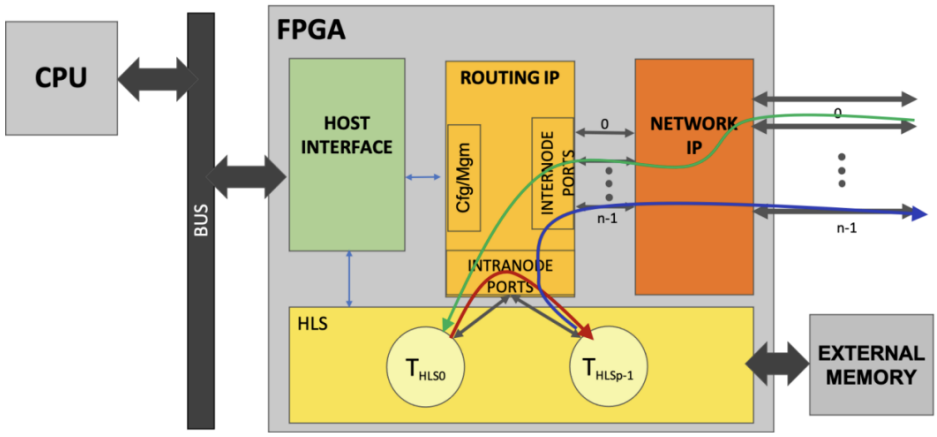


Figure 4.3: Communication IP hardware block structure with HLS kernels performing intra-node (red line) and inter-node (green line – receive, blue line– send) communications.

Figure 4.3 shows the Communication IP hardware block structure, containing a Network IP and a Routing IP, both developed in VHDL for Xilinx Alveo U200 and U280 cards.
The Routing IP defines the switching technique and routing algorithm and consists of the Switch component, the Configuration/Status Registers and the InterNode and IntraNode interfaces.
The Switch component dynamically interconnects all ports of the IP, routing 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 solves contentions between packets requiring the same port. For inter-node communications, the routing policy applied is the dimension-order (DOR) one: each FPGA is uniquely identified by its coordinates in a N dimensional torus, the DOR consists in reducing the offset along one dimension to zero before considering the offset in the next dimension in antilexicographic order. The employed switching technique (i.e., when and how messages are transferred)

is Virtual Cut-Through (VCT): 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. The transmission is packet-based: the Communication IP sends, receives and routes packets with a header, a variable size payload and a footer.

4.2.2 Software Stack

The APEIRON software stack includes three main components: 1) runtime support, 2) HAPECOM C++ communication library, and 3) tools for automatic linking of computational tasks and IPs for bitstream generation.

4.2.2.1 Runtime Support

The APEIRON framework currently supports Xilinx Ultrascale PCIe-based accelerator cards. We designed a runtime software stack based on the Xilinx Runtime (XRT) architecture, which is implemented as a combination of user-space and kernel driver components [3].

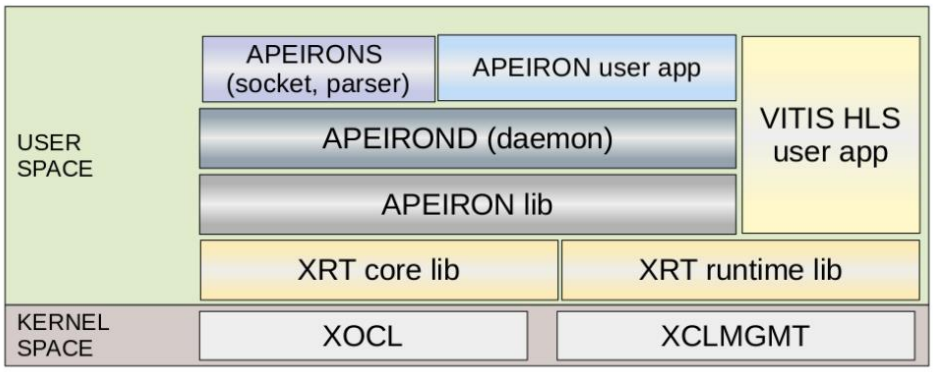


Figure 4.4: APEIRON Software Stack

The APEIRON runtime software stack is built on top of the XRT one, adding three layers as shown in Figure 4.4, to:

• add the functionalities required to manage multiple FPGA execution platforms (e.g., program the devices, configure the IPs, start/stop execution, monitor the status of IPs, ...);

• eliminate, or at least reduce, the impact of changes in XRT API introduced with any new version of Vitis on the APEIRON host-side applications;

• decouple the APEIRON software stack from the specific platform, easing the future porting of the framework to different platforms/vendors, ideally by extending the APEIRON library layer only.

Apeirond is a persistent daemon used to manage multiple access requests from user apps to the board. It uses functions exposed by the APEIRON library to operate on the devices. Apeirond module accepts client connections over a network socket (using the module called apeirons) and oversees creating the socket with the client and handling the incoming command (e.g., reading a register or flashing the board). The protocol currently used is based on a TCP/IP socket while messages are serialized and deserialized in JSON format to simplify the parsing phase.

4.2.2.2 HAPECOM Communication Library

The communication between kernels is expressed through HAPECOM: a lightweight C++ API based on non-blocking send() and blocking receive() operations. This simple API allows the HLS developer to perform communications between kernels, either deployed on the same FPGA (intra-node communication) or on different FPGAs (inter-node communication) without knowing the details of the underlying network stack. The Communication Library leverages AXI4-Stream Side-Channels to encode all the information needed to forge the packet header. Two APEIRON HLS IPs defined in the library manage the adaptation toward/from IntraNode ports of the Routing IP: they are Aggregator and Dispatcher, as shown in Figure 4.5.

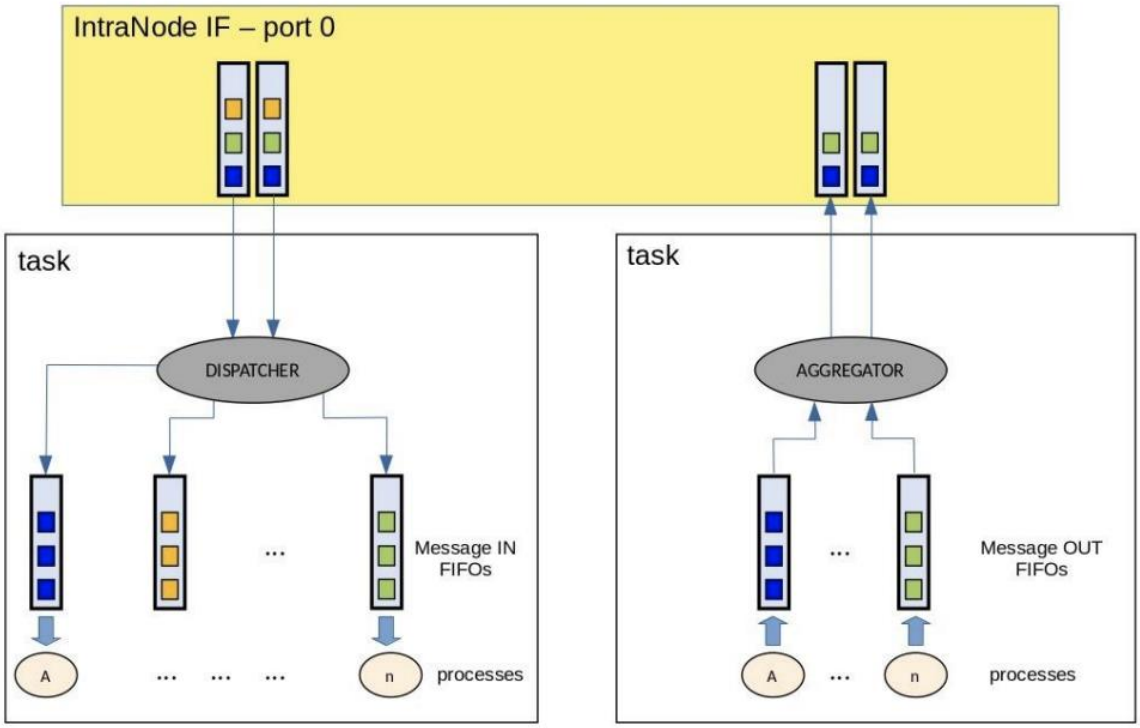


Figure 4.5: Interface between Intranode Port 0 and the corresponding HLS Task mediated by Aggregator and Dispatcher.

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, then filling the header/data FIFOs of the Routing IP IntraNode port.

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

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

Where:

- `msg` is the message to be sent and `size` its size in Bytes;
- `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);
- `recv_buf` is the receive buffer of the destination HLS kernel.

4.2.2.3 Automatic Linking of Computational Kernels and Communication IP for Bitstream Generation

Users must prepare a YAML configuration file describing the attributes of each HLS computational kernel (number of input and output channels, IntraNode port of the Communication IP to which the kernel is connected). Starting from this, 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 in order to be linked to the framework is in the format of their prototype that must adhere to this form:

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

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

If the instantiation interval on the receiving side must be kept low to maximize the design throughput, it is not advisable to use the blocking `receive()` function, and the direct access to the `message_data_in` stream through the `read()` method must be used instead, parallelizing data reads and processing, as shown in listing 4.3.

4.3 Latency and Bandwidth of Communications between HLS Tasks in the APEIRON Framework

We developed a set of APEIRON applications to assess the performance (i.e. end to end latency and one-way bandwidth) of communications between HLS kernels under different conditions, here we report the more relevant ones:

- Intra-node: communicating tasks deployed on the same FPGA.
- Inter-node: communicating tasks deployed on different FPGAs (1 hop distance).
- DDR+sync/BRAM: send and receive buffers hosted on DDR or BRAM memory. For the DDR also the “sync” operation to ensure data coherence between CPU and FPGA is taken into account.

Using the most performant configuration of the Communication IP (256 bit datapath, 200 MHz clock frequency) we have measured the performance reported in Table 4.1 (for latency) and Table 4.2 (for bandwidth). For latency, the reported measurements were collected with 16B payload packets, while for bandwidth we used 4kB payload packets.

	DDR+Sync (ns)	**BRAM (ns)**
Intra-node	533	213
Inter-node	1065	768

Table 4.1: Communication Latency Between HLS Tasks in APEIRON (256 bit, 200 MHz)

	DDR+Sync (MB/s)	**BRAM (MB/s)**
Intra-node	3938	5967
Inter-node	3937	4658

Table 4.2: Communication Bandwidth Between HLS Tasks in APEIRON (256 bit, 200 MHz)

For the complete set of measurements, the detailed description of the APEIRON applications and a reference to their source code please refer to D2.9 - IP for low-latency inter-node communication links, part 2.

4.4 Example of APEIRON Application

We adopt a preliminary version of our RAIDER application as use case of the APEIRON framework, highlighting the steps needed to scale a standard Vitis HLS application towards a multi-FPGA implementation. RAIDERS's task is to perform particle identification (PID) on the stream of events generated by the RICH (Ring Imaging CHerenkov) detector in the CERN NA62 experiment at a rate of about 10 MHz, using neural networks.

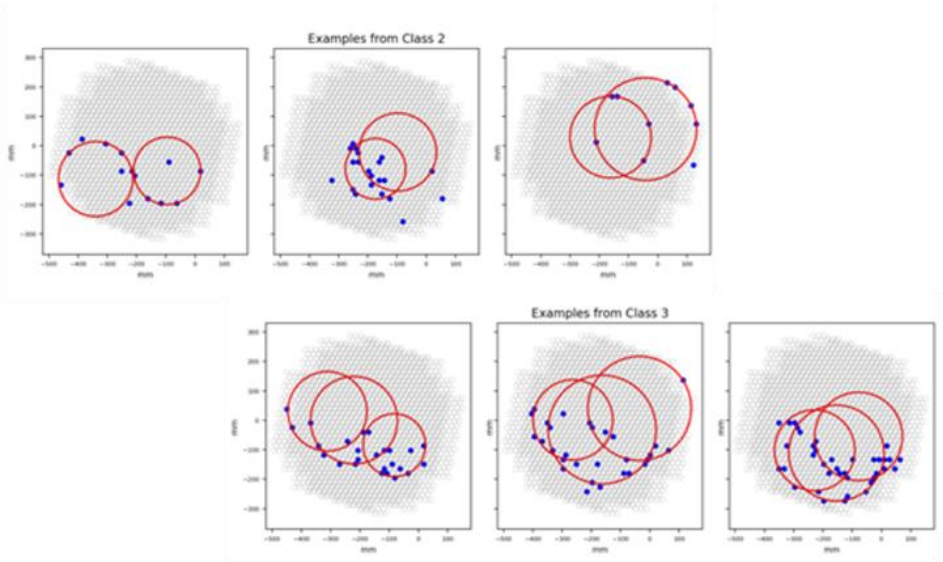


Figure 4.6 Examples of events belonging to class 2 and 3 (2 or >=3 charged particles) as detected by the array of RICH photomultipliers (blue dots are the hit photomultipliers, red circles are the tracks reconstructed offline by the NA62 experiment offline analysis software framework)

The inference task consists in providing an estimate for the number of charged particles (0, 1, 2, >=3) for any RICH detector event, that corresponds to the number of ring tracks that can be reconstructed from the pattern of photomultipliers that have been illuminated (hit) by the Cherenkov light cone emitted by a charged particle traversing the detector, as shown in Figure 4.6. The inference task is implemented with a preprocessing stage (*Imagifier*) and a Convolutional Neural Network (*CNN*). The CNN model has been developed using Tensorflow/Keras and deployed on FPGA with the HLS4ML [4] software package, refer to deliverable D4.8 - Framework for efficient CNNs inference on a TEXTAROSSA node for a complete description of this workflow. The CNN receives in input the output of the Imagifier kernel, a 16x16 image of the hit photomultipliers (PMTs) map for each physics event and produces an estimate for the number of charged particles it contains. Considering the high event rate of the experiment, sustaining an adequate processing throughput is the main challenge for such a system.
In deliverable D6.2 - Initial Application Benchmarks and Results we reported results obtained on two single-FPGA implementations of the application, including one and two inference pipelines respectively. Here we scale the number of Xilinx Alveo U200 FPGAs from 2 to 4, in order to increase further the reconstruction throughput, deploying the HLS processing tasks according to what is shown in Figure 4.7.

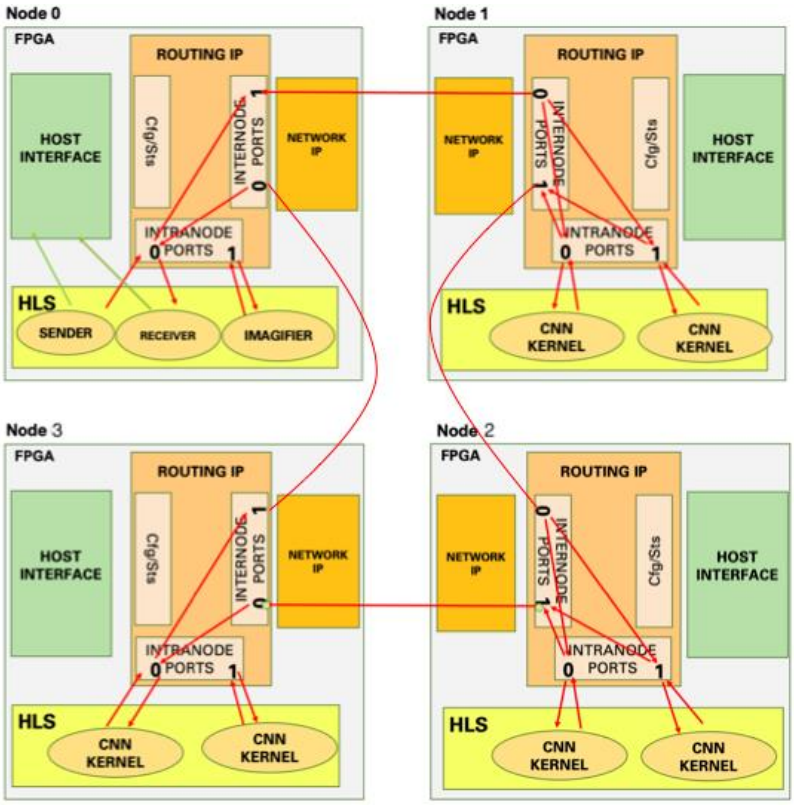


Figure 4.7 RAIDER HLS processing tasks deployed on the 4 FPGAs execution platform.

As the Figure 4.7 shows, there are two kinds of nodes (and hence the overall Multi-FPGA design includes two different bistreams):

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

Since for each type of node we need a different bitstream, two different YAML configuration files are needed for APEIRON to generate the firmware to be flashed on each kind of node. These two files are reported below:

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

```
    switch_port: 0


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


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


config:
  freq: 100
  links: 2
```

Listing 4.1: “Preprocessing node" YAML configuration file

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


 -name: cnn_kernel
    input_channels: 1
    output_channels: 1
    switch_port: 1


config:
  freq: 100
  links: 2
```

Listing 4.2: “Preprocessing node" YAML configuration file

The file format includes two main sections: the kernels section lists the number of computing kernels in the design by name, indicating the IntraNode port of the Communication IP they are connected to and the number of I/O channels they use, the config section is used to specify the number of InterNode ports and the target clock frequency for the synthesis of the overall design.

Note that since this version of the application is based on the preliminary version of the Communication IP, described in deliverable D2.8 - IP for low-latency inter-node communication links, part 1 , the clock frequency of the overall design to be synthesized is set to 100 MHz.

```
extern "C" void imagifier (unsigned int nports, unsigned int nboards,
                           message_stream_t  message_data_out[N_INPUT_CHANNELS],
                           message_stream_t  message_data_in[N_OUTPUT_CHANNELS]) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=message_data_in
#pragma HLS interface axis port=message_data_out

        static unsigned ch_id = 0;
        word_t word;
        word_t buff_out[5];
        ap_uint<IMAGE_SIZE*IMAGE_SIZE> image = 0;
        size_t size=0;
        for (int i=0; i<MAX_WORD; i++) {

            //Streaming events reception ==> reading of TEXTAROSSA and event headers
            if(size==0){
                   auto hd = message_data_in[ch_id].read();  //TEXTAROSSA header
                   size = hd.range(size_start_bitpos,size_end_bitpos)/sizeof(word_t);
                   buff_out[0] = message_data_in[ch_id].read(); //event header
                   size--;
            }
            if(size>0) word = message_data_in[ch_id].read();

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

            if (size==0){
                auto ftr = message_data_in[ch_id].read();
                break;
            }
        }

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

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

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

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

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

Listing 4.3: Imagifier HLS kernel

Starting from the I/O and Preprocessing node, the "imagifier" HLS kernel is reported in Listing 4.3. From interfaces defined with Vitis pragmas (in particular "`#pragma HLS interface ap_ctrl_none port=return`"), we can notice that this is defined as a free-running kernel: a kernel which starts with the bitstream loading on the device, without any call by the CPU host (which is required by "sender" and "receiver" kernels, instead).

The "imagifier" works on packets of data coming from the network with the TEXTAROSSA communication protocol, each of them corresponding to a single physics event. To increase the overall design throughput, and so to work in streaming mode, we decided to not use the HAPECOM receive() API by directly reading data from input channels with the Vitis read() function. However, in this way, to have the packet size information, we must access to a certain bit address of the TEXTAROSSA header bounded by `size_start_bitpos` and `size_end_bitpos` enumerations. After that, we proceed with the receiving of the single event header, which has the information relative to the number of words composing the event and the event timestamp (as can be seen in Figure 4.8), and then we work on each event word to obtain the PMT hitlist and to convert it to a 16x16 image.

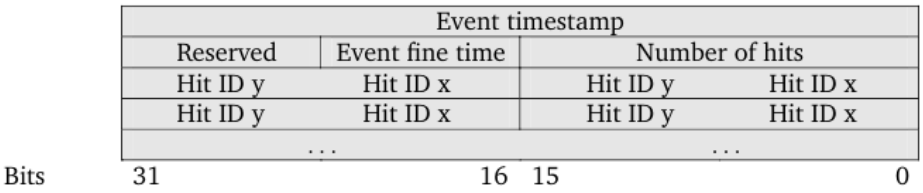


Figure 4.8 Structure of events data coming from NA62 RICH readout boards. Since in the RAIDER application we work on 128 bit words, we refer as "event header" to the fields: event timestamp, number of hits, event fine time, and reserved.

As last step of the preprocessing, this image (and the event timestamp) is sent via HAPECOM send() API to one of the Computing nodes in a "round robin" way, selecting each CNN kernel of each node as possible destination.

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

    hls::stream<input_t> nnet_input;
    hls::stream<result_t> nnet_output;
    hls::stream<ap_axis<128,0,0,0>> stream_timestamp;

    read_from(message_data_in, nnet_input, stream_timestamp);
    hwfunc(nnet_input, nnet_output);
    get_class(nnet_output, message_data_out, stream_timestamp);
}
```

Listing 4.4: "cnn_kernel" HLS code

As the main component of Computing nodes of the setup, the HLS code of "cnn_kernel" is reported in Listing 4.3. From interfaces, we can notice that this is defined as a free-running kernel (as for "imagifier" one) and it is composed by different task functions pipelined via HLS dataflow Vitis pragma. This allows

functions to overlap in their operation, increasing the overall throughput of design by increasing concurrency of the RTL tasks implementation. In detail:

- "read_from" receives packets from the networks, extracting information to be streamed as CNN input (nnet_input, 256 bit image) and to be streamed to the get_class() function to set the timestamp to the processed event (stream_timestamp);
- "hwfunc" is the task in which the FPGA implemented CNN (obtained from the HLS4ML framework) processes streaming input images;
- "get_class" receives CNN output and extract the predicted ring class. This is timestamped and sent through the network via the HAPECOM send() API.

We have scaled the system from 2 nodes (one I/O and preprocessing and one computing) up to 4 increasing the number of computing nodes as shown in Fig.4.7 and measured the processing time per event and the integrated processing throughput of the system; results are tabulated for the former and plotted for the latter in Figure 4.8 (throughput is in millions of events per second, MHz in figure).

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

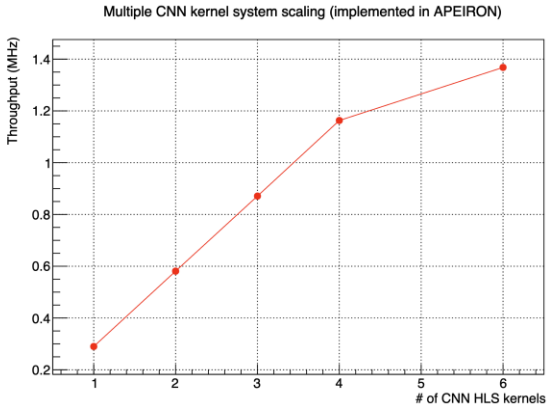


Figure 4.8: Scaling of processing time per event (left) and processing throughput (right) with the number of deployed CNN kernels

The presented results show the good scaling of system performance with the number of nodes. The flattening slope of the curve when the number of CNNs goes beyond 4 is mainly due to the saturation of the data injection rate in the "sender" on the Preprocessing node.
The co-design of APEIRON software stack along with its Communication IP allowed reaching very low and deterministic latency and a high fraction of the channel raw bandwidth for communications between FPGAs, addressing two fundamental bottlenecks for real-time distributed streaming applications at the same time, while allowing for a straightforward development and deployment of multi-FPGA HLS designs.

4.5 References

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

[2] R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. Paolucci, E. Pastorelli, F. Simula et al., The Next Generation of Exascale-Class Systems: The ExaNeSt Project, in

Proceedings - 20th Euromicro Conference on Digital System Design, DSD 2017, edited by M. Novotny, H. Kubatova, A. Skavhaug (IEEE, United States, 2017), pp. 510–515, 20th Euromicro Conference on Digital System Design, DSD 2017

[3] https://xilinx.github.io/xrt/master/html/index.html

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

5 TAFFO in the HLS flow

Approximate Computing is an increasingly popular approach to achieve large performance and energy improvements in error-tolerant applications [1,2]. This class of techniques aims at trading off computation accuracy for performance and energy. Within Approximate Computing, a key issue is to perform each computation on the most energy and performance-efficient data type that allows to preserve the desired minimum accuracy. This non-trivial task is usually performed manually by embedded systems programmers, and in general by software developers that need to achieve high performance with limited resources. However, this operation is error-prone and tedious, especially when large code bases are involved. Thus, a significant research effort has been spent over the recent years to build compiler-based tools to support or entirely replace the programmer effort [3].

TAFFO is an autotuning framework that aims at optimising the selection of data types in C and C++ programs, particularly by replacing floating point operations with equivalent ones based on other representations, including fixed point. TAFFO has been proven to enable major speedups on most error-tolerant classes of applications when targeting embedded microcontrollers that lack hardware support for floating point operations, but can also help improve performance and energy on more high-end platforms. TAFFO is implemented as a set of plugins for the industry-grade LLVM compiler framework, enabling its deployment in most modern systems. Additionally, this feature allows TAFFO to be seamlessly integrated within HLS tools that also are based on LLVM, such as AMD Vitis. In this section we introduce TAFFO, its architecture and means of operation, and how it can be used in an HLS flow in order to enable mixed-precision tuning on FPGA-based platforms.

5.1 Architecture of TAFFO

TAFFO tackles all the challenges of precision tuning [3], and it does so by using safe and deterministic static analyses. The user is expected to annotate the source code to provide information on the dynamic value range for the input variables and on the scope of the optimization. In general, to achieve the best results, the optimization scope should be a mathematically intensive computational kernel. The annotations to insert depends on the input data to the program, therefore the typical user of TAFFO is a domain expert who has access to the required information. A static data flow analysis propagates the value ranges to all the intermediate values in the program and determines the set of variables that need to be changed in type. This analysis is able to operate across function calls and loops if required.

Based on the fine-grained description of the dynamic value ranges produced by the data flow analyses, TAFFO performs the Data Type Allocation. For each dynamic value range, a constraint is derived on the data type such instruction can use. At this point, TAFFO determines the data type to assign to each variable automatically. Two different approaches are available for this task. The first one exploits a local best-fit algorithm that performs adjacent similar type coalescing. A customisable cost function can take into account the overhead introduced by type cast operations only, which varies depending on the target architecture. A second more complex algorithm [4] builds a partial mathematical model of the program that describes the variation in execution time and output error for a given architecture depending on the data type selection. This model is fed into an integer-linear-programming constraint solver to select the optimal data types for each variable that must be optimized. This new approach requires a more through architectural model and is more effective for embedded platforms, while the simpler local best-fit algorithm is more effective on superscalar architectures.

Once the data type has been decided, TAFFO performs the code conversion on the LLVM-IR. It automatically searches for instructions known to be equivalent for the target data type and -- if they exist -- replaces their use in the code. Whenever an equivalent instruction (or pattern of instructions) is not immediately available --- e.g. in the case of a function call to a generic function --- TAFFO implements a two-way best-effort approach. In case of called functions whose definition lives within the same file, TAFFO creates an ad-hoc version of the callee. Mathematical library functions with known behaviour --- e.g. sin and cos --- and no fixed point equivalent have their implementation generated on demand [5]. In the rest of unknown cases, TAFFO rejects the proposed data type and locally uses the original one. Appropriate type-cast operations are inserted if required.

Finally, TAFFO performs a functional and performance estimation of the conversion's benefits via static analysis techniques. It is worth to mention that both the cost function from the data type allocation, and the performance estimator from this last stage require an architectural model of the target machine. In absence of such model, TAFFO uses default values which may be suboptimal for the target architecture.

5.2 Structure of the Software

TAFFO is shipped as a set of LLVM passes, i.e. elementary units of the compiler pipeline. In particular, the execution flow runs through five stages: Initialization, Value Range Analysis, Data Type Allocation, Code Conversion, Feedback Estimator. The outline of the compilation pipeline of TAFFO is shown in Figure 5.1. The passes can be inserted in any point of the optimization pipeline, but usually they are executed immediately after the frontend. Other normalization and analysis passes are automatically scheduled by LLVM as required.

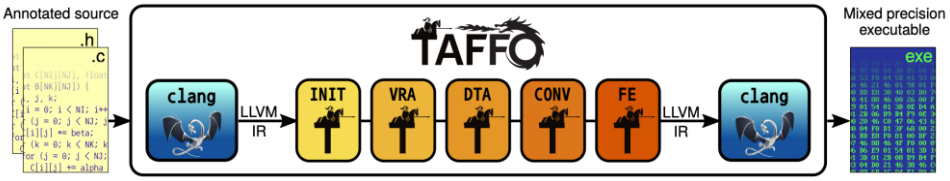


Figure 5.1 TAFFO compilation pipeline

The **Initialization** pass processes the user annotations, and determines the scope of the transformation which will be performed by later passes --- in other words which instructions are affected by the type change of the annotated variables. This process is performed through a reverse depth-first iteration in the data flow graph. Annotations can be placed on any variable declaration, and contain information about the value range of the variable itself, and additional directives that affect TAFFO's operation. An example of how these annotations appear in the source code is shown in Figure 5.2, which also shows the output LLVM-IR after the rest of the analyses and transformations performed by TAFFO.

```
int main() {
  float __attribute__((annotate("target('main')
      scalar(range(1, 100))"))) xs[len];
  ...
  for (int j = 0; j < len; ++j) {
    float x = xs[j];
    float y = ys[j];
    res[j] = sqrt((x * x) + (y * y));
  }
}
```

```
%vla.u7_25fixp = alloca i32, i64 10, !taffo.info !24,
    !taffo.target !27
      ...
%4 = zext i32 %u7_25fixp2 to i64
%5 = zext i32 %u7_25fixp2 to i64
%6 = mul i64 %4, %5
%7 = lshr i64 %6, 33
%mul27.u15_17fixp = trunc i64 %7 to i32
%8 = zext i32 %u7_25fixp to i64
%9 = zext i32 %u7_25fixp to i64
%10 = mul i64 %8, %9
%11 = lshr i64 %10, 33
%12 = trunc i64 %11 to i32
%u15_17fixp = add i32 %12, %mul27.u15_17fixp
      ...
!24 = !{!25, !26, i1 false, i2 1}
!25 = !{!"fixp", i32 32, i32 25}
!26 = !{double 1.000000e+00, double 1.000000e+02}
!27 = !{!"main"}
```

Figure 5.2: Illustration of how to use TAFFO annotations, and the resulting LLVM-IR produced by TAFFO.

The **Value Range Analysis** pass performs a fine-grained requirements analysis meant to augment the existing LLVM-IR with additional information based on Interval Analysis [6], and a simplified symbolic execution framework.

The **Data Type Allocation** pass decides which data type should be used for each intermediate value. It is designed to avoid overflow. Secondarily, it attempts to minimise accuracy loss and performance degradation due to type cast overhead. The data types supported by TAFFO are single and double precision floating point, bfloat16 and fixed point types of arbitrary size and point position. The set of types allowed in the optimized program can be adjusted at compile time.

The **code conversion and supplementary code generation** pass modifies the program code in order to enforce the usage of the data types determined by the previous passes of TAFFO. It preserves code semantics within the given value ranges. Whenever the conversion of an unknown external procedure is required, the data type is retained to the original version.

This **Error Propagation Pass** performs an additional code analysis which estimates the error in the output with respect to the original code. The error analysis is based on affine number theory [7,8] in order to handle cancellation errors.

5.3 Integration with AMD Vitis

Vitis is the platform provided by AMD for development of all software related to their hardware products, and it seamlessly integrates with Vivado, their hardware design toolchain. Within the overall Vitis toolchain, Vitis HLS is the industry-leading solution for High Level Synthesis. Vitis HLS integrates with Vivado in order to provide a seamless workflow for transforming C and C++ code into a hardware design suitable for AMD's FPGA offerings. While in the past Vivado and Vitis were bundled as a single product, AMD has modularized the overall toolchain, and open-sourced several of its parts, including the Vitis HLS frontend software. As the Vitis HLS frontend is based on LLVM, it is possible --- at least in principle --- to adapt TAFFO to behave like a plugin for Vitis HLS to enable mixed precision in HLS workflows. The overall compilation toolchain with the addition of TAFFO is shown in Figure 5.3.

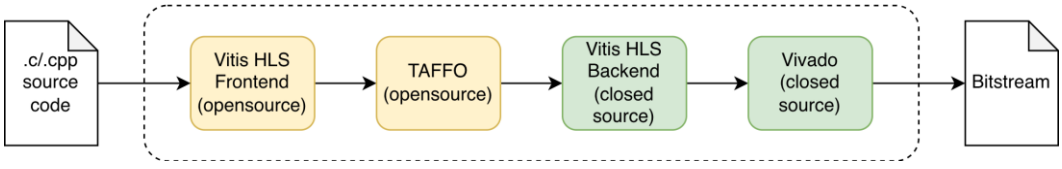


Figure 5.3 Mixed precision HLS compilation toolchain. Components that require modifications are shown in yellow, while other components are in green.

The integration of TAFFO with Vitis involves the insertion of TAFFO inside the standard Vitis pipeline, and also requires some modification to the Vitis frontend. The addition of TAFFO to the pipeline is performed by taking advantage of the Tcl scripting system provided by Vitis. The scripting system allows for insertion of additional compilation steps after the Vitis frontend, and we use this feature to insert TAFFO. The Vitis frontend itself needs to be adapted in order to ignore and preserve TAFFO annotations, and to produce debugging information that is compatible with upstream LLVM in order to enable the later processing of the IR by TAFFO. TAFFO itself is modified to enable the integration. First of all, TAFFO has been backported to LLVM 8, from LLVM 15, as Vitis still uses an out-of-date base version of the framework. Specifically, LLVM 8 has been released in 2019. Moreover, TAFFO needs to process correctly the additional metadata inserted by Vitis for guiding later backend stages. This was done by appropriately modifying the Initializer pass of TAFFO.

5.4 Using TAFFO with Vitis

Optimizing an HLS application with TAFFO can be performed by following a few simple steps. First of all, it is necessary to install TAFFO and Vitis from their respective websites. Currently we support the version of Vitis HLS included in Vivado 2023.1, without additional modifications, at the expense of non-complete debugging information available later in the workflow. The required version of TAFFO is a customized release, named "TAFFO-HLS", which will be available from the TAFFO website (https://github.com/TAFFO-org/TAFFO). Installing TAFFO also requires the installation of LLVM and Clang version 7.0.1 in a systemwide location.

After all the prerequisites have been installed, a mixed-precision synthesis of an HLS application with TAFFO is performed by creating and running custom Tcl script. A template for this script is shown in Listing 5.1. The template script needs to be customized for the specific use-case. In particular, the commands "open_project", "open_solution", "add_files", "set_top", "set_part" and "create_clock" are fully customizable. When using the Vivado IDE, it is possible to automatically create a suitable Tcl file from an existing project which then can be modified to add the TAFFO-specific commands: "set ::LLVM_CUSTOM_OPT" and "set ::LLVM_CUSTOM_CMD". Notice that the "set ::LLVM_CUSTOM_CMD" can be further modified from what is shown here to modify the options passed to TAFFO in order to more precisely configure the autotuning process. In a system-level perspective, this script positions TAFFO in the "User input LLVM IR" step of the HLS pipeline. After creating the script, it needs to be run from the Vivado Tcl console.

```
open_project -reset 'proj_name'
open_solution -reset 'solution1'
add_files 'your_design.c'
add_files -tb 'your_tb.c'
set_top 'your_function'
set_part 'your_device'
create_clock -period 'your_clock'
set ::LLVM_CUSTOM_OPT taffo
set ::LLVM_CUSTOM_CMD {$LLVM_CUSTOM_OPT -vitis-hls -emit-bc}
csynth_design
```

Listing 5.1 Template Tcl script for TAFFO and Vitis HLS integration

Running the script shown above will allow Vivado to automatically tune the application with TAFFO and then synthesize it. A separate script is needed for co-simulation, as shown in Listing 5.2. This other script is used just like the first one.

```
open_project -reset 'proj_name'
open_solution -reset 'solution1'
```

```
add_files 'your_design.c'
add_files -tb 'your_tb.c'
set_top 'your_function'
set_part 'your_device'
create_clock -period 'your_clock'
set ::LLVM_CUSTOM_OPT taffo
set ::LLVM_CUSTOM_CMD {$LLVM_CUSTOM_OPT -vitis-hls-cosim -emit-bc}
cosim_design
```

Listing 5.2 Template Tcl script for co-simulation with TAFFO and HLS Vitis

5.5 Modifying an application

Once the development environment has been set up, the C/C++ code of the HLS application needs to be modified to add TAFFO annotations appropriately on variable declarations. This is done by employing the TAFFO annotation language. The TAFFO Annotation Language is used to specify various properties on the variables of the program, in order to allow TAFFO to properly infer the ranges and the errors of all the other values involved in a computation. The overall grammar of the language is shown in Figure 5.3.

```
⟨c_ann⟩    → __attribute((annotate(" ⟨top⟩ ")))
⟨top⟩      → ⟨target⟩ ⟨back⟩ ⟨datat⟩
⟨target⟩   → target(' ⟨id⟩ ') | ε
⟨back⟩     → backtracking | ε
⟨datat⟩    → ⟨scalar⟩ | ⟨struct⟩
⟨struct⟩   → struct( ⟨structl⟩ )
⟨structl⟩  → ⟨structv⟩ , ⟨structl⟩ | ⟨structv⟩
⟨structv⟩  → ⟨datav⟩ | void
⟨scalar⟩   → scalar( ⟨datav⟩ )
⟨datav⟩    → ⟨range⟩ ⟨error⟩ ⟨final⟩
⟨range⟩    → range( ⟨num⟩ , ⟨num⟩ ) | ε
⟨error⟩    → error( ⟨num⟩ ) | ε
⟨final⟩    → final | ε
```

Figure 5.3 TAFFO annotation grammar

The main components of the language are attributes and data type patterns. Attributes are used to state a property of the variable being annotated. Data type patterns are used to bind attributes to a specific part of the variable; thus, most attributes are contained inside a data type pattern. Note that, except when explicitly specified, the syntax of the TAFFO annotation language is always case-sensitive.

The TAFFO Annotation Language supports string and numeric literals. String literals are represented by a sequence of characters enclosed by single quotes ('). Any quoted character except for @ and ' is interpreted literally. The special sequence @' allows to include the single quote character inside the string, and the special sequence @@ allows to include the at-sign character. Integer-number literals are a sequence of one or more consecutive numeric characters. Such literals follow the same syntax as integer literals in the C programming language. Thus, the 0 prefix indicates that the following characters are a octal-base literal, and the 0x prefix indicates that the following characters are a hexadecimal-base literal. Boolean literals are words that specify a binary truth value. The words true and yes are used to specify a true value. The words false and no are used to specify a false value. It is not possible to implicitly cast an integer literal to a boolean literal. Real-number literals are similar to integer-number literals but also have a decimal part. The decimal separator is the dot (.). Again, the syntax is the same as the one used by the C programming language.

At the topmost level, an annotation string is composed by zero or more top-level attributes, and exactly one data type pattern except for void. The order of these elements is irrelevant, and they are optionally separated by any amount of whitespace. Each top-level attribute can appear only once, and each of them enable specific features of TAFFO which do not depend on the data type of the variable. Top level attributes can be "errtarget", "target" and "backtracking".

Data type patterns contain the attributes that relate to the actual data stored in the variable. We refer to such attributes as data attributes. For uniform data types, i.e. arrays and variables of primitive types, the "scalar" pattern is used. Instead, for arrays and variables of composed types, such as structs, the "struct" pattern needs to be introduced. The "scalar" pattern includes only one Data Attribute, while the "struct" pattern includes an attribute for each element of the struct. Notice that since TAFFO works at LLVM-IR level it cannot take into account the fact that certain programming languages introduce additional hidden members in struct types, and/or internally cast struct data to/from large integers to support specific Application Binary Interface (ABI) details. Therefore, the "struct" data type pattern needs to be used with extreme caution. TAFFO however can detect any discrepancy between the expected amount and type of struct members and the information specified in the annotations. When the discrepancies found do not allow the compilation process to continue, a detailed diagnostic is output to inform the user.

Finally, data attributes are used to specify properties of the data stored in the variable. The range attribute specifies the range of the values that can be stored in the annotated variable. The first literal is the minimum value, the second literal is the maximum value. The disabled keyword specifies that this data item should not be modified by Conversion even though it is annotated. The final keyword prevents the value range analysis stage from widening the initial range supplied for this variable. Note that this can be done as far as TAFFO is able to keep track of the annotated variable in LLVM IR, so this works best for arrays and structs.

5.6 References

[1] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, D. Grossman, Enerj: Approximate data types for safe and general low-power computation, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, p. 164–174.

[2] S. Mittal, A survey of techniques for approximate computing, ACM Computing Surveys 48 (4) (2016) 62:1–62:33.

[3] S. Cherubin, G. Agosta, Tools for reduced precision computation: a survey, ACM Computing Surveys 53 (2) (Apr 2020).

[4] D. Cattaneo, M. Chiari, N. Fossati, S. Cherubin, G. Agosta, Architecture-aware precision tuning with multiple number representation systems, in: 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 673–678.

[5] D. Cattaneo, M. Chiari, G. Magnani, N. Fossati, S. Cherubin, G. Agosta, FixM: Code generation of fixed point mathematical functions, Sustainable Computing: Informatics and Systems 29 (March 2021).

[6] R. E. Moore, et al., Introduction to interval analysis, Siam, 2009.

[7] L. H. de Figueiredo, J. Stolfi, Affine arithmetic: Concepts and applications, Numerical Algorithms 37 (1) (2004) 147–158.

[8] E. Darulova, V. Kuncak, Towards a compiler for reals, ACM Trans. Program. Lang. Syst. 39 (2) (2017) 8:1–8:28.

6 Efficient use of the HLS flow in the streaming model

6.1 Fine-grain parallelism in the streaming model

In this section, let's analyze some techniques that could be used to efficiently implement algorithms through the HLS flow in the streaming model. A streaming kernel is a device that receives a continuous sequence of input data and produces a continuous sequence of output data. Let's indicate with the ordered set

$$IS = \{di_i \mid di_i \text{ is an input data, } i=1, 2, \ldots\}$$

the input sequence and, similarly, with

$$OS = \{do_i \mid do_i \text{ is an output data, } i=1, 2, \ldots\}$$

the output sequence, which can have a number of data different from the input sequence. The kernel K implements the transformation from IS to OS. In a streaming model, indices 'i' are temporally related as data flow through input/output channels characterized by a certain finite bandwidth (BW). These data cannot be presented to the kernel simultaneously. Instead, they can be fed to 'K' in groups, with the group size GS depending on the BW of the input channel, which, in turn, is determined by the width W of the channel,

$$GS = W / \text{sizeof}(di)$$

GS is the parallelism on input data available, i.e. we can read contemporaneously GS input element di.

Let’s distinguish two different cases:

- the output depends only on input data and
- the output depends on input data and previous output data

6.1.1 Output depends only on input data

Let’s consider WS_i, a subset of IS, defined as follows:

$$WS_i=\{di_j \mid di_j \text{ is used by K to produce } do_i\}$$

WS_i contains the input data needed by K to produce do_i. If the algorithm implemented by K is regular (i.e. the distance dependence is constant), we can define the buffering space to compute do_i (BSi) as the set of all the input data belonging to IS and going from i-d- to $i+d^+$, where d- (and d+) represents the maximal distance separating d_i from all previous (subsequent) element in WS_i

$$BS_i=\{di_j \mid di_j \in IS,\ j=i-d^-,\ldots,\ i+d^+\}$$

Due to the regular nature of the algorithm implemented by K, the size of the buffer set is constant: every time new data enters BS, the oldest element exits from BS. The number of input elements necessary to pass from BS_i to BS_{i+1} represents the input that must be read to produce the new output do_{i+1}. If the number of elements that can be read at each time step is greater than the number required to compute the new output, the computation can generate at least one new output value at each time step If 'GS' is sufficiently

large to allow more output values to be computed, the kernel functionality can be replicated multiple times to generate the number of new values permitted by the 'GS' size. In a more formal context, we can express this as follows: if

$$GS \geq |BS_{i+1} \setminus BS_i|$$

then, new outputs can be computed at each time step. Specifically, 'p' new elements can be computed at each time step by replicating the kernel 'p' times, where 'p' is calculated as

$$p = Ceil\left(\frac{GS}{|BS_{i+1} \setminus BS_i|}\right)$$

The variable 'p' serves as a measure of the data parallelism that can be harnessed in the implementation of kernel 'K,' given the constraint associated with the width of the input channel. This is based on the assumption that there are sufficient resources to implement 'p' replicas of the functionality of kernel 'K.'

In the preceding discussion, our primary focus was on the input channel, assuming that the output channel had the capacity to support the required throughput. However, it's crucial to acknowledge that if the output channel does not possess the necessary capacity, a similar examination should be conducted on the output channel's capabilities. In this scenario, the most stringent condition, whether it pertains to the input or output channel, should be considered to determine the degree of parallelism 'p.' The degree of parallelism 'p' should be determined by the component (input or output) that imposes the stricter limitation, ensuring that the system operates efficiently and avoids bottlenecks in either data input or output.

In the context of the computation Kf:IS->OS, where do_i = Kf(WS_i) is determined based on specific input values, it's important to note that the computations of two distinct output values, do_i and do_j, are independent of each other and can be parallelized. As previously mentioned, it's possible to compute up to 'p' output values in parallel once the sets WS_i, …, WS_{i+p-1} have been read from the input channels.

If the computation of the kernel functionality Kf requires more than one time step, due to the independence of these different Kf computations, we can leverage temporal parallelism. This can be achieved by implementing a pipeline structure for Kf, allowing different phases of Kf for different Kf computations to overlap in time during execution.

We call fine-grained the previous two types of parallelism (data and temporal parallelism).

6.1.2 Output depends on previous outputs data

Let's focus on cases where the output depends on the current input data and the previous output. In such scenarios, the computation is typically described by the classical recurrence equation:

$$do_i = do_{i-1} \lozenge f(di_i)$$

Here, ◊ represents an associative and commutative binary operator that combines the previous output with some function of the current input(s). A classic example of this is computing the scalar product between two vectors[1]:

[1] For a detailed description of the scalar product implemented as sketched in this paragraph, refer to Marongiu A., Palazzari P.: "Using High-Level Synthesis to Implement the Matrix-Vector Multiplication on FPGA", Proc. Of the ISC 2020. June 22-25, 2020

$$s = \sum_{i=0}^{n} a_i b_i$$

This computation can be expressed through the recurrence relation:

$$s_i = s_{i-1} + a_i b_i, \quad i = 1,2,\dots,n;\ s_0 = 0;\ s = s_n$$

In the above expression, we can identify the '+' operator as the associative and commutative operator and the product between two numbers as the function f().

To leverage fine-grain data and pipeline parallelism, we can partition the input data into GS subsets, for example, using a cyclic ordering scheme like the following:

$V_j = \{v_{i \bmod GS = j}, i=1, 2, \dots,n\}$ j =0, 1, …, GS-1

Within each subset, Vj, the partial scalar product can be computed, generating GS partial results s_j that can subsequently be merged to produce the desired result, as shown below:

$$s = \sum_{j=0}^{GS-1} s_j$$

Should the ◊ operator (in the example, the '+' operator) not be computable in a combinatorial way but should involve L cycles of latency to produce its output, we should introduce L auxiliary partial results for each of the GS already defined partial results; at each clock cycle a new partial result computation begins, while the other already started partial computations proceed. The introduction of this other dimension of the partial results, sized L, implies partitioning each of the GS partial vectors V_i into L sub-vectors $V_{i,j}$. After L cycles, do_{i-L} exits from the pipeline and can be accumulated with the input to compute

$do_i = do_{i-L} \lozenge f(di_i)$

Summarizing, input data (vectors **a** and **b** in the example considered) should be partitioned in GS×L subsets to compute GS×L partial results:

$$s_{i,j,k} = s_{i-1,j,k} + a_{i,j,k} b_{i,j,k}, \quad s_{0,j,k} = 0; \quad i = 1,2,\dots,n/L;\ j = 1,2,\dots,L;\ k = 0,1,\dots,GS-1$$

The partial results must be merged through the associative and commutative operator, i.e.

$$s = \sum_{j=0}^{GS-1} \sum_{k=1}^{L} s_{j,k}$$

Following the previous scheme, at each time instant GSxL operations are being executed: each of the GS parallel functions has L different instances of the ◊ operator at different stages of execution.

6.2 Vitis implementation of the fine-grain parallelism

6.2.1 Output depending only on input data

To implement fine-grain parallelism in Vitis HLS, let's consider as explanatory example the following simple code for transforming an RGB image into the corresponding Y image.

```
#define W 256   //width in bit of the I/O streams
#define S 8     // number of bits to encode one image component (R,G,B,Y)
typedef ap_uint<W> T;
static void RGB2Y(  hls::stream<T>& sR,
                    hls::stream<T>& sG,
                    hls::stream<T>& sB,
                    hls::stream<T>& outStreamY,
                    unsigned int ImgSize,
                    unsigned int NbImages)
{
       T tmpR, tmpG, tmpB, resY;
       unsigned char r, g, b, Y;
       unsigned char cY;
       for (unsigned int k=0; k<NbImages; k++) {
       for (int i = 0; i < (ImgSize*S/W); i++) {
       #pragma HLS pipeline
              tmpR = sR.read();
              tmpG = sG.read();
              tmpB = sB.read();

              for (int j=0; j<(W/S); j++) {
              #pragma HLS unroll

                     r = (tmpR.range(S * (j + 1) - 1, S * j).to_int());
                     g = (tmpG.range(S * (j + 1) - 1, S * j).to_int());
                     b = (tmpB.range(S * (j + 1) - 1, S * j).to_int());
                     cY = (871*r+2929*g+296*b)>>12;
                     resY.range(S*(j+1)-1,S*j) = cY;
              }
              outStreamY.write(resY);
       }
       }
}
```

The T datatype represents a fixed-point integer with W=256 bits. The width of each stream is 256 bits, so it transports 32 image components. Let's combine the sR, sG and sB streams into one (logical) input stream IStream having width W = 768 bits. WS_i for the RGB2Y kernel is composed of the R, G, and B components of the i^{th} pixel and the buffering space is $BS_i=WS_i=\{R_i, B_i, G_i\}$ in this case,

$$BS_{i+1} \setminus BS_i = \{R_{i+1}, G_{i+1}, B_{i+1}\}$$

so

$$GS = W/\ sizeof(di) = 768/(8) = 96$$

and

$$p = Ceil\left(\frac{GS}{|BS_{i+1} \setminus BS_i|}\right) = \frac{96}{3} = 32$$

This means that the available data parallelism is 32, which allows us to read 32 sets of 3 values <R, G, B> at each time step. You achieve this through the following instructions, which are scheduled in parallel because they reference different streams:

```
tmpR = sR.read();
tmpG = sG.read();
tmpB = sB.read();
```

These instructions effectively read the 32 components for R (R0-R31), G (G0-G31), and B (B0-B31) in parallel, taking full advantage of the data parallelism available in the design.

The following loop

```
for (int j=0; j<(W/S); j++)
```

is fully unrolled, which means it effectively breaks down into individual operations for each value of 'j' without any loop control. This unrolling allows for the parallel extraction of all the 32 'r,' 'g,' and 'b' components from the 'tmpR,' 'tmpG,' and 'tmpB' variables. To access the 'r,' 'g,' and 'b' components, the .range(S * (j + 1) - 1, S * j) method is used, which retrieves the S=8 bits ranging from 'S * j' up to 'S * (j + 1) - 1'.

Thanks to the complete unrolling of the loop, the instruction

```
cY = (871*r+2929*g+296*b)>>12;
```

instantiates 32 independent operations to compute the 32 cY components starting from the 32 just extracted r, g, and b components.

The last instruction of the unrolled loop is

```
resY.range(S*(j+1)-1,S*j) = cY;
```

that inserts the 32 just computed cY components into the proper position of the resY variable. This is done for all 32 components in parallel, ensuring efficient processing and taking full advantage of fine-grain parallelism.

Thanks to the **`#pragma HLS pipeline`**, the **`for`** `(int i = 0; i < (ImgSize*S/W); i++)` is pipelined, i.e. at each time step a new instance of its body is scheduled to be started, while the other previous instances are still under processing. In this case, the stages of the pipeline are

1. Read input data from the sR, sG and sB streams
2. Compute in parallel the 32 cy components and assign them to the resY output variable
3. Write resY into the outStreamY stream

The previous structure can be used as a template for many streaming regular problems. Let's give a reference template structure, to be adapted to the specific function to be implemented.

```
typedef ap_uint<W> T;
static void RGB2Y(  hls::stream<T>& sI1,
                    …,
                    hls::stream<T>& sIN,
                    hls::stream<T>& outS1,
                    …
                    hls::stream<T>& outSM,
                    … // other parameters specific to the kernel to be implemented)
{
       T tmp1, …, tmpN, outV1, …, outVM;
       scalarType v1, v2, …;
       scalarType res1, …, resM;
       for (int i = 0; i < NumberOfInputsToBeRead; i++) {
       #pragma HLS pipeline
       //read all input data required to produce the desired output
              tmp1 = sI1.read();
              …
              tmpN = sIN.read();

              for (int j=0; j<p; j++) {
              #pragma HLS unroll
```

```
                // process in parallel all the p instances of the just-read inputs
                        v1 = (tmpI1.range(S * (j + 1) - 1, S * j).to_int());
                        …
                        vN = (tmpIN.range(S * (j + 1) - 1, S * j).to_int());
                        outV1 = f1(v1, …, vN);
                        …
                        outVM = fM(v1, …, vN);

                        res1.range(S*(j+1)-1,S*j) = outV1;
                        …
                        resM.range(S*(j+1)-1,S*j) = outVM;
                 }
                outS1.write(res1);
                …
                outSM.write(resM);
        }
}
```

6.2.2 Output depending on previous output data

To illustrate the implementation of fine-grain data and pipeline parallelism in the presence of computations involving, other than input data, previously computed outputs, we refer to the scalar product between two vectors.

The function takes input data from the **upStream** and **leftStream** streams, each transporting 8 floating point data (thus GS = 8). The length of the vector to be multiplied is N, a function parameter.

To accumulate the GSxL partial results the vector "**float r[SUM_LATENCY][GS]**" is used; this vector is completely unrolled in its 2nd dimensions, i.e. there are GS independent vectors, each sized SUM_LATENCY, used to store the partial results.

SUM_LATENCY is the latency of the '+' floating point operator (in our test SUM_LATENCY = 4).

After the starting loop which initializes vector **r[][]**, there is a pipelined loop that computes the GSxL partial results. Thanks to its structure, HLS compiler is able to instantiate 8 floating point adders and 8 floating point multipliers, feeding into the adders, at each clock cycle, 8 different floating point values.

After the computation loop there is the final accumulation step, which sums all the partial results to obtain the final result.

```
void scalarProduct( hls::stream<ap_int<256>>& upStream,
                    hls::stream<ap_int<256>>& leftStream,
                    unsigned int N)
{
    float r[SUM_LATENCY][GS];
    float tmp[SUM_LATENCY][GS];
    float scalarResult;
#pragma HLS ARRAY_PARTITION variable=r complete dim=2
#pragma HLS ARRAY_PARTITION variable=tmp complete dim=2

       // Init the accumulator r
       for (unsigned int j = 0; j < SUM_LATENCY; j++) {
       #pragma HLS unroll
              for (unsigned int k = 0; k < GS; k++) {
              #pragma HLS unroll
                     r[j][k] = 0.0;
              }
       }
       //Compute the partial scalar products stored in the r vector
       ap_int<256> val1,val2;
```

```
        unsigned int a1, a2;
        float tmp_add;
        for (unsigned int i = 0; i <(int)N/GS-(SUM_LATENCY-1); i += SUM_LATENCY) {
        #pragma HLS pipeline
                for (unsigned int j = 0; j < SUM_LATENCY; j++) {
                #pragma HLS unroll
                        val1 = upStream.read();
                        val2 = leftStream.read();
                        for (int k = 0; k < GS; k++) {
                        #pragma HLS unroll
                                a1=ap_int<32>(val1.range(32*(k+1)-1,32*k)).to_int();
                                a2=ap_int<32>(val2.range(32*(k+1)-1,32*k)).to_int();
                                tmp[j][k] = *(float*) ((&a1)) * *(float*) ((&a2));
                                r[j][k] = tmp[j][k] + r[j][k];
                        }
                }
        }
// compute the scalar product through the sum of the partial scalar products
        scalarResult = ((((r[0][0] + r[0][1]) + (r[0][2] + r[0][3])))
                        + (((r[0][4] + r[0][5]) + (r[0][6] + r[0][7])))
                        + (((r[1][0] + r[1][1]) + (r[1][2] + r[1][3])))
                        + (((r[1][4] + r[1][5]) + (r[1][6] + r[1][7]))))
                        + ((((r[2][0] + r[2][1]) + (r[2][2] + r[2][3])))
                        + (((r[2][4] + r[2][5]) + (r[2][6] + r[2][7])))
                        + (((r[3][0] + r[3][1]) + (r[3][2] + r[3][3])))
                        + (((r[3][4] + r[3][5]) + (r[3][6] + r[3][7]))))
                        + (((r[4][0] + r[4][1]) + (r[4][2] + r[4][3])))
                        + (((r[4][4] + r[4][5]) + (r[4][6] + r[4][7])));
}
```

6.3 The medium-grain parallelism

In several cases, there is a natural increase in data granularity. Consider matrix computations, where individual rows (or columns) of the matrix can serve as atomic elements for expressing matrix operations. For instance, new matrix elements can be thought of as scalar products between rows and columns. Similarly, in image processing, certain computations can be viewed as operations between image lines. When applying a convolution filter with a radius 'r,' the new line of an image can be seen as the result of processing 2r+1 input lines.

Let's formalize this concept.

The kernel computation vKf: IS->OS operates on medium-grain I/O data, such as vectors and image lines. We can denote the input vector sequence as:

$$IVS = \{vi_i \mid vi_i \text{ is an input vector, } i=1, 2, \ldots\}$$

Similarly, the output vector sequence can be represented as:

$$OVS = \{vo_i \mid vo_i \text{ is an output vector, } i=1, 2, \ldots\}$$

The input and output vectors are ordered sets of input and output data (di and do)

The vector kernel function generates one new output vector from the set VWS_i which contains the input vectors needed,

$$VWS_i=\{vi_j \mid vi_j \text{ is used by vkf to produce } vo_i\}$$

The implementation of the vector kernel function can still leverage the fine-grain parallelism described in the previous paragraph.

However, due to the large size of VWS_i, which has constant size for uniform problems, when data are read from the input stream, they are not fed directly to the kernel function. Instead, they are stored in buffers allocated in the local memory, characterized by fast access time, typically with a data latency of 1 clock cycle.

To define the structure of the pipelined medium-grain implementation, we introduce some auxiliary functions:

- **`readVector(instream, v, vSize)`**: This function reads **`vSize`** bytes from instream, using data and temporal parallelism, to read **`W`** bits per cycle from the input stream. It stores them in **`v`** in a pipelined manner.
- **`push(v)`**: This function inserts **`v`** into **`VWS`**, while removing the oldest vector previously stored in it.
- **`writeVector(outstream, v, vSize)`**: This function reads **`v`** from the local memory and writes it to **`outstream`**. Again, it employs data and temporal parallelism, writing **`W`** bits per cycle into the output stream.

The vector kernel function, defined as **`vkf(VWS, v)`**, reads data from the **`VWS`** to generate the output vector **`v`**. It follows the structure outlined in the previous paragraph to leverage data and temporal parallelism. The key distinction is that input data is not read or written from/to streams but instead from memory modules, typically the SRAM modules within FPGAs.

The kernel algorithm is implemented in a pipelined way and is organized as a sequence of macro-steps. In the steady state, during each macro-step, each function (readVector(), compute vkf(), writeVector()) is active and processes a different input VWS.

Let's determine the number of vectors needed to implement the computation:

- 1 vector is necessary to read, during each macro-step, the input vector that will be used in the next macro-step to compute the output vector.
- N vectors are needed to store the **`VWS`**, with N representing the number of input vectors required to compute an output vector.
- 2 vectors are essential to store the output vectors, following a double buffering scheme. One vector is used to store the results computed by the vector kernel function, while the other is used by the **`writeVector(...)`** function to send the result vector computed in the previous macro-step to the output streams. The roles of these vectors are exchanged during each macro-step.

Let's give the general structure of the computation that implements the kernel which leverages the medium-grain pipelined parallelism:

```
/*
pipeline preamble to reach the steady state. In this preamble it is managed the
eventual production of output vectors when input vectors are not enough to produce
an output
*/
…
while (not all input data have been consumed) {
readVector(instream, v1, vSize);
       if (doubleBufferingPhase == 'A') {
              vkf(VWS,vOutA);
              writeVector(outstream, vOutB);
       }
       else {
```

```
                vkf(VWS,vOutB);
                writeVector(outstream, vOutA);
        }
        flip(doubleBufferingPhase);
        push(v1);
}
/*
Postamble to manage the tail of the pipeline. No more data are read from the input.
*/
…
```

6.4 Vitis HLS implementation of the medium-grain pipeline

In this paragraph, we give an example to show how the previous scheme could be implemented in the Vitis HLS flow.

As an illustrative example, we will use the implementation of a 3x3 filter to process images received through the input stream. Below, you'll find the code to implement the kernel function. Within this code, the function

- **`readline<T>(inStream, line, NbParallelInputWords)`** reads the image line **`'line'`** from **`inStream`**. Both **`inStream`** and the vector **`line`** handle data with a datatype of **`T`**. In our example, T is an **`ap_uint<256>`**, meaning it is a 32-byte wide data. This function corresponds to what was previously referred to as **`readVector()`** in the previous paragraph.
- **`filter_one_line_3x3<T,S,W>(l1, l2, l3, lout1, ,…)`** reads line **`l1, l2, l3`** to produce the output line **`lout1`**; this function corresponds to the **`vkf()`** in the previous paragraph; in this case the input working set for the function is **`VWS={l1, l2, l3}`**
- **`writeline<T>(outStream, line, NbParallelInputWords)`** writes the image line **`'line'`** into **`outStream`**. Both **`outStream`** and the vector **`line`** handle data with a datatype of **`T`**. This function corresponds to what was previously referred to as **`writeVector()`** in the previous paragraph.

The **`push(v)`** function has not been implemented, as the VWS has been explicitly managed, taking into account all four possible cases for the rotation of the 4 input lines. In other words, VWS is structured as follows: **`VWS = {l1, l2, l3} | {l2, l3, l4} | {l3, l4, l1} | {l4, l1, l2}`**. The **`filter_one_line_3x3()`** function is then called with the appropriate VWS configuration.

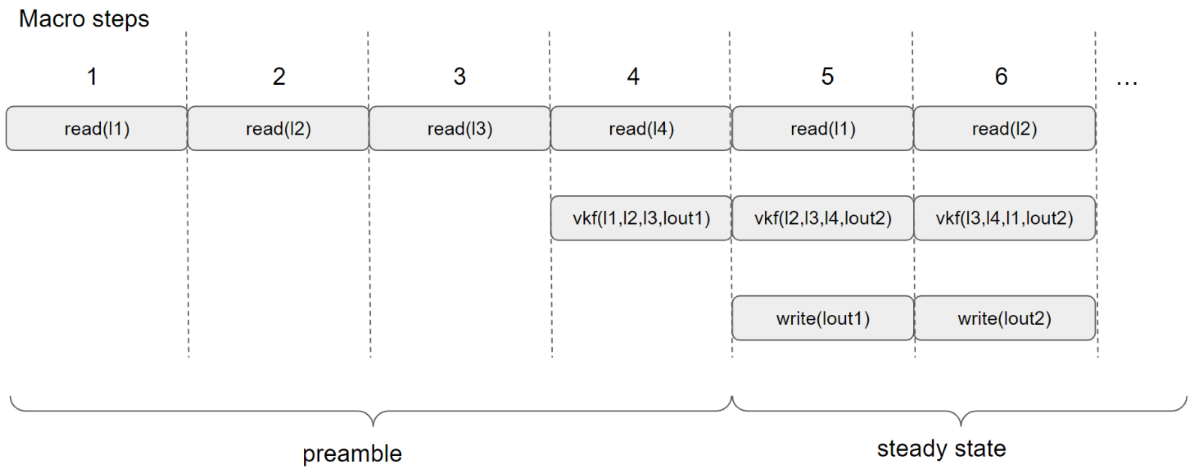
The double buffering is achieved using the variable **`output_line`** which determines the line (lout1|lout2) to be passed to the **`writeline()`** function. Conversely, the complementary line is passed to the **`filter_one_line_3x3() function.`**

```
template <typename T, int S, int W>
// S is the size, in bits, of the pixel component.
// W is the width, in bits, of data type T
void do_3x3_filtering(hls::stream<T>& inStream,hls::stream<T>& out_stream,
  unsigned short int ImgRows, unsigned short int ImgCols,
  … // other parameters)
{
        int i;
        T l1[MAX_COMPONENT_LINE_SIZE];
        T l2[MAX_COMPONENT_LINE_SIZE];
        T l3[MAX_COMPONENT_LINE_SIZE];
        T l4[MAX_COMPONENT_LINE_SIZE];
        T lout1[MAX_COMPONENT_LINE_SIZE];
        T lout2[MAX_COMPONENT_LINE_SIZE];
```

```
short int new_read_line = 1;
short int output_line = 1;

// pipeline preamble
readline<T>(inStream,line1,NbParallelInputWords);
readline<T>(inStream,line2,NbParallelInputWords);
readline<T>(inStream,l3,NbParallelInputWords);
readline<T>(inStream,l4,NbParallelInputWords);
filter_one_line_3x3<T,S,W>(l1, l2, l3, lout1, ,…);

// pipeline steady state
for (i=4; i<ImgRows; i++)  // Steady state code
{
       if ((new_read_line == 1) && (output_line == 1))
       {
              readline<T>(inStream,l1,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l2, l3, l4, lout2 , ...);
              writeline<T>(out_stream, lout1, NbParallelInputWords);
              new_read_line = 2; output_line = 2;
       }
       else if ((new_read_line == 1) && (output_line == 2))
       {
              readline<T>(inStream,l1,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l2, l3, l4, lout1 , ...);
              writeline<T>(out_stream, lout2, NbParallelInputWords);
              new_read_line = 2; output_line = 1;
       }
       else if ((new_read_line == 2) && (output_line == 1))
       {
              readline<T>(inStream,l2,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l3, l4, l1, lout2 , ...);
              writeline<T>(out_stream, lout1, NbParallelInputWords);
              new_read_line = 3; output_line = 2;
       }
       else if ((new_read_line == 2) && (output_line == 2))
       {
              readline<T>(inStream,l2,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l3, l4, l1, lout1 , ...);
              writeline<T>(out_stream, lout2, NbParallelInputWords);
              new_read_line = 3; output_line = 1;
       }
       else if ((new_read_line == 3) && (output_line == 1))
       {
              readline<T>(inStream,l3,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l4, l1, l2, lout2 , ...);
              writeline<T>(out_stream, lout1, NbParallelInputWords);
              new_read_line = 4; output_line = 2;
       }
       else if ((new_read_line == 3) && (output_line == 2))
       {
              readline<T>(inStream,l3,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l4, l1, l2, lout1 , ...);
              writeline<T>(out_stream, lout2, NbParallelInputWords);
              new_read_line = 4; output_line = 1;
       }
       else if ((new_read_line == 4) && (output_line == 1))
       {
              readline<T>(inStream,l4,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l1, l2, l3, lout2 , ...);
              writeline<T>(out_stream, lout1, NbParallelInputWords);
              new_read_line = 1; output_line = 2;
       }
       else if ((new_read_line == 4) && (output_line == 2))
       {
              readline<T>(inStream,l4,NbParallelInputWords);
              filter_one_line_3x3<T,S,W>(l1, l2, l3, lout1 , ...);
              writeline<T>(out_stream, lout2, NbParallelInputWords);
              new_read_line = 1; output_line = 1;
```

```
            }
      }
      // end of the pipeline steady state

      // first stage of the postamble - We do not read anymore new lines
      if ((new_read_line == 1) && (output_line == 1))
      {
            filter_one_line_3x3<T,S,W>(l2, l3, l4, lout2 , ...);
            writeline<T>(out_stream, lout1, NbParallelInputWords);
            output_line = 2;
      }
      else if ((new_read_line == 1) && (output_line == 2))
      …
      else if ((new_read_line == 4) && (output_line == 2))
      {
            filter_one_line_3x3<T,S,W>(l1, l2, l3, lout1 , ...);
            writeline<T>(out_stream, lout2, NbParallelInputWords);
            output_line = 1;
      }

      // second stage of the postamble - We do not do more processing
      if (output_line == 1)
            writeline<T>(out_stream, lout1, NbParallelInputWords);
      else
            writeline<T>(out_stream, lout2, NbParallelInputWords);
}
```

The following figure displays the Gantt chart illustrating how the previous code is executed. The time steps of the pipeline are referred to as 'macro steps,' and each of them encompasses operations that require many clock cycles. In this figure, it is assumed that the time required to read one line from the input stream, write one line to the output stream, and compute the output line from the three input lines is equivalent. This assumption is reasonable since all the functions involved are pipelined and operate with the same data parallelism. If the functions had different execution times, the length of the macro step would be determined by the maximum of these execution times.



6.5 The coarse-grain pipeline

In some cases, the algorithm structure necessitates processing all input data before starting to transmit output values. This is particularly true for operations like vertical mirroring of an image, where the last line of the image must be received before initiating the output transmission. In such scenarios, we are in a

'store and forward'-like computation, where data from the input stream is first completely read (store phase), which may include some preprocessing, and then it is re-read for further processing and forwarding to the output stream (forward phase). To ensure a continuous flow of data in these situations, the computing kernel responsible for processing needs to be duplicated, and appropriate splitting and merging devices must be introduced to maintain the uninterrupted data flow towards one of the two kernels.

The input set is still composed of (large) vector data, and we process a sequence of such vectors (for instance, a sequence of images)

$$IVS = \{vi_i \mid vi_i \text{ is an input vector, } i=1, 2, \ldots\}$$

Similarly, the output vector sequence can be represented as:

$$OVS = \{vo_i \mid vo_i \text{ is an output vector, } i=1, 2, \ldots\}$$

The vector kernel function generates one new output vector from the set VWS_i which contains the input vectors needed,

$$VWS_i=\{vi_j \mid vi_j \text{ is used by vkf to produce } vo_i\}$$

The implementation of the vector kernel function can still leverage the fine-grain and medium-grain parallelism described in the previous paragraph.

In order to keep the notation simple, let's suppose that $vo_i = vkf(vi_i)$, meaning the output vector depends only on the current input vector. The more general case can be properly addressed by adding additional buffering to store all the necessary input vectors. However, given the use of large vectors, they cannot be stored in the internal SRAM of the FPGA memory but must be placed in the larger external DDR memory.

The vkf has the following structure:

```
while (there are input data to be processed)
{
        readVector(inStream,v);
        processAndWrite(v, outStream);
}
```

The **`readVector()`** function is responsible for reading the entire vector **`v`** from the **`inStream`**. Since the data processing depends on the contents of **`v`**, the **`processAndWrite()`** function is initiated only after the completion of **`readVector()`**.

The **`processAndWrite()`** function employs a combination of fine-grain and medium-grain parallelism to read and process data from the **`v`** vector. As soon as the data is processed, it is sent to the **`outstream`** in a pipelined way. When analyzing the I/O streams, the kernel's behavior can be divided into two distinct phases:

1. In the first phase, there is continuous data streaming through **`inStream`**, while no activity occurs on **`outstream`**. During this phase, the VKF kernel is executing the **`readVector()`** function.
2. In the second phase, **`inStream`** remains inactive, and there is continuous data streaming on **`outStream`**. This phase corresponds to the VKF kernel's execution of the **`processAndWrite()`** function.

As both readVector() and processAndWrite() functions are pipelined, their execution time, neglecting the initiation latency, is given by

$$T_{readVector} = \frac{|vin| \times sizeof(di)}{W_{inStream} \times f_{ck}}$$

$$T_{processAndWrite} = \frac{|vout| \times sizeof(do)}{W_{outStream} \times f_{ck}}$$

In the previous expressions,

- W indicates the bit-width of the stream,
- |v| the number of elements d contained in vector v;
- $W \times f_{ck}$ is the bandwidth available on the stream.

When dealing with such coarse-grain kernels, continuous streaming on the I/O streams can be achieved by replicating the vkf twice and using a pair of Split/Merge functions. The Split function takes a continuous sequence of input vectors from instream and sends them to two output streams, in an alternate way; the first vector is sent, in a completely pipelined way, to the first out stream, the second vector to the second out stream, the third vector is sent again on the first out stream and so on, repeating the process until all input vectors are processed. The Merge function has a dual behavior. It receives sequences of vectors alternately from its two input streams and sends them to the output stream. The following picture explicates this behavior.

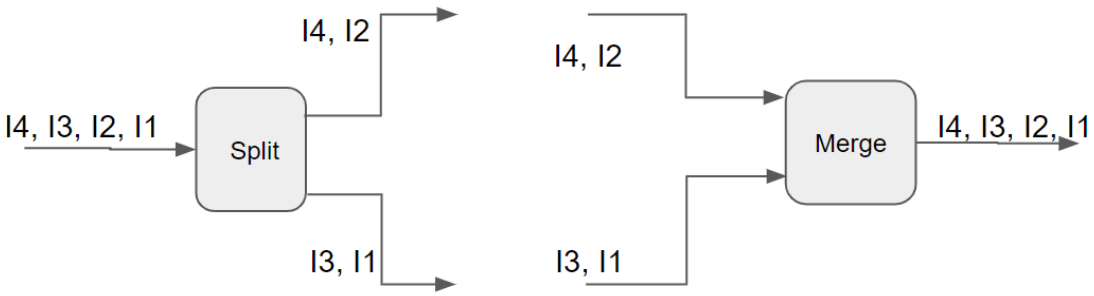


Figure: behavior of the Split and Merge functions

In order to have continuous streaming on both the input and output streams, the throughput must be balanced, i.e. $T_{readVector}=T_{processAndWrite}$ must result: this means that $W_{inStream}$ and $W_{outStream}$, the parameters controlling fine-grain data parallelism, must be properly selected, resulting

$$W_{outStream} = W_{inStream} \frac{|vout| \times sizeof(do)}{|vin| \times sizeof(di)}$$

Putting the things together, in order to have continuous streaming in the case of coarse-grain functions, the following scheme must be adopted:

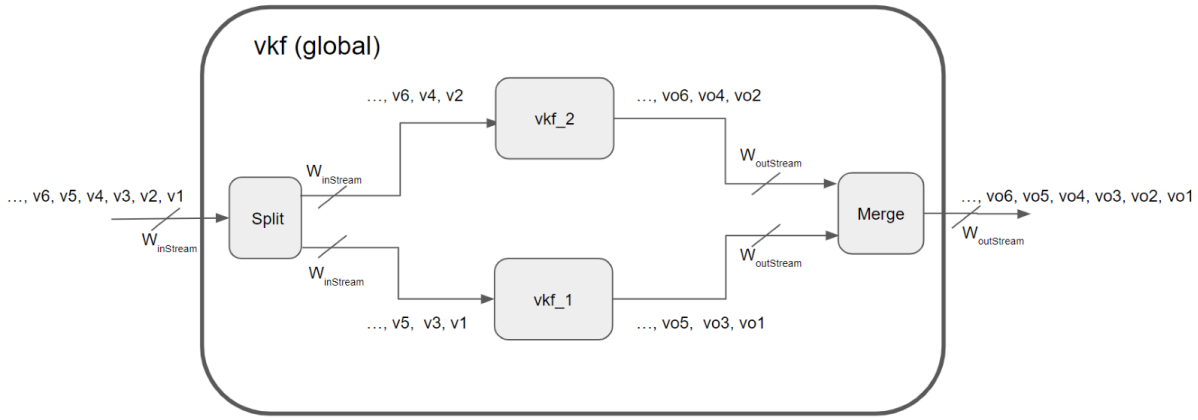


Figure: structure of the vkf global kernel, allowing continuous streaming on the I/O streams

6.6 Vitis HLS implementation of the coarse-grain pipeline

To illustrate the implementation of a coarse-grain kernel, we will refer to a kernel designed to enhance the contrast of an image received through the input stream. The process involves receiving the entire image to determine the maximum and minimum values assumed by the image pixels. While receiving the image, it is temporarily stored in a locally allocated buffer within the external memory.

Once the maximum and minimum values have been computed, the image is read again, and new pixel values are calculated using linear scaling. These processed pixel values are then transmitted in a pipelined way through the output stream.

The behavior of the inner vector kernel function, here called `'contrastEnhance'`, clearly falls in the coarse-grain model: each computation on an image is independent of other images and the entire image must be received before starting to output the new image.

The code to implement the function is the following:

```
template <typename T, int S, int W>
// T is the datatype associated to input and output streams
// S is the size, in bits, of the pixel component.
// W is the width, in bits, of data type T
void contrastEnhance(T* Mem,                    // external memory buffer
                     hls::stream<T>& inStream,
                     hls::stream<T>& outStream,
                     unsigned int ImgSize, unsigned int NbImages)
{
       T tmp, minValues, maxValues, outVal;
       ap_uint<S> minV,maxV,currV,tmpMin, tmpMax;
       unsigned short int value;
       for (unsigned int k=0; k<NbImages; k++)  {
              for (int j=0; j<(W/S); j++)       {
              #pragma HLS unroll
                     minValues.range(S*(j+1)-1,S*j) = 255;
                     maxValues.range(S*(j+1)-1,S*j) = 0;
               }
              // read the image,
              // search for the maximum and the minimum, and
              // store the image in an external memory bank
              for (int i = 0; i < (ImgSize*S/W); i++) {
```

```
                #pragma HLS pipeline
                        tmp = inStream.read();
                        Mem[i] = tmp;
                        for (int j=0; j<(W/S); j++) {
                        #pragma HLS unroll

                                currV = (tmp.range(S*(j+1)-1, S*j).to_int());
                                tmpMin = (minValues.range(S*(j+1)-1, S*j).to_int());
                                tmpMax = (maxValues.range(S*(j+1)-1, S*j).to_int());
                                if (currV < tmpMin)
                                        tmpMin = currV;
                                if (currV > tmpMax)
                                        tmpMax = currV;
                                minValues.range(S*(j+1)-1, S*j) = tmpMin;
                                maxValues.range(S*(j+1)-1, S*j) = tmpMax;
                        }
                }
                minV = 255;
                maxV = 0;
                for (int j=0; j<(W/S); j++) {
                #pragma HLS unroll
                tmpMin = minValues.range(S*(j+1)-1,S*j);
                        tmpMax = maxValues.range(S*(j+1)-1,S*j);
                        if (minV > tmpMin)
                        minV = tmpMin;
                        if (maxV < tmpMax)
                        maxV = tmpMax;
                 }  //minV and maxV are minimum and maximum values in the image;

                // re-read image from memory and
                // change the pixel values to use all the pixel dynamics
                for (int i = 0; i < (ImgSize*S/W); i++) {
                #pragma HLS pipeline
                        tmp = Mem[i];
                        for (int j=0; j<(W/S); j++) {
                        #pragma HLS unroll
                                currV = (tmp.range(S * (j + 1) - 1, S * j).to_int());
                                if (maxV != minV)
                                {
                                        unsigned short tmp1, tmp2, tmp3;
                                        tmp1 = (unsigned short)(maxV-minV);
                                        tmp2 = (unsigned short)(currV - minV);
                                        value = (((255*256)/tmp1)*tmp2)/256;
                                        if (value > 255)
                                                value = 255;
                                        }
                                        else
                                                value = currV;
                                        outVal.range(S*(j+1)-1,S*j) = value;
                                }
                                outStream.write(outVal);
                        }
                }
}
```

The kernel starts with the loop

```
for (unsigned int k=0; k<NbImages; k++)
```

This loop iterates to process multiple images, demonstrating coarse-grain pipelining as it overlaps the storage and processing phases for these images.

Following the fine-grain parallelism scheme, **`W/S`** (width divided by size) data elements are concurrently read from the stream and stored in the external memory buffer **`Mem`**. During this read and storage

operation, the algorithm concurrently searches for maximum and minimum values. This parallelism results in **W/S** searches happening simultaneously, yielding **W/S** values for both the maximum and minimum at the end of the loop.

Subsequently, another loop operates on these **W/S** maximum and minimum values to derive the maximum and minimum values for the entire image.

After previous step, the image is stored in memory and the computation of Max and Min values is complete. The store phase of the algorithm has concluded.

The forward phase is performed through the following loop

```
for (int i = 0; i < (ImgSize*S/W); i++)
```

This loop operates in a pipelined manner (**#pragma HLS pipeline**) and performs the following steps

- reads data from the external memory buffer (**Mem**)
- performs a linear scaling on the just read **W/S** pixels
- writes the **W/S** scaled pixel values into **outStream**

In the previous paragraph, we discussed the need for a continuous data stream through the I/O streams. To achieve this, the vector kernel function must be instantiated twice. While one instance is in the store phase, the other is in the forward phase, and they switch roles at the completion of each image.

To implement the continuous switching of the flow between the two instances, the split and merge functions must be implemented. They both use the fine-grain parallelism, and are implemented as follows:

```
template <typename T, int S, int W>
// T is the datatype associated to input and output streams
// S is the size, in bits, of the pixel component.
// W is the width, in bits, of data type T
void split2Channels(hls::stream<T>& sin,
                    hls::stream<T>& soutA, hls::stream<T>& soutB,
                    unsigned int ImgSize, unsigned int NbImages)
{
       T tmp;
       unsigned short phase = 1;
       for (int k=0; k<NbImages; k++)  {
              if (phase == 1) {
                     for (int i = 0; i < (ImgSize*S/W); i++) {
                     #pragma HLS pipeline
                            tmp = sin.read();
                            soutA.write(tmp);
                     }
                     phase = 2;
              }
              else   {
                     for (int i = 0; i < (ImgSize*S/W); i++) {
                     #pragma HLS pipeline
                            tmp = sin.read();
                            soutB.write(tmp);
                     }
                     phase = 1;
              }
       }
}

template <typename T, int S, int W>
void merge2Channels(hls::stream<T>& sinA, hls::stream<T>& sinB,
                    hls::stream<T>& sout,
                    unsigned int ImgSize, unsigned int NbImages)
{
```

```
        T tmp;
        unsigned short phase = 1;
        for (int k=0; k<NbImages; k++)   {
                if (phase == 1)
                {
                        for (int i = 0; i < (ImgSize*S/W); i++) {
                        #pragma HLS pipeline
                                tmp = sinA.read();
                                sout.write(tmp);
                        }
                        phase = 2;
                }
                else
                {
                        for (int i = 0; i < (ImgSize*S/W); i++) {
                        #pragma HLS pipeline
                                tmp = sinB.read();
                                sout.write(tmp);
                        }
                        phase = 1;
                }
        }
}
```

As we see, the two functions are perfectly symmetric:

- **split2Channels** iterates on **NbImages** and, each image is alternatively sent through one of the two output streams in a pipelined way, with data parallelism given by **W/S** pixels.
- **merge2Channels** iterates on **NbImages** and, each image is alternatively received through one of the two input streams in a pipelined way, with data parallelism given by **W/S** pixels.

The code snippet below outlines the implementation of the alternate behavior that allows continuous streaming:

```
#pragma HLS dataflow
   setNbImages(NbImages, NbImagesA, NbImagesB);
   split2Channels(sin, sA, sB, ImgSize, NbImages);
   contrastEnhance(BufferA, sA, soutA, ImgSize, NbImagesA);
   contrastEnhance(BufferB, sB, soutB, ImgSize, NbImagesB);
   merge2Channels(soutA, soutB, sout, ImgSize, NbImages);
```

The dataflow pragma instructs the compiler to execute all tasks in parallel, synchronizing based on data availability;

The **setNbImages**() function divides the **NbImages** variable by two, assigning half of its value to **NbImagesA** and the other half to **NbImagesB**. If **NbImages** is odd, **NbImagesA** is assigned the value of **NbImagesB + 1**.

The **split2Channels()** function reads **NbImages** from **sin** and alternately sends **NbImagesA** images to **sA** and **NbImagesB** to **sB**.

The behaviour of **merge2Channels()** is dual: it reads alternately **NbImagesA** images from **soutA** and **NbImagesB** from **soutB** and sends **NbImages** to **sout**.

The two instances of the **contrastEnhance** kernel refer to two distinct temporary buffers **BufferA** and **BufferB**, used to store and reload images during the store and forward phases.

6.7 Efficient burst access to DDR banks

As accessing external memory incurs an initial penalty cost, it's essential to optimize data access in external memory by performing burst accesses. In some cases, continuous reading or writing of data from adjacent memory locations isn't possible. For example, when data originates from a data compressing engine, it may not be available at every clock cycle because the output data is typically smaller than the input data. In such scenarios, we recommend using a small intermediate buffer allocated in the fast internal BRAM modules. This buffer allows for handling irregular accesses without incurring any startup penalties, and it can be efficiently moved to and from the external memory using burst accesses.

To enable continuous memory access, the adoption of a double buffering scheme is essential. In this scheme, while one internal buffer is being burst-accessed in the external memory (either for reading or writing), the other buffer is utilized by the kernel for data access, and vice versa. Below, we present the HLS code for implementing this double buffering scheme when writing data to external memory.

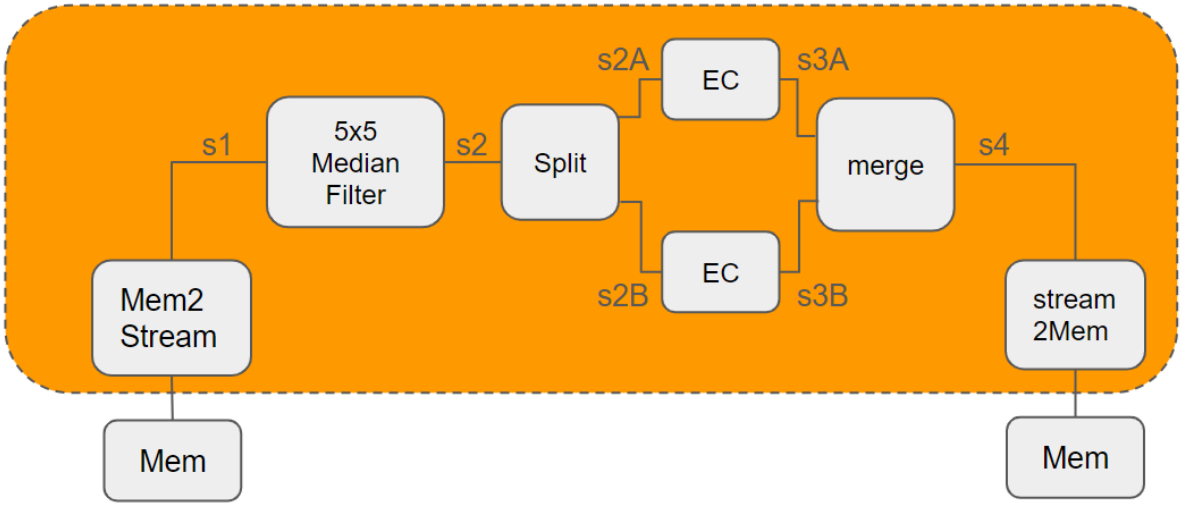
```
bool Stream2Buff(hls::stream<dt>& inStream,
                 dt* Buffer,
                 unsigned int *NbData)
{
       dt tmp;
       unsigned int i=0;
       tmp = inStream.read();
       while ((tmp.last==false)&&(i<BUF_SIZE-1)) {
              Buffer[i] = tmp.data;
              tmp = inStream.read();
              i++;
       }
       Buffer[i] = tmp.data;
       *NbData = i+1;
       return tmp.last;
}

void Buff2Mem(dt* Buffer, dt* Mem, unsigned int NbData, unsigned int offset)
{
       for (unsigned int i=0; i<NbData; i++)
              Mem[offset + i] = Buffer[i];
}

void Stream2Memory( dt* out,
                    hls::stream<dt>& inStream) {
       unsigned int offset = 0;
       unsigned int i=0;
       unsigned int NbData1, NbData2;
       bool phase = false;
       dt Buffer1[BUF_SIZE];
       dt Buffer2[BUF_SIZE];
       bool last = false;
       last = Stream2Buff(inStream, Buffer1, &NbData1);
       while (last == false)
       {
              if (phase == false)
              {
                     Buff2Mem(Buffer1, out, NbData1, offset);
                     last = Stream2Buff(inStream, Buffer2, &NbData2);
                     offset += NbData1;
              }
              else
              {
                     Buff2Mem(Buffer2, out, NbData2, offset);
```

```
                    last = Stream2Buff(inStream, Buffer1, &NbData1);
                    offset += NbData2;
                }
                phase = !phase;
        }
        if (phase == false)
                Buff2Mem(Buffer1, out, NbData1, offset);
        else
                Buff2Mem(Buffer2, out, NbData2, offset);
}
```

As we can observe, we have defined two auxiliary functions: **`Stream2Buff`**() and **`Buff2Stream`**(). These functions facilitate the movement of data between an internal buffer, allocated on BRAM modules, and the input stream. The behavior of **`StreamToBuff`**() is controlled by the **`'last'`** field of the input data. If packetized communication, controlled by the 'eop' signal, is not in use, we can rely on the message's size being transferred to determine the end-of-transmission condition. Both functions are easily pipelined by the Vitis HLS compiler, allowing each of these functions to read and write the bytes contained in the 'dt' data type (e.g., 16 or 32 bytes) at each clock cycle.
The actual interfacing between the input stream **`'inStream'`** and the external memory **`'out'`** is carried out by the **`'Stream2Memory`**()' function, which implements a double buffering scheme. In both phases of the 'ping-pong,' the HLS compiler detects that **`'Buff2Mem`**()' and **`'Stream2Buff`**()' are accessing different buffers, allowing them to be scheduled in parallel. This overlapping scheduling optimizes data movement between the stream and one internal buffer, along with the movement between the other internal buffer and the memory.

6.8 Using HDL simulation to look inside FPGA parallelism

To demonstrate the implementation of parallelism at the various levels of granularity, we've developed a simple image elaboration algorithm, as shown in the figure below. This algorithm incorporates two data mover modules (Mem2Stream and Stream2Mem), a 5x5 median filter, a split-merge pair, and two instances of the EnhanceContrast kernel.

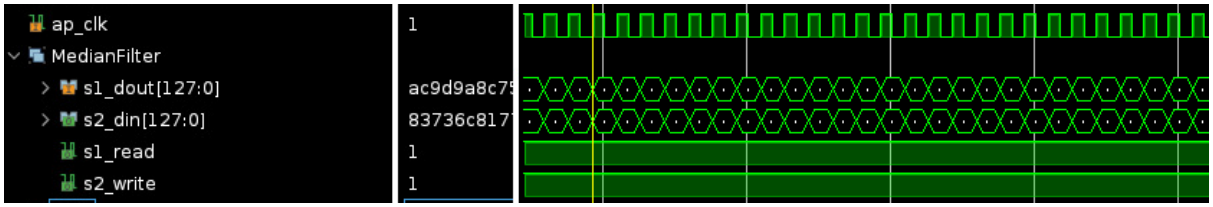


In the sections that follow, we present excerpts from the simulation waveforms to showcase the different levels of parallelism granularity.

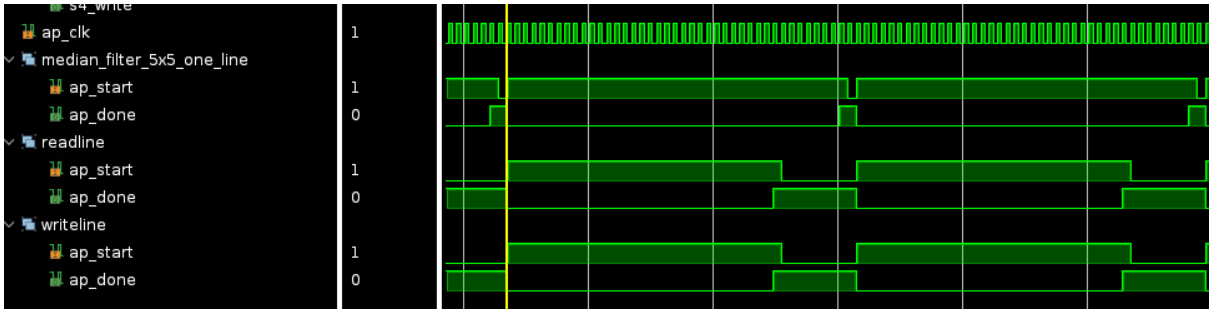
In the following figure, we present the I/O behavior of the median filter. As depicted, during each clock cycle, stream s1 reads while s2 writes a set of 16 bytes. Both signals, s1_read and s2_write, remain

continuously asserted, and the width of s1 and s2 is 128 bits. The module reads from the output side of s1 (s1_dout) and writes to the input side of s2 (s2_din).

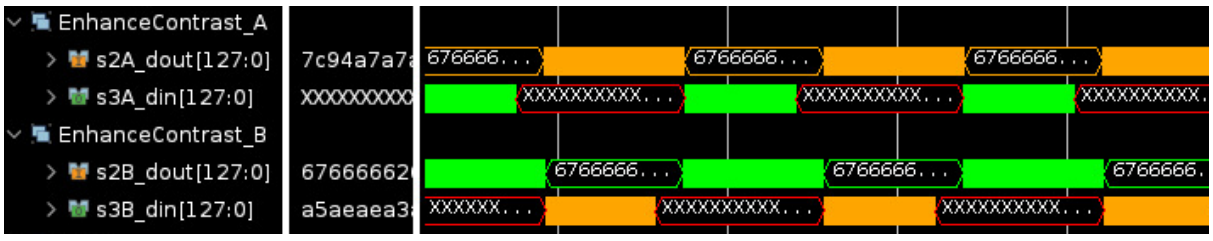
The ability to read and process 16 bytes per clock cycle has been achieved using fine-grained data and pipeline parallelism.



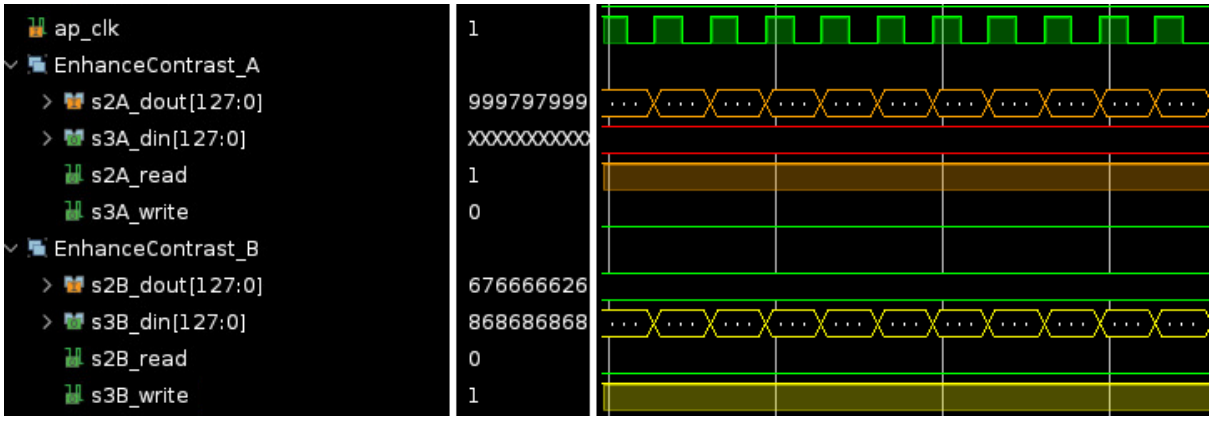
To illustrate the operation of medium-grain parallelism utilized in the median filter, we present the start/done signals of the **`readline()`**, **`writeline()`**, and **`median_filter_5x5_one_line()`** modules, which constitute the three stages of the asynchronous medium-grain pipeline. It is evident that all three modules are concurrently active, and the pipeline computation progresses at the pace dictated by the module with the longest execution time, which in this case is **`median_filter_5x5_one_line`**. Each of these modules is responsible for processing a different line: **`readline()`** reads the i^{th} line, **`median_filter_5x5_one_line()`** processes the $(i-1)^{th}$ to $(i-5)^{th}$ lines, and **`writeline()`** writes the result line computed in the previous step, processing the $(i-2)^{th}$ to $(i-6)^{th}$ lines



The image below illustrates the achievement of coarse-grained parallelism through the utilization of two identical EnhanceContrast modules in conjunction with a split/merge pair. As depicted, while EnhanceContrast_A reads an image from s2A, the EnhanceContrast_B module simultaneously writes the just-processed image to s3B (orange waveforms). Once this phase is completed, EnhanceContrast_A transitions to writing the just-processed image on s3A, while EnhanceContrast_B begins reading the new input image from s2B (green waveforms).



Thanks to the fine-grained parallelism, the reading and writing of new groups of 16 bytes are executed during each clock cycle, as evident in the image below



7 Conclusions

The significance of High-Level Synthesis (HLS) within the software framework developed in TEXTAROSSA for managing FPGA accelerators underscores the substantial efforts invested in leveraging existing HLS flows. Specifically, considerable focus has been directed towards utilizing established HLS methodologies, notably the Vitis HLS flow from AMD, and integrating them with both the hardware (HW) and software (SW) components under development within the project.

This document outlines the interactions and utilization of various tools and Intellectual Properties (IPs) developed within the TEXTAROSSA project that interface with and utilize Vitis.

For both the streaming and task-based parallel models, comprehensive discussions delve into the interfacing and utilization of the Vitis flow, accompanied by presented results highlighting the advantageous outcomes stemming from the adoption of FPGA accelerators.

The APEIRON communication IP and its associated software stack are showcased, emphasizing their interfacing with the Vitis HLS flow. Additionally, the document introduces the defined API intended for accessing the communication hardware/software layer through the standard C++ Vitis programming style.

Moreover, the document elucidates the integration of the TAFFO library into the Vitis flow. TAFFO's purpose lies in managing limited precision arithmetic, offering methods for code analysis destined for implementation on the accelerator. It also facilitates minimizing the size of variables used to maintain a specified precision level in the results.

Lastly, recognizing that programming for High-Level Synthesis entails distinct considerations and skills compared to those in standard CPU/GPU programming, a dedicated section provides guidelines for effectively utilizing the HLS flow in the streaming model.