Version	**1.1**
Date	**2021/07/31**
Author	**INAF**
Dissemination status	**PU**
Document reference	**D2.6**

D2.6: Final ported application software

Project acronym: EuroEXA
Project name: EuroEXA - Co-designed Innovation and System for Resilient Exascale Computing in Europe: From Applications to Silicon.
Call and contract: H2020-EU.1.2.2. - FET Proactive
Grant agreement No: 754337
Project duration: H2020-EU.1.2.2. - FET Proactive

Funded by the Horizon 2020 Framework Programme of the European Union

EuroEXA: Co-designed Innovation and System for Resilient Exascale Computing in Europe: From Applications to Silicon

Project No: 754337

H2020-EU.1.2.2.-FET Proactive
D2.6: Final ported application software

Due date of deliverable:	2021/07/31
Actual submission date:	2022/05/20

Executive summary

This document describes the ported software of the EuroEXA applications to the single CRDB testbed and it discusses the experiences extracted from porting and optimization activities that should be actively taken into account in future redesign and optimization. This document accompanies the ported application software, found in the EuroEXA private repository (https://github.com/euroexa). In particular, this document describes the status of the software for each of the EuroEXA applications, sketches the redesign and optimization strategy for each application, discusses issues and difficulties faced during the porting activities and the relative lesson learned. A few preliminary evaluation results have been presented, however the full evaluation will be discussed in deliverable 2.8.

List of authors

Author	Affiliation
Giuliano Taffoni	INAF (lead)
Ugo Becciani	INAF
Valentina Cesare	INAF
David Goz	INAF
Luca Tornatore	INAF
Fabio Vitello	INAF
Paul Carpenter	BSC
Omar Shaaban Ibrahim Ali	BSC
Michael Lange ECMWF	ECMWF
Olivier Marsden	ECMWF
Balthasar Reuter	ECMWF
Tampouratzis Nikolaos	EXA
Yannis Papaefstathiou	EXA
Daniel Grűnewald	FRAUN
Matthias Balzer	FRAUN
Christos Kozanitis	FORTH
Apostolos Dollas	FORTH
Ioannis Kalomoiris	FORTH
Antonios-Georgios Pitsis	FORTH
Aggelos Ioannou	FORTH
Georgious Goumas	ICCS
Nikela Papadopoulou	ICCS
Tom Vander Aa	IMEC
Enrico Calore	INFN
Sebastiano Fabio Schifano	INFN
Francesco Simula	INFN
Georgios Smaragdos	NEUR
Jan-Harm Betting	NEUR
Christos Strydis	NEUR
Sergi Siso	STFC
Nuno Miguel Nobre	UNIMAN
William Toms	UNIMAN

Deliverable details

Dissemination Level		
PU	X	Public
PP		Restricted to other programme participants
RE		Restricted to a group specified by the consortium
CO		Confidential, only for member of the consortium
		Where restricted, access granted to:

Nature		
R	X	Report
P		Prototype
D		Demonstrator
O		Other

Review status	
X	Draft
	WP Leader accepted
	QA approved
	Coordinator accepted

History chart

ISSUE	DATE	CHANGED PAGE(S)	CAUSE OF CHANGE	IMPLEMENTED BY
0.1	2021/06/02	All sections	New document template	INAF
0.1	2021/07/03	All sections	Partner contributions	ALL
0.2	2021/07/10	All Section + Introduction	Added introduction, section rearrange and editing	INAF
0.2	2021/07/14	Section 5,9	Add new figures	INAF, INFN

0.3	2021/07/14	Introduction, Summary	Revision	INAF
0.4	2021/07/15	Section 9 update	Revision	INAF
0.4	2021/07/15	Introduction and summary	Revision	INAF
0.5	2021/07/21	Integration form partners	Revision	ALL
0.7	2021/07/23	Reviewers comments:	Revision	ALL
0.8	2021/07/27	Integration	Sections revision	UNIMAN, INAF, NEUR
1.0	2021/07/31	Final version		INAF
1.1	2022/01/12	Integration	MAX5 porting and LOFAR portint	INAF, NEUR
1.1.1	2022/01/17	Integration	TB3 porting details	ECMWF
1.1.2	2022/01/18	Integration	Lattice Boltzmann updates	INFN
1.1.3	2022/01/20	Integration	Porting Updates	FRAUN, FORTH
1.1.4	2022/01/22	REVIEW	Internal review	INFN, ICCS
1.1.5	2022/01/30	Integrations	Revision	FRAUN, FORTH, INAF
1.1.6	2022/05/20	Submitted version	Submission	INAF

EuroEXA Consortium

No	Participant organization name	Short name	Country
1 (Coordinator)	INSTITUTE OF COMMUNICATION AND COMPUTER SYSTEMS	ICCS	Greece
2	THE UNIVERSITY OF MANCHESTER	UNIMAN	United Kingdom
3	BARCELONA SUPERCOMPUTING CENTER - CENTRO NACIONAL DE SUPERCOMPUTACION	BSC	Spain
4	FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS	FORTH	Greece
5	SCIENCE AND TECHNOLOGY FACILITIES COUNCIL	STFC	United Kingdom
6	INTERUNIVERSITAIR MICRO-ELECTRONICA CENTRUM	IMEC	Belgium
7	ZEROPOINT TECHNOLOGIES AB	ZeroPoint	Sweden
8	ICEOTOPE RESEARCH & DEVELOPMENT LTD	ICE	United Kingdom
9	ARM LIMITED	ARM	United Kingdom
10	SYNELIXIS LYSEIS PLIROFORIKIS AUTOMATISMOU & TILEPIKOINONION MONOPROSOPI EPE	SYNELIXIS	Greece
11	MAXELER TECHNOLOGIES LIMITED	Maxeler	United Kingdom

12	NEURASMUS BV	Neurasmus BV	Netherlands
13	ISTITUTO NAZIONALE DI FISICA NUCLEARE	INFN	Italy
14	ISTITUTO NAZIONALE DI ASTROFISICA	INAF	Italy
15	EUROPEAN CENTRE FOR MEDIUM-RANGE WEATHER FORECASTS	ECMWF	United Kingdom
16	FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V.	Fraunhofer	Germany

Table of Contents

List of Tables

List of Figures

1 Introduction

In this deliverable we discuss the final effort in porting and optimization applications, which includes the re-design of application software infrastructure, the identification and optimization of algorithms, the optimization of applications networking and communication, the optimization of the IO.

The main goals of Work Package 2 was to contribute in the design, development and evaluation of the EuroEXA platform starting from collecting the different Applications requirements (D2.1) and actively contributing to the co-design (D2.3 and D2.4). A set of key applications has been re-designed and optimized to evaluate the platform and more in general to develop a set of relevant Exa-scale ready applications meant as examples of the potential of scientific exploitation of new architectures. The different phases of porting and optimizations have been discussed in D2.2 and D2.5.

The adaptation, porting and optimization activities were the objective of task 2.2 and they impact on co-design task 2.3 and on the evaluation of the platform (task 2.5). Those tasks impact several areas in which EuroEXA was committed to advance and optimize the HPC technologies (see *e.g.* EuroEXA Objectives 1, 7, 8, 9 and 10, Result 2 and Result 5).

Each application differs in terms of algorithms and architecture and it has different performance and data requirements (see D2.1). For this reason applications implement specific porting and optimization strategies, timeline for adaptation and requirements in terms of programming models and runtimes. The porting and optimization strategy and timeline for each application has been identified and is documented in this deliverable. A large effort has been done to adapt and optimize applications towards a full Exa-scale platform that is a heterogeneous system combining general-purpose processors with accelerators, high-speed networks and composite storage. This implies a full redesign of algorithms not limited to the use of FPGAs but also to optimize network utilization, IO performance and memory usage. Some of the re-design activities presented in this document have general perspective in the frame of re-engineering and adapting the scientific HPC code to the future Exa-scale platforms which have the same high-level feature of EuroEXA project. In fact, enhancing the adaptability, the memory-hierarchy awareness while lowering the amount of data travelling over the network are key features the applications must be equipped with. Moreover, a good design that allows to “swap” fundamental building blocks and/or algorithms and their implementation is also of fundamental importance.

During porting and adaptation, applications developers matured an important experience in terms of FPGAs and more in general EuroEXA Platform utilization that we discuss specifically for each application and we summarize in the final section of this deliverable.

The document also clarifies the form of the delivered software and licensing and the requested co-design features that have been implemented by runtime and hardware developers.

Notice that due to the delay in Testbed development and implementation, in this deliverable we emphasize on the porting and evaluation of most applications in the EuroEXA runtimes and on single and dual CRDBs nodes. Further optimization will be done as quad-CRDB and TB3 will be available.

1.1 Document Structure

The remainder of this document is structured as follows: Sections 2 to 12 are divided into three main subsections. The 1st subsection describes the porting and optimization efforts performed by each partner contributing an application in EuroEXA as part of WP2, while the 2nd subsection describes the EuroEXA specific features that they are exploiting (including runtimes) The 3rd subsection presents the lesson learned in porting applications on the EuroEXA testbed. Finally, in Section 13 we summarize the work done and we present an overview of the porting issues and lessons learned.

Table 1 Programming models for EuroEXA application software

	Programming models - Initial	Programming models - Ported
InfOli	C	MaxJ, OmpSs@Cluster, Vivado HLS, OpenMP
NEMO	MPI, OpenMP	OpenCL
Neuromarketing	C/C++, OpenMP, MPI	multi node using OmpSs@FPGA, Vivado HLS
DPSNN	C/C++, MPI	Vivado HLS
LBM	MPI, OpenMP	Vivado HLS, OmpSs@FPGA, GASPI (GPI-2)
LFRic	MPI, OpenMP	Vivado HLS
FRTM	GASPI (GPI-2)	GPI-2, Vivado HLS
SMURFF	GASPI, MPI, OpenMP	Vivado HLS, OmpSs@FPGA, MPI
GADGET	MPI, OpenMP	OmpSs@FPGA, MPI, OpenMP
AVU-GaiaGSR	MPI, OpenMP	MPI, OmpSs, OpenMP, OpenACC
LOFAR	C++11 multi-threading	MPI, OmpSs@FPGA
Alya	MPI, OpenMP, CUDA,	MPI, OmpSs@Cluster

	OpenACC, OmpSs	
ESCAPE dwarves	OpenMP	MaxJ
Satellite Image Processing	CUDA	Vivado HLS

Table 2 Platform requirements for EuroEXA application software. We report the initial requirements and final platform and architecture used to port and optimize the applications (e.g. CRDB TB2, Quad-CRDB, TB3, etc).

	Platform requirements - Initial			PR Ported (M27)			PR Planned (M28-M36)			Final Porting and Optimization		
	Arch.	Type	Accel.	Arch.	Type	Accel.	Arch.	Type	Accel.	Arch.	Type	Accel.
InfOli	x86	Single core		Maxeler DFE	Single node	Yes	Maxeler DFE	Single-node	Yes	TB3 simulator Maxeler DFE		Yes
				ARM	Multi-node		ARM	Multi-node		MareNostrum 4	Multi-Node	No
				Xilinx FPGA	Single node	Yes	Xilinx FPGA	Single node	Yes			Yes
NEMO	x86	Multi-node		Xilinx FPGA	Single node	Yes	x86 Xilinx FPGA	Multi-node	Yes	x86 Xilinx FPGA	Multi-node	Yes
				ARM	Single node		ARM	Multi-node	No	ARM	Multi-node	No

Neuro marketing	x86/ARM	Single core		Xilinx FPGA	Multi-node	Yes	ARM Xilinx FPGA	Multi-node	Yes	CRDB	Multi-Node	yes
DPSNN	x86/ARM	Multi-node		Xilinx FPGA	Single node	Yes	ARM Xilinx FPGA	Multi-node	Yes	CRDB	Single-Node	Yes
LBM	x86	Multi-node	Yes (GPUs)	Xilinx FPGA	Single node	Yes	Xilinx FPGA	Multi-node	Yes	CRDB	Multi-Node	Yes
				Arm	multi-node		Arm	Multi-node		Arm	Multi-node	Yes
				x86	multi-node		x86	Multi-node		x86	Multi-node	Yes
LFRic	x86/ARM	Multi-node		Xilinx FPGA	single-node	Yes	ARM Xilinx FPGA	Multi-node	Yes	CRDB	Single-Node	Yes
FRTM	x86	Multi-node		ARM+ Xilinx FPGA	single-node	Yes	ARM Xilinx FPGA	Multi-node	Yes	CRDB	Multi-Node	Yes
SMURFF	x86	Multi-node		ARM + Xilinx FPGA	single-node	Yes	ARM Xilinx FPGA	Multi node	Yes	CRDB	Single-Node	

							Maxeler DFE					
GADGET	x86/ARM	Multi-node		ARM	multi-node	Yes	ARM Xilinx FPGA	Multi-node	Yes	CRDB	Single-Node	Yes
AVU-GaiaGSR	x86	Multi-node		ARM	multi-node		ARM Xilinx FPGA	Multi-node	Yes	CRDB	Single-Node	Yes
LOFAR	x86	Single node	Image Domain Gridder (GPUs)	ARM	Single node	Image Domain Gridder (GPUs)	ARM		Image Domain Gridder (GPUs)	Arm	Multi-Node	Yes
Alya	x86/ARM	Multi-node	Yes (GPUs)				ARM	Multi-node		Marenostrum	Multi-Node	NO
ESCAPE dwarves	x86	Single node		ARM	Single node		Xilinx FPGA Maxeler DFE	Single-node	Yes	CRDB	Single-Node	Yes

IFS	x86	Multi-node		ARM	Multi-node		ARM	Multi-node		CRDB	Single-Node	No
Satellite Image Processing	x86	Single node	Yes (GPUs)	QFDB	Single node	Yes	TB2 Devel Testbed	Multi-node	Yes	CRDB	Single-Node	Yes

EuroEXA – H2020 – EU.1.2.2. – FET Proactive

2 NEUR applications

2.1 Biophysically realistic simulator of the brain Inferior Olive (InfOli)

As a reminder from previous deliverables, the InfOli application targets three platforms: 1) the Maxeler DFE (Data Flow Engine), 2) the multi-node cluster based on the OmpSs@Cluster programming model and 3) FPGA acceleration using the OmpSs@FPGA programming model. The application has two basic computational blocks: i) the Gap-Junction Computations that model the connectivity between the inferior-olivary nucleus, and ii) the Neuronal-Compartment Computations. Since there are real data dependencies across simulation steps (this is a transient simulator that constantly solves the same Ordinary-Differential-Equation system), the application can be parallelized only in space and not in time.

The Neuronal-Compartment Computations are a purely dataflow block. In a version of the application where no interneuron connectivity is modelled, the InfOli application is embarrassingly parallel and able to exploit parallelism in the best way possible. The gap-junction connectivity complicates things a bit further. It breaks the dataflow nature of the application, since the gap-junction loop is required to be finished before the compartment computation is finished. This makes the area usage of the FPGA slightly less efficient, as the logic dedicated to the compartment computations is required to waste operation ticks till the gap-junction computation is finished. Nevertheless, both blocks can be accelerated using typical parallelization techniques such as loop unrolling for the gap-junction block and fine-grain pipelining for the compartment computations.

In this last project year, work was focused on optimizing the Maxeler port and developing the OmpSs@FPGA and OmpSs ports.

2.1.1 Final Ported Application Software

Maxeler Port

The DFE implementation of the InfOli application is depicted in Figure 2.1. It implements 3 internal pipelines, one for each part of the neuron (Dendrite, Soma, Axon), each performing the respective Neuron Compartment Computations. The state parameters for every neuron are stored in separate BRAM blocks (one per state variable) for faster and higher-bandwidth access

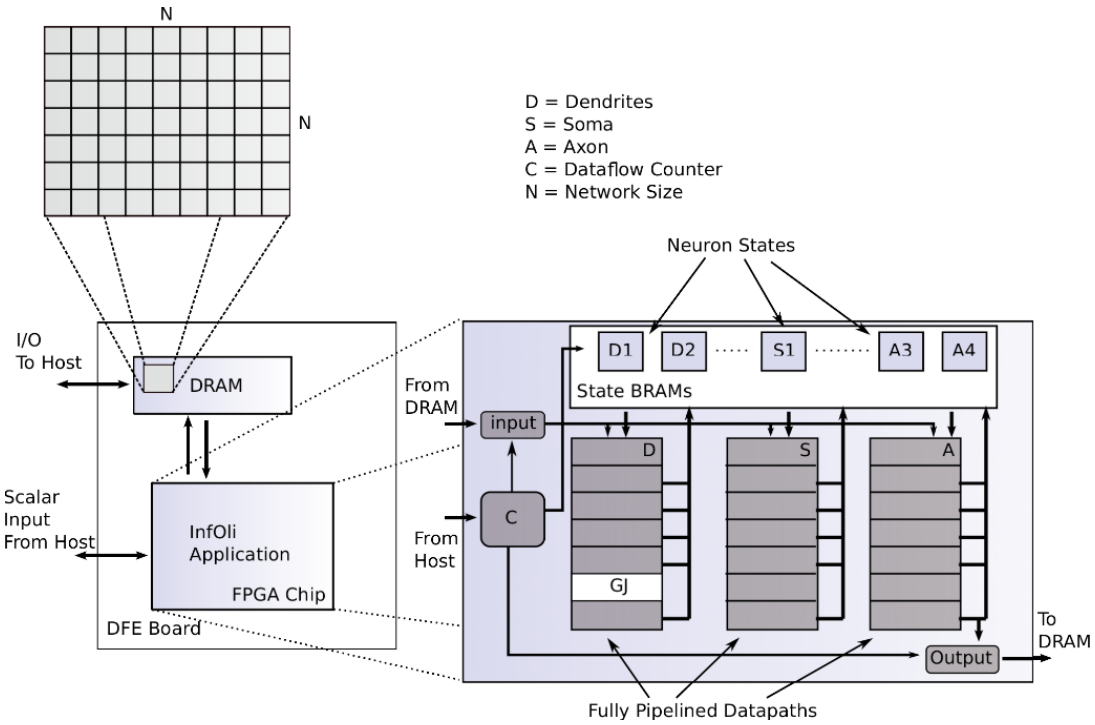


Figure 2.1 DFE implementation of InfOli

Using the ODE libraries

In the previous years, the ODE libraries, flexHH, have been developed as part of Deliverable 3.3. Initial porting was on MAX4 and then they have been ported to MAX5. They are now mature enough to actually be used to support the InfOli application and are currently being developed and validated even for multi-node support. Thus, from this point on, for executing and evaluating the InfOli application on the Maxeler platform, we will be using the flexHH application as it provides a natural path to an InfOli Maxeler multi-accelerator port.

MAX5 port

Porting to the MAX5 hardware requires a few simple changes on source code to adhere to the updated Maxeler toolflow (2018.3). A summary of the changes required can be seen on Table 3

Table 3 Maxeler toolflow command update

Older Commands	*New Commands*

DebugLevel dbg = new DebugLevel();
dbg.setHasStreamStatus(true);
debug.setDebugLevel(dbg);
	setHasStreamStatus(true);

config.setAllowNonMultipleTransitions(true);	setAllowNonMultipleTransitions(true);
DFELink cpu2lmem = addStreamToOnCardMemory	DFELink cpu2lmem = iface.addStreamToLMem("cpu2lmem",

("cpu2lmem", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);	LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);	DFELink lmem2cpu = iface.addStreamFromLMem("lmem2cpu", LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
DFELink ini = addStreamFromOnCardMemory("Ini", MemoryAccessPattern.LINEAR_1D);	DFELink ini = iface.addStreamFromLMem("Ini", LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
DFELink iApp = addStreamFromOnCardMemory("iApp", MemoryAccessPattern.LINEAR_1D);	DFELink iApp = iface.addStreamFromLMem("iApp", LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
DFELink IC = addStreamFromOnCardMemory("IC", MemoryAccessPattern.LINEAR_1D);	DFELink IC = iface.addStreamFromLMem("IC", LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
DFELink NeuronChar = addStreamFromOnCardMemory("NeuronChar", kernel.getOutput("NeurCondStream"));	DFELink NeuronChar = iface.addStreamFromLMem("NeuronChar", kernel.getOutput("NeurCondStream"));
DFELink Conn = addStreamFromOnCardMemory("Conn", kernel.getOutput("ConnStream"));	DFELink Conn = iface.addStreamFromLMem("Conn", kernel.getOutput("ConnStream"));
configBuild method	

buildConfig.setMPPRCostTableSearchRange(params.getMPPRStart(), params.getMPPREnd());
 	
buildConfig.setMPPRParallelism(params.getMPPRNumThreads());
 	
buildConfig.setMPPRRetryNearMissesThreshold (params.getMPPRRetryThreshold());
	buildConfig.setEnableTimingAnalysis(true);
	
	buildConfig.setParallelism(params.getMPPRNumThreads());
	
	buildConfig.setImplementationNearMissThreshold (params.getMPPRRetryThreshold());
	
	buildConfig.setImplementationStrategies (ImplementationStrategy.PERFORMANCE_NET_DELAY_HIGH);

Besides the port to MAX5 the code is changed in favour of portability. This is done by creating an interface for the manager. With the use of this interface the platform specific hardware, such as I/O and resource usage is splitted from the platform independent kernel and consequently making porting between different platforms/hardware easier and less time consuming.

Kernel

The implementation of flexHH on MAX5 consists of one cell kernel and two gap kernels.This is shown in Figure 2.2.

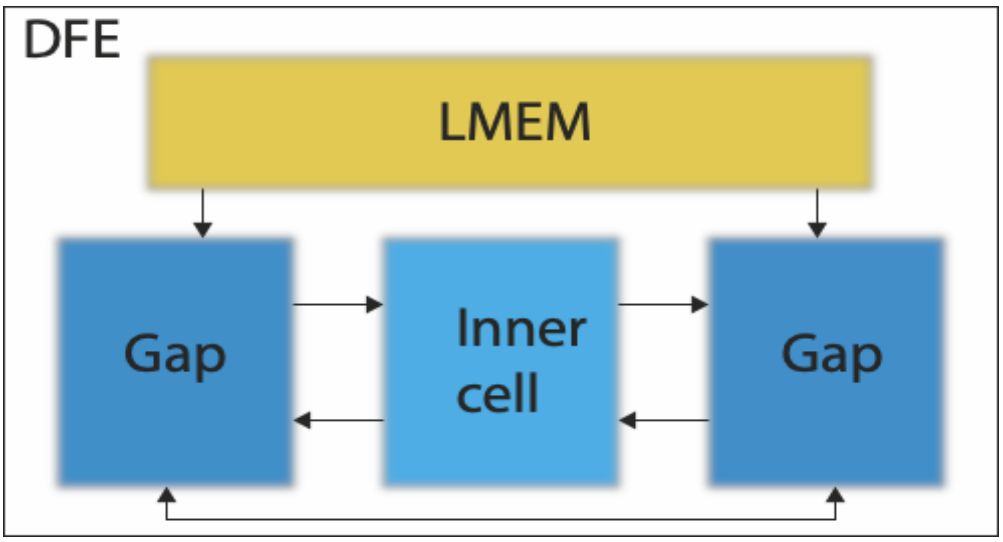


Figure 2.2 Visual representation of kernels on a single DFE node

The cell kernel calculates the inner cell dynamics and the gap kernel calculates the dynamics of the gap junctions. For the dynamics of the gap-junctions the voltages of the other gap kernels are required. Consequently, for the support of multiple accelerators the voltages of each gap kernel are required to be transferred to each other gap kernel. In order to implement this the dynamics used to calculate the gap-junction currents will be calculated in stages. In each of these stages the voltages of one gap kernel will be used to update the gap-junction current. The process of calculations is shown in Equations (2.1) and (2.2), for the case when the number of DFEs (N_{DFEs}) is equal to 4.

$$I_{gap,i} = \sum_{j=0}^{N_{cells}-1} (w_{i,j}(c_0 \exp(c_1 \cdot (V_i - V_j)^2) + c_2)(V_i - V_j)) \tag{2.1}$$

$$\begin{aligned} &= \sum_{j=0}^{N_{Cells}/(N_{DFEs=4})} (w_{i,j}(c_0 \exp(c_1 \cdot (V_i - Vs_{DFE,0}[j])^2) + c_2)(V_i - Vs_{DFE,0}[j])) \\ &+ \sum_{j=0}^{N_{Cells}/(N_{DFEs=4})} (w_{i,j}(c_0 \exp(c_1 \cdot (V_i - Vs_{DFE,1}[j])^2) + c_2)(V_i - Vs_{DFE,1}[j])) \\ &+ \sum_{j=0}^{N_{Cells}/(N_{DFEs=4})} (w_{i,j}(c_0 \exp(c_1 \cdot (V_i - Vs_{DFE,2}[j])^2) + c_2)(V_i - Vs_{DFE,2}[j])) \\ &+ \sum_{j=0}^{N_{Cells}/(N_{DFEs=4})} (w_{i,j}(c_0 \exp(c_1 \cdot (V_i - Vs_{DFE,3}[j])^2) + c_2)(V_i - Vs_{DFE,3}[j])) \end{aligned} \tag{2.2}$$

For the program three different phases can be distinguished:

- **Start:** During the start of the program, the junction currents are partially calculated with the voltages which are stored on their own DFEs and these voltages are sent to the next neighbouring DFE. Therefore, the calculations and transferring of data happen concurrently.
- **Middle:** During this phase, the gap-junction currents are already updated with the part for which the voltages of the own DFE are responsible. Furthermore, as in the previous phase the voltages were already sent to the next DFE, the gap junction calculations can continue immediately. The only difference with the previous phase is that voltages were received from the MaxRing, instead of being stored in FMem on the DFE. Furthermore, again the voltages are sent to the next neighbouring DFE, as these will be required there during the next stage.
- **End:** During this phase, the final voltages are received and the final values of the gap-junction currents are calculated. Thereafter, the voltages are updated. Furthermore, no data is sent as all the voltages were already passed around.

The working of these phases is visually presented in Figure 2.3. For simplicity only a single gap kernel is shown, instead of 2 gap kernels and 1 cell kernel; in fact the flow does not change depending on how many kernels there are on a single DFE.

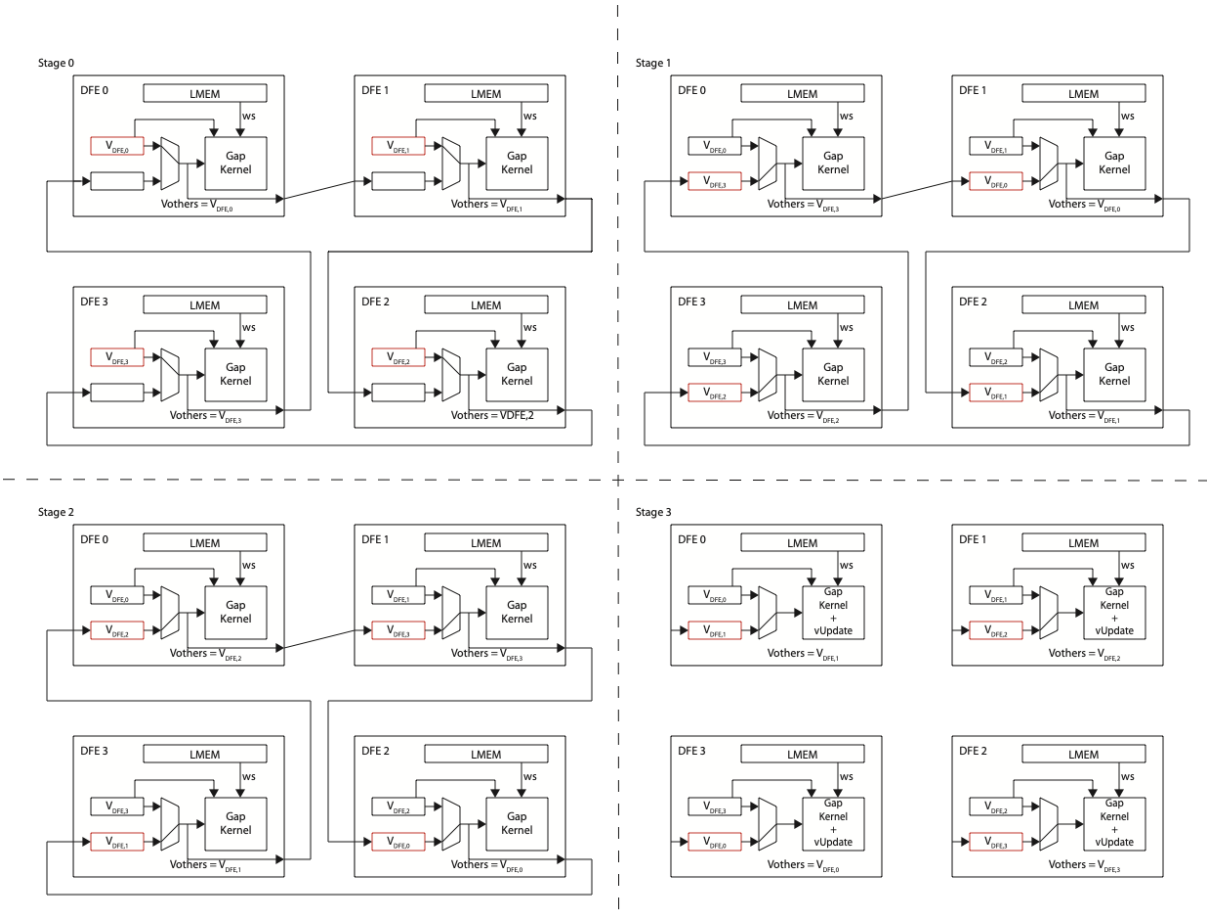


Figure 2.3 Visual representation of program flow

This program flow has shown to work in simulations. The number of gap kernels per cell kernel is a hardware parameter as the gap kernels and connections to and from the cell kernel are required in hardware. Consequently, this allows for easier ports to different hardware. Furthermore, the number of DFEs is a run-time parameter. Therefore, this number can be changed based on the topology of the system without the time-consuming synthesis of new kernels.

PLATFORM REQUIREMENTS	Required	Achieved/Implemented
Hardware	Max5 DFE	yes
Operating system	CentOS 7.x	yes
Languages	C, MaxJ	yes
Compilers	Maxcompiler, Xilinx	yes

Programming models	MaxJ	yes
Low-level libraries	-	
High-level libraries	-	

License

Copyright Holder: Neurasmus B.V. Contributor: Rene Miedema - published under LGPL.v3.

OmpSs@FPGA Port

Figure 2.4 depicts the general block diagram of the FPGA accelerated kernel using the Vivado HLS tool-flow. The kernel consists of multiple identical parallel neuron-processing modules, each modelling the dendrite, soma and axon parts of a single InfOli cell. The design further includes a set of BRAMs for storing the evoked input to the network, the connectivity matrix for the gap-junctions as well as the neuron states, which are updated after each simulation step. The kernel provides acceleration by the parallel execution of the compartment computations, the pipelining of the gap junction computations and by time-multiplexing the kernels hardware to run multiple simulated cells within an execution run.

Currently, what has been delivered is the optimized kernel, developed and validated using the Vivado HLS toolflow, targeting the Zedboard/Zynq hardware. The kernel was validated on the Vivado 2017.3 environment. It includes only a few instances of the kernel, as the validation originally targeted the ZedBoard device, but can be easily extended to exploit chip area that would be available on larger FPGAs. We developed the algorithm using traditional tools (Xilinx SDx) and successfully validated it on a Xilinx ZedBoard FPGA for a small number of neurons.

We ported the system to OmpSs@FPGA and tested it successfully on a Zynq UltraSCALE+ development board. The next step will consist of cleaning up the code and moving to the testbed. The kernel source code and the preliminary OmpSs@FPGA code are available on the EuroEXA repository (https://github.com/euroexa/infoli/tree/master/OmpSs%20FPGA) and will be later available in a public repository.

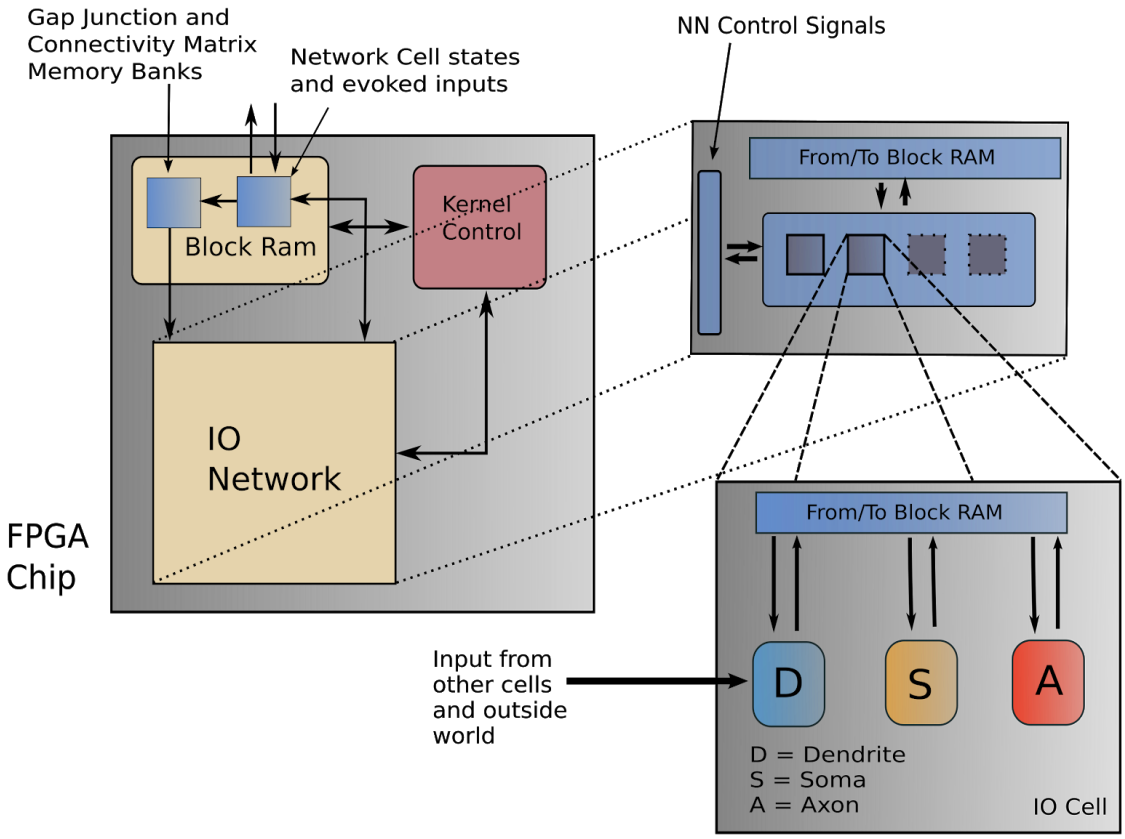


Figure 2.4 Block diagram for the FPGA-accelerated kernel of InfOli

PLATFORM REQUIREMENTS	Required	Achieved/implemented
Hardware	Zedboard, EuroEXA TB2	yes
Operating system	Any OS supporting Xilinx tools	yes
Languages	C/C++	yes
Compilers	Vivado HLS, OMPSs+FPGA	yes
Programming models	-	
Low-level libraries	-	
High-level libraries	-	

OmpSs@Cluster port for distributed memory

The OmpSs@cluster implementation of InfOli is based on the MPI version, which was first simplified to remove the explicit MPI communication. The code has the two computational kernels mentioned in the above section, corresponding to the Gap-Junction Computations and the Neuronal-Compartment Computations, respectively. A sketch of the high-level of the OmpSs@cluster implementation is given in Figure 2.5. The outer loop iterates over the timesteps of the computation. Inside the loop, the gap-junction computations read the voltages of the neighbouring neurons, as indexed by the neighbour Id, neighId, from the full array of neuron voltages, V_dend. The overall result is the conductance of each neuron, which is written to I_c. A single task is offloaded to each node, using the “pragma oss task” annotation, which provides the direction of each function argument and the destination node via the node(nodeNum) clause. The offloaded tasks have weak dependencies (weakin, weakinout or weakout), meaning that these tasks do not access the data directly. Instead, they create sufficient subtasks or “task for”’s to keep all cores busy, and these subtasks/task for’s access the data, via strong (normal) dependencies. The (parent) tasks can therefore be immediately offloaded, before the data is available, and can therefore build a distributed dependency graph and instantiate work in advance. Following the gap-junction computations, the neuronal compartment computations take as input the conductances calculated by the gap-junction functions, and they update all the variables in the state of the neuron, including the voltage that is input to the gap-junction calculations in the next timestep. Similarly to the gap-junction functions, a single task with weak dependencies is offloaded to each node, which instantiates the actual work to keep all computing cores busy.

The cross-node dependencies are those from the weakinout(V_dend[target_cell; cells_per_node]) dependency of the neuronal-compartment computations to weakin(V_dend[0; IO_NETWORK_SIZE]) dependency of the gap-junction computations. In addition to enforcing these dependencies, Nanos6@cluster performs all necessary data transfers. Here, the dependency describes the access type (e.g. weakinout), the name of the array (e.g. V_dend), the offset of the first element (e.g. target_cell) and the number of elements (e.g. cells_per_node). Unlike OpenMP, OmpSs@cluster supports region dependencies with fragmentation. In this case, each compartment calculation task computes a subregion of the array and each gap-junction computation task reads the entire array. This is a concise way to describe an all-to-all dependency.

The compartment computations have a large number of dependencies in the task annotation, owing to the original organization of the program using a separate array for each scalar value. Reorganizing the state as an Array of Structures (AoS) may slightly decrease the runtime overhead, as it would reduce the number of dependencies that need to be monitored by the runtime system. But it is a somewhat intrusive optimization, and, crucially, all these dependencies pass from a task to a successor task on the same node. Such dependencies are automatically optimized by the runtime system to not leave that node, and except at the beginning and end of the computation they do not typically involve the parent in any way, so the overhead is small. This optimization is described in D3.3.

Overall, no significant changes were required to the structure of the InfOli application or the contents of the computing kernels.

```
    for(int sim_step = 0; sim_step < total_simulation_steps; ++sim_step)
    {
       // Gap junction computations
       for (int nodeNum = 0; nodeNum < numNodes; nodeNum ++)
       {
           int target_cell = nodeNum * cells_per_node;
           #pragma oss task weakin(V_dend[0; IO_NETWORK_SIZE])                        \
               weakin(neighId[nodeNum *neighbours_per_node;neighbours_per_node]) \
               weakin(neededCond[nodeNum *neighbours_per_node;neighbours_per_chunk])   \
               weakout(I_c[nodeNum * cells_per_node; cells_per_node])                  \
               node(nodeNum) label("gjf_weak_task")
               {
                       // ...
               }
       }
       // Neuronal-Compartment computations
       for (int node = 0; node < numNodes; node++)
       {
           int target_cell = nodeNum * cells_per_node;
           #pragma oss task
               weakinout(I_c[target_cell; cells_per_node])             \
               weakinout(V_dend[target_cell; cells_per_node])          \
               weakinout(V_soma[target_cell; cells_per_node])          \
               weakinout(V_axon[target_cell; cells_per_node])          \
               weakinout(Hcurrent_q[target_cell; cells_per_node])      \
               weakinout(Calcium_r[target_cell; cells_per_node)        \
               weakinout(Ca2Plus[target_cell; cells_per_node]) \
               weakinout(Potassium_s[target_cell;cells_per_node])      \
               weakinout(I_CaH[target_cell; cells_per_node])           \
               weakinout(Calcium_k[target_cell; cells_per_node])       \
               weakinout(Calcium_l[target_cell; cells_per_node])       \
               weakinout(Sodium_m[target_cell; cells_per_node])        \
               weakinout(Sodium_h[target_cell; cells_per_node])        \
               weakinout(Potassium_n[target_cell;cells_per_node])      \
               weakinout(Potassium_p[target_cell; cells_per_node])     \
               weakinout(Potassium_x_s[target_cell;cells_per_node])    \
               weakinout(Sodium_h_a[target_cell; cells_per_node])      \
               weakinout(Potassium_x_a[target_cell; cells_per_node])   \
               weakinout(Sodium_m_a[target_cell; cells_per_node])      \
               weakin(iAppIn[target_cell; cells_per_node])             \
               weakin(g_CaL[target_cell; cells_per_node])              \
               node(nodeNum) label("updateState_weak_task")
               {
                       // ...
               }
       }
}
```

Figure 2.5: Outline of OmpSs@Cluster implementation of InfOli

Although porting the application was straightforward, initial performance was poor as the application exposed several performance issues in the Nanos6@cluster runtime. The

modifications to address these issues are reported in D3.3 and overall, they provided more than 10x improvement in performance. In brief, firstly, the stability and scope of Extrae tracing were improved to provide full visibility of runtime behaviour, including features for expert users to identify where the time is spent in the runtime system. Secondly, the excessive number of control messages was reduced by disabling fine-grained early release of dependencies for offloaded tasks when the successor task is not on the same node. Thirdly, it was found that a race condition meant that there were multiple duplicate data transfers, which was resolved using a combination of versioning of data dependencies and modifications to the dependency system. Fourthly, "*task for*"s were made compatible with OmpSs@Cluster, which avoids the overhead of fine-grained dependencies.

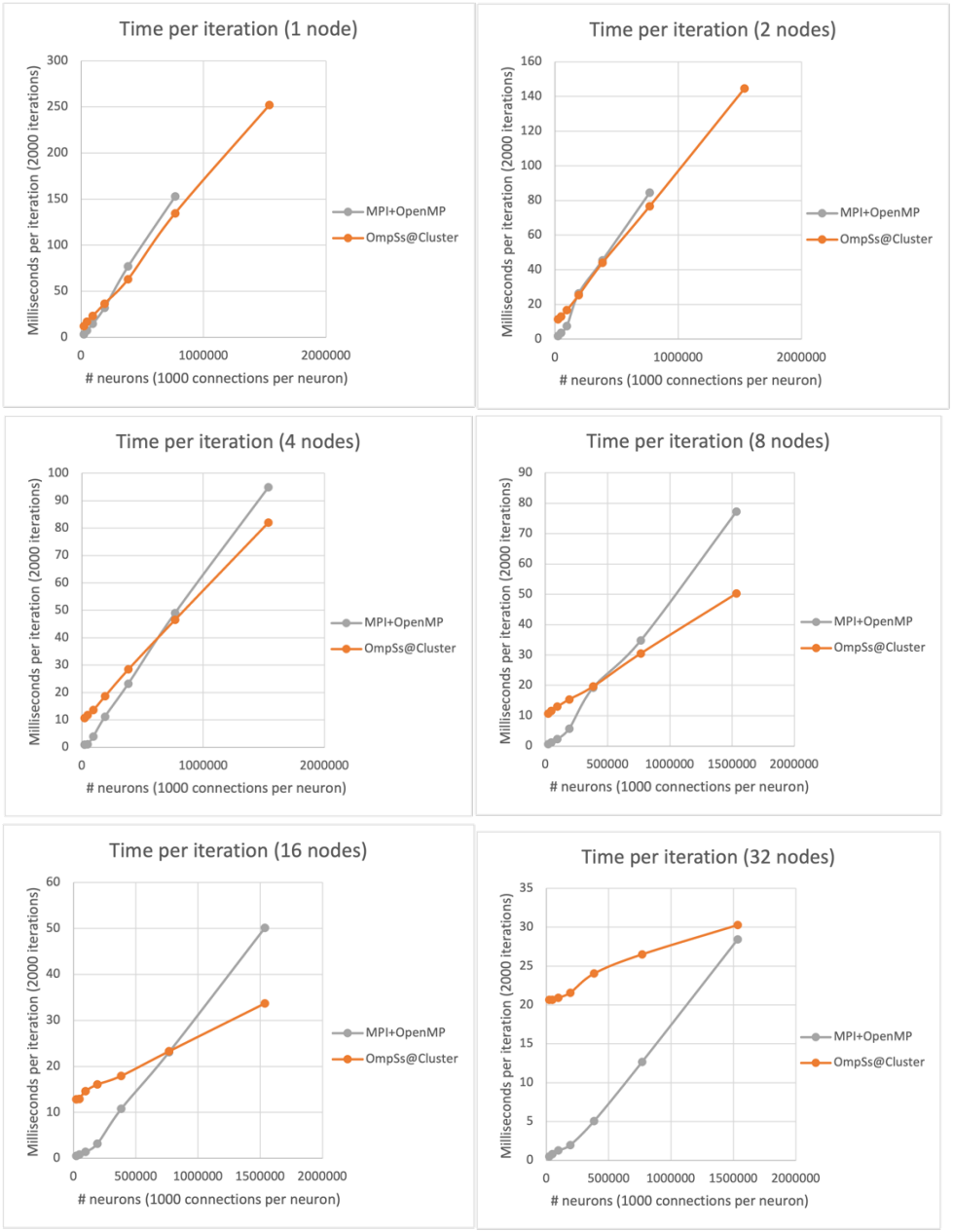


Figure 2.6: Comparison of MPI and OmpSs@Cluster implementations on 1 to 32 nodes of MareNostrum 4

Figure 2.6 shows a comparison of the execution time per timestep for the MPI+OpenMP and OmpSs@cluster versions, from 1 to 32 nodes, on BSC's MareNostrum 4. The computational kernels for the two versions were compiled using the same compiler, with the same settings. In all plots, the *x*-axis is the number of neurons, up to about 1.5 million, and the *y*-axis is the execution time per timestep, in milliseconds. As can be seen, OmpSs@cluster introduces a significant overhead for small problem sizes, but the overhead becomes less significant as the problem size increases. In fact, the OmpSs@cluster runtime achieves a better computation–communication overlap than MPI+OpenMP due to the asynchronous execution. The maximum

benefit is seen on 8 nodes, where the OmpSs version is 40% faster at the largest problem size. However, the increasing overhead catches up and the MPI version is better on 32 nodes

PLATFORM REQUIREMENTS	Required	Achieved/implemented
Hardware	Distributed Memory Systems	yes
Operating system	GNU based Linux OS	yes
Languages	C/C++	yes
Compilers	Mercurium, GCC	yes
Programming models	OmpSs-2@Cluster	yes
Low-level libraries	-	-
High-level libraries	-	-

2.1.2 Unique EuroEXA Features Supported and Exploited

OmpSs@FPGA Port

The OmpSs@FPGA port makes use of several features of the EuroEXA platform, making it easier to convert the original Vivado C code to a synthesizable kernel. First of all, whereas the original Vivado C code required to define and configure the data flows between host and device using a complicated graphical interface, OmpSs@FPGA allows us to simply define the clock speed and the output device in a *makefile* and generate the bitstream; it takes care of the data transfers automatically. Whereas the SDSoC implementation for infOli generates a complete system image that has to be placed on an SD card in order to boot the board from, OmpSs@FPGA allows the generation of a single bitstream file, which can be placed in an existing Linux installation, after which the kernel can be loaded into the FPGA.

Maxeler Port

The Maxeler port benefits from most of the Maxeler toolflow and platform unique features. The level of abstraction and the optimization of the platform for dataflow applications, which the InfOli application largely is, allows for great performance benefits that would be far more time-consuming to accomplish using traditional FPGA design. The low-level hardware control that is allowed through the tool-flow also allows for potentially even more low-level optimization, similar to what is currently being attempted with the neuron libraries that Neurasmus is developing for WP3. The same method can potentially be applied to the hardcoded InfOli application as well. The ability to have fine grained control of the arithmetic units used within MaxJ facilitates this potential. The code is also immediately compatible with the Maxeler-developed MAX5 hardware, VU9-based Xilinx Alveo cards and Amazon F1 instances besides the use on the EuroEXA platform through the Maxeler runtimes

Distributed memory OmpSs-2@Cluster Port

Leveraging OmpSs-2@Cluster, the InfOli application benefited from the straightforward transformation of the existing MPI code into OmpSs-2@Cluster version without significant changes to the overall structure of the original computing kernels. The user surrounds each computing kernel inside the main simulation loop with a *task* pragma directive. Seamlessly, the user then describes dependencies between each task by defining all shared data and memory accesses between these tasks via in/out/inout clauses. The code is then compiled by Mercurium source-to-source compiler that translates the source code written with OmpSs-2@Cluster directives into a parallel task-ified version. The Nanos6 (the runtime implementation of the OmpSs-2@Cluster programming model) would then take control of everything during the program execution, hiding all intricate details and operations from the user such as: compute and link dependencies between different tasks, send and fetch data between computing nodes, manage memory allocation on different nodes, taking scheduling decisions automatically or based on a user-specific choice using an additional node clause. In addition, the runtime allows the user to manually configure (via environment variables or .toml file) multiple parameters before running the application, such as scheduler type, CPU threads management policy, virtual memory size and others.

2.1.3 Porting Issues and Lessons Learned

OmpSs@FPGA Port

We had some issues when porting our code from traditional Xilinx tools to OmpSs@FPGA to run it on the ZedBoard. Even though our application could be synthesized without issues on Xilinx tools, the OmpSs@FPGA tools gave error messages that the area was too small. This was resolved by using a different board. However, this board needed to be ordered and delivered to us, which took a significant amount of time. There were a few issues with the OmpSs@FPGA tools, which were still in development, but we had good communication with BSC and thus gradually resolved all issues.

Maxeler Port

With the knowledge of implementing flexHH on a single node, we decided to first develop the multi-node implementation using the simulator tools of Maxeler. This decision was made to limit time-consuming hardware synthetizations and easier debugging. However, MaxRing, the high-speed interconnect required to use the multi-node in hardware, is not supported in simulation mode. Therefore, we had to mimic its behaviour by creating an artificial connection between kernels to simulate the MaxRing. Another difficulty was porting between generations (MAX4 and MAX5). Because of the differences between architectures, efficient use of the hardware resources is not transferable between generations and the newer port required some further exploration. This issue presented itself in timing errors and redoing the exploration of hardware parameters of

the kernel (the maximum number of cells and the unroll/parallelization factor). For the timing errors, the support by Maxeler was essential.

OmpSs Port

Obtaining a taskified OmpSs-2@Cluster version of the existing MPI version was a straightforward process. However, the issue was that the original code uses a large number of huge data arrays. Hence, we had to ensure that each task accesses the correct subset of the array, especially if the array size was not divisible by the number of CPU cores in the system.

In addition, the InfOli application pattern exposed a couple of issues with the way the Nanos6 runtime handles specific cases internally that was very helpful to solve these issues. The first issue is that we needed an explicit way to schedule particular tasks on a specific node. We solved this by introducing the node clause that would take a node number as an argument; then, the runtime would thereafter use this as a scheduling decision.

Another issue was that each task requires performing data copy for all sub-arrays accesses needed for that task, stressing the communication layer and increasing the number of messages sent and received by the runtime, consequently leading to poor scalability as the number of nodes increases and as the simulation dataset size increases. The solution introduced was to perform a single big data copy at the beginning of each iteration and group all messages, then send them at once.

Finally, OmpSs@Cluster brings a more productive programming model that, in its pure form, avoids the need to develop and maintain MPI code. In the case of InfOli, we started from an initially developed MPI version, which first had to be simplified, so the development cost for the MPI version was already paid. Nevertheless, the OmpSs@Cluster variant arguably has a simpler structure than the MPI code and is easier to maintain. Over the course of the project, it has become clear that the pure OmpSs@Cluster approach is a good fit for a small number of nodes (up to 16 or 32), but a more scalable approach for large-scale applications is to use hybrid MPI + OmpSs@Cluster. This combines the scalability of MPI with automatic inter-node load balancing from OmpSs@Cluster. An initial implementation of hybrid MPI + OmpSs@Cluster was developed for BSC's Alya application (Section 10.1).

3 STFC Applications

3.1 Nucleus for European Modelling of the Ocean (NEMO)

NEMO is a state-of-the-art modelling framework for research activities and forecasting services in ocean and climate sciences developed by a European consortium. NEMO is a large application with more than 95k lines of Fortran90 code. In this project, we aim to port NEMOLite2D, a simplified vertically averaged version of the dynamical part of NEMO to the EuroEXA platform.

The NEMOLite2D application, also written in Fortran 90, implements a continuity equation for the update of the sea surface height and two vertically integrated momentum equations for the two velocity components, respectively. For simplicity, NEMOLite2D implements the explicit Eulerian forward time-stepping method, except for the bottom friction, which takes a semi-implicit form for the sake of model stability. As done in the original version of NEMO, a constant or Smagorinsky horizontal viscosity coefficient is used for the horizontal viscosity term, the Coriolis force can be set in explicit or implicit form and the advection term is computed with a first-order upwind scheme. The model also includes external forcing from surface wind stress, bottom friction, open-boundary barotropic forcing and a lateral-slip boundary condition is applied along the coastlines. The open boundary condition can be set as a clipped or Flather's radiation condition.

For the EuroEXA project, STFC aims to automatically generate a distributed parallel code capable of targeting FPGAs using the PSyclone source-to-source code-generation tool developed at the Hartree Centre. All presented work is open-source and has been upstreamed into the master branches of the following repositories:

- PSyclone: https://github.com/stfc/PSyclone
- NEMOLite2D: https://github.com/stfc/PSycloneBench
- FortCL: https://github.com/stfc/FortCL
- NEMOLite2D Infrastrutture: https://github.com/stfc/dl_esm_inf

The main achievements presented in this report comared to D2.5 are the ability to generate hybrid FPGA and MPI versions of the NemoLite2D code (previously we could only generate each one independenly), and a significant improvement of the FPGA code as measured on the Xilinx U200 FPGA. The produced code is not yet ready to run on the EuroEXA CRDB or Testbed 2 platforms.

3.1.1 Final Ported Applications Software

To port and optimize NEMOLite2D to the EuroEXA FPGA platform we use the PSyclone code generation tool. The purpose of PSyclone is to provide a separation of concerns between the science code and the parallel and performance-oriented details of the software. As such, it is also responsible for the performance portability to different target architectures and can do so without the need to modify the underlying science code.

Prior to this project, PSyclone was able to parse the NEMOLite2D science description and generate Fortran with OpenMP constructs to provide parallelism. To support the EuroEXA platform we had to extend the capabilities of PSyclone to generate OpenCL (as a language that can be compiled for FGPAs with the Xilinx toolchain), add multi-node distributed memory parallelism to the NEMOLite2D application (using MPI) and finally optimize the code generation to produce an appropriate OpenCL structure for FPGAs. We discuss these three aspects in turn.

Enabling OpenCL generation

NEMOLite2D is written in Fortran (with embedded DSL meta-data); we keep the host code in Fortran but we use a source-to-source language translation mechanism to trans-pile the kernel subroutines to OpenCL. We chose OpenCL as the target language to produce FPGA binaries for two reasons: it represents a portable programming language that can be useful in multiple architectures (as we have shown by also porting it to GPUs) and it has a mature runtime supported by the FPGA vendor Xilinx.

The host part of the application sets up the OpenCL devices, instantiates the OpenCL kernels and controls the execution of the kernels using the OpenCL API. To interface with the OpenCL runtime we implemented the FortCL library, a fork of the open-source CLFORTRAN module. This library wraps the OpenCL functionality into a Fortran module using the C interoperability ISO C binding intrinsics.

For the kernels part we focused our efforts in implementing a language-independent, internal representation of the PSyclone components called the PSyclone Intermediate Representation (PSyIR). As part of this work, the PSyIR has been extended to support all the features found in the NEMOLite2D kernels and a new back-end infrastructure which uses the Visitor Pattern to traverse the PSyIR kernels and convert them to an equivalent OpenCL representation.

The PSyclone architecture for converting NEMOLite2D can be seen in Figure 3.1.

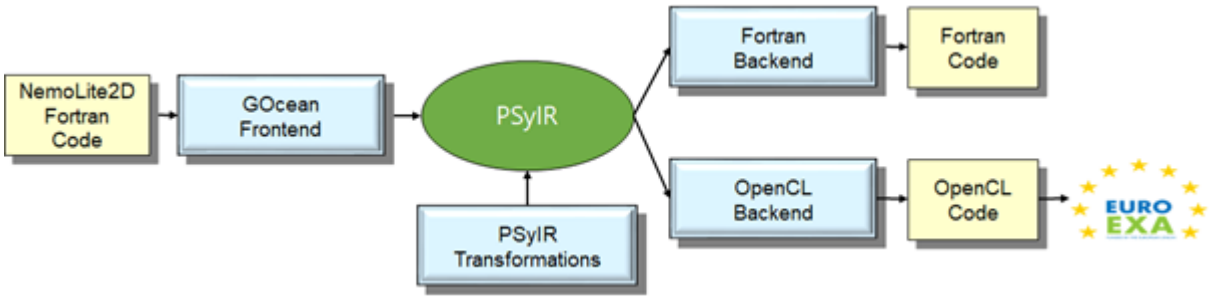


Figure 3.1 The PSyclone architecture for converting NEMOLite2D

This mix of Fortran and OpenCL allows for the generation of a valid, functionally correct, version of NEMOLite2D code that can be executed on CPU, GPU and FPGA platforms. However, the performance of the kernels is unlikely to be portable across different platforms when directly translated. To achieve good performance on the FPGA platform, a set of appropriate PSyIR transformations need to be applied in order to lay out the algorithm in an appropriate style for the

target architecture. This will be discussed in more detail in the “FPGA-specific OpenCL" section below.

However, to verify that the OpenCL translation works as expected, we tested the implementation on GPU accelerated devices. This shows correct results (by checking the checksum of the fields at the final step of the simulation), and, as presented in Figure 3.2, it provides better time-to-solution and energy-to-solution characteristics than the single-node CPU executions on the COKA system (32 cores Intel Xeon Gold 6130 CPU + NVidia V100 GPU with full system power drain monitoring).

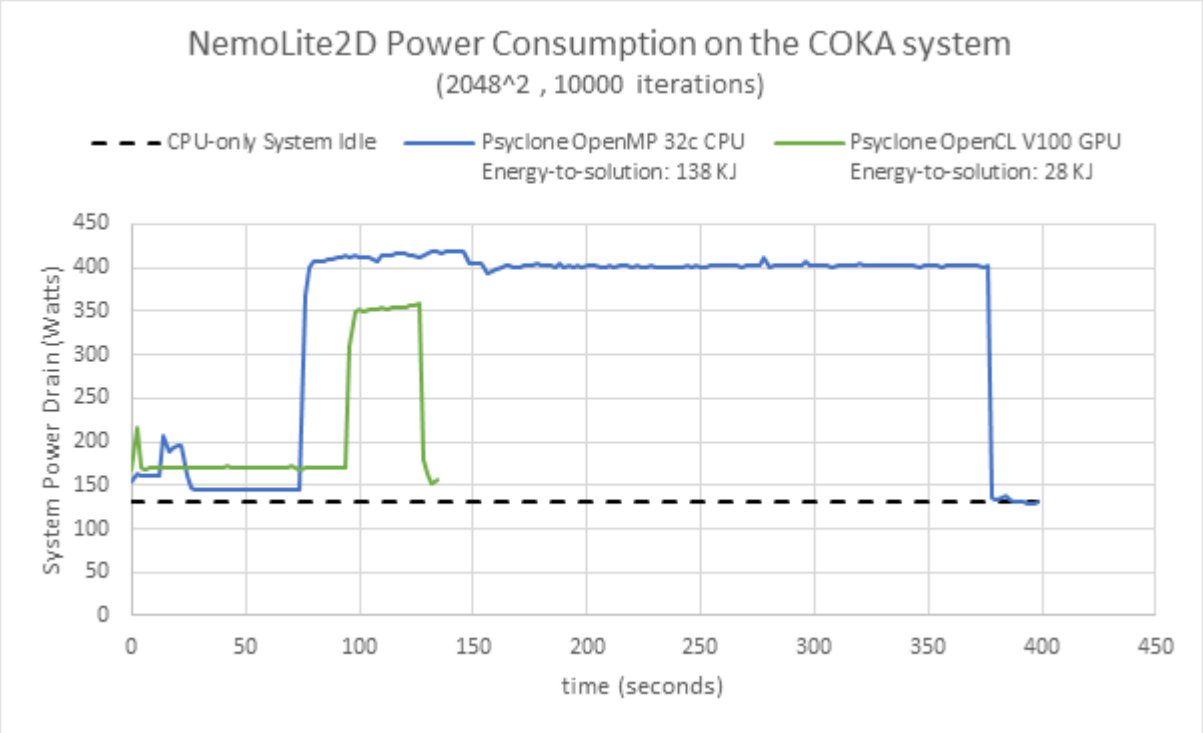


Figure 3.2 NemoLite 2D full system energy consumption and power drain monitoring on COKA

Enabling distributed memory in the NEMOLite2D application

For the distributed memory parallelism in NEMOLite2D we use MPI. PSyclone was already able to provide an MPI domain-distribution implementation for other applications and during the EuroEXA project we extended this implementation to NEMOLite2D. However, additional work was required in order to combine the MPI with the accelerated programming model. This was not a trivial change since the MPI communication happens in a supporting library (called *dl_esm_inf*) that is not aware of the accelerated programming language that PSyclone has chosen. Also, the ‘how’ and ‘when’ data is transferred back from the devices are fundamental to performance and we wanted PSyclone to have control over those.

To generate an appropriate mechanism for data transfer we have extended the NEMOLite2D supporting library, *dl_esm_inf*, to provide callback functions when the data is in a device. This allows PSyclone to take control of the accelerator-language specific instructions and decide the best approach for the communication between the host and device.

Again, we have proven the approach in a GPU environment (since our current FPGA set-up is limited to a single node). The current MPI implementation shows good strong scalability for CPU cores and CPU nodes; it also works on multi-GPU executions, but the scalability is sometimes hindered by the device-to-host communication of the non-contiguous halo regions. We will explore better implementations of these communications as well as direct device communications in the future.

Table 4 PSyclone-generated NEMOLite2D Strong Scalability tested on JUWELS (48c Intel Xeon Platinum 8168 and Nvidia A100)

NEMOLite2D Strong Scalability								
Single-Node Scalability (Size=2048x2048)			**Multi-Node Scalability** (Size=8192x8192)			**Multi-GPU Scalability** (Size=6000x6000)		
Cores	time/it	Speedup	Nodes (ranks)	time/it	Speedup	GPUs (nodes)	time/it	Speed up
1	0.635	1.00	1 (48)	0.29	1.00	1(1)	0.0203	1.00
6	0.112	5.60	2 (96)	0.151	1.92	2 (1)	0.0117	1.73
12	0.056	11.1	4 (192)	0.076	3.79	4 (1)	0.0064	3.15
24	0.034	18.6	8 (384)	0.041	7.01	8 (2)	0.0043	4.74
48	0.021	29.8	16 (768)	0.021	13.91	16 (4)	0.0029	7.07

Optimization of FPGA-specific OpenCL

The default OpenCL implementation using NDRange kernels did not perform well on the FPGA. This is mainly because the FPGA is not managing to load the data from contiguous indexed kernels in single burst reads and writes. To solve this issue we implemented a task-based implementation (with the range loops inside the OpenCL kernels) and manually cached contiguous data in local arrays. In contrast, the replication of Compute Units (CU) and the use of Functional Parallelism (FU) between kernels have both proved to be worthwhile optimizations and can be used in combination with other optimizations.

Figure 3.3 shows the performance of each implemented FPGA optimization that is fully generated by PSyclone (green) or partially generated by PSyclone with some manual modification (blue), and some non-FPGA performance results for comparison. We have plans to extend PSyclone to eventually support all tested code transformations.

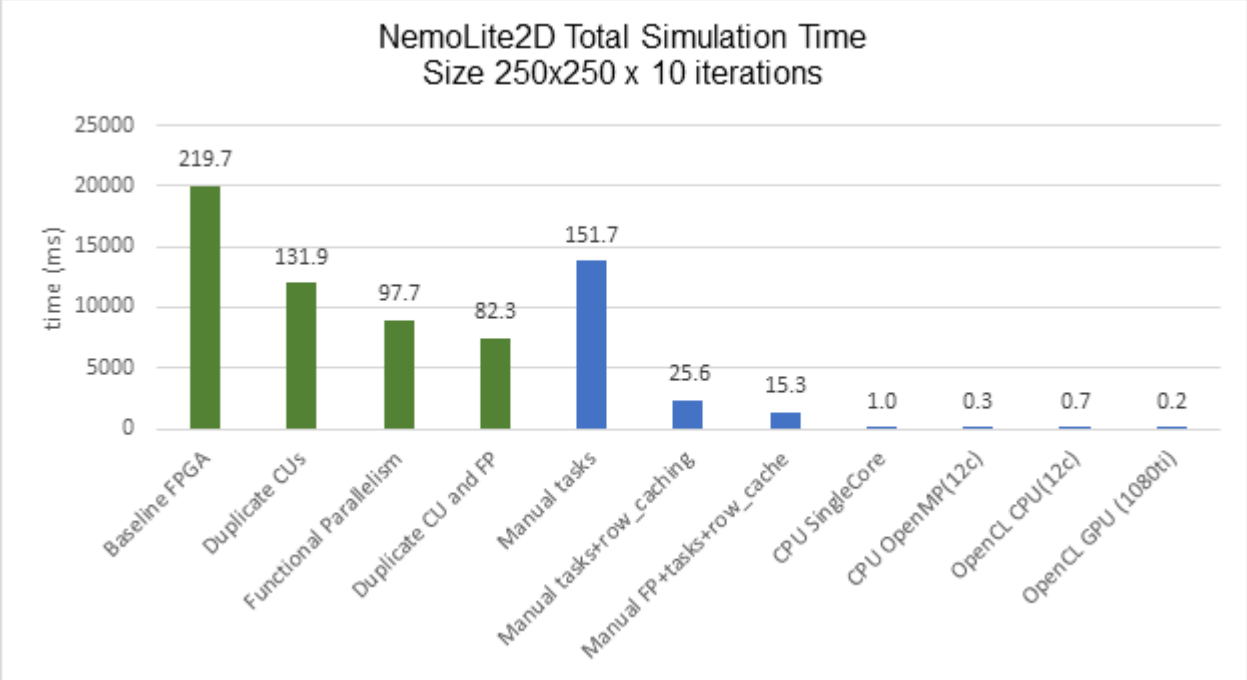


Figure 3.3 NemoLite2D Simulation time of different OpenCL implementations of NemoLite2D on multiple architectures

The best FPGA performance obtained (as in time-to-solution) still lags behind the CPU implementation by a factor of 15; we believe the optimization space hasn't been yet fully explored and we have plans to continue improving the FPGA performance. Perhaps the most significant issue with the current FPGA performance is that the current implementations only use one of the four DDR memory banks available in the Xilinx U200 device. Since we know NEMOLite2D is a memory bandwidth bound application, this has the potential to significantly increase the memory bandwidth of the implementation.

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86 + Xilinx U200 FPGA	Yes
Operating system	Linux	Yes
Languages	Generation: Fortran, Python Execution: Fortran, OpenCL	Yes
Compilers	Fortran compiler, Xilinx OpenCL SDK	Yes
Programming models	OpenCL	
Low-level libraries	MPI	Yes
High-level libraries	-	-

License

3-clause BSD.

3.1.2 Unique EuroEXA Features Supported and Exploited

STFC is leveraging the PSyclone code generation technology to port and optimize NemoLite2D into the EuroEXA platform. The EuroEXA architecture is significantly different from all other systems that PSyclone was targeting before this project and therefore we are implementing a set of unique infrastructure and optimizations to take advantage of the capabilities provided exclusively by the EuroEXA platform. Currently these are exploited using the Xilinx Vivado OpenCL toolchain but the use of an EuroEXA-specific OmpSs@FPGA backend has also been explored. We created a version of the kernel dispatching layer that uses the OmpSs tasks system and it has been tested in a CPU environment with good results. Waiting for a full access to a full CRDB cluster with MPI support, at least a quad-CRDB system is required to optimize the application. Besides the programming model, the EuroEXA platform features presented in the table below are also needed for a successful application scalability:

System Part	Specific Feature	Level of Need	Achieved Implemeted
FPGA	Error Correction Codes in On-Chip memories	Crucial	
Interconnect	Error Detection Codes	Crucial	
Interconnect	Bit Error Rate reporting	No	
Interconnect	Cable hot plug support	No	
Interconnect	Redundant links	No	
Storage	Error Correction of low-level components	Crucial	

Storage	Isolation of defective storage devices	Crucial
Programming Model	Coordinated checkpointing/restart	No

3.1.3 Porting Issues and Lesson Learned

We found that porting applications to the FPGA is a more complex process than porting it to other acceleration devices. We used a portable programming language, OpenCL, which can run the same code on all devices. This is useful to get correct code as it can quickly be verified in other platforms during development. However, when it comes to obtaining optimal performance for the FPGA the code has to be tuned quite differently than for other devices. This is a time-consuming process due to the long compilation times which creates a slow feedback loop between the code modifications to the production of performance results.

Additionally, even though OpenCL is supported by the FPGA vendor Xilinx, we found that the ecosystem is not as mature as other traditional FPGA programming languages. Some compiler options have changed from version to version during the development of the project and some functionality that is available to other programming languages like HLS is not yet supported in OpenCL.

We also learnt the importance of considering the memory interface when programming for FPGAs. How the kernels communicate with the memory or among them is fundamental to achieve the desired performance from the FPGA devices. For instance, we improved the performance significantly by keeping most of the data all the time in the accelerator device and just transfer the contiguous boundaries for a 1D domain decomposed distributed-memory parallelization of the code (this is different from the 2D-decomposition used in the CPU version). Additionally, we had big performance gains by doing the transfer from FPGA DDR memory to the registers using burst reads and writes of contiguous data. The optimization work in this area is not finished, the current FPGA implementation is only using one of the DDR banks, but we can improve the performance by utilizing the combined bandwidth of the 4 memory banks available in the FPGA.

4 EXA applications (ex SYN)

4.1 Neuromarketing using EEG processing

Neuromarketing is the application of neuroscience to marketing. It includes the direct use of brain imaging, scanning or other brain activity measurement technology in an attempt to measure a subject's response to specific products, packaging, advertising or other marketing elements. An approach towards this direction is to make use of fMRI technology which requires the whole procedure to be performed inside an expensive laboratory environment. Furthermore, this approach cannot scale due to the physical and cost limitations that these devices have. EEG is a very promising, cheaper alternative technology to fMRI. Being able to scale and at the same time be used outside labs gives a great potential of getting insights about the preference of the user on certain consumer behavior aspects.

To be more specific about the EEG processing algorithm, we make use of a modified FF (free-froward) neural net which integrates a special type of preprocessing for the EEG signal known as CSP (Common Spatial Pattern). We could apply it on local or distributed data that could be scaled up to Exascale (*i.e.* using EEG big data corpus). The proposed algorithm is based on a deep neural network architecture (end-to-end trainable), spread both in depth and width. The key components of this architecture are the CSP spatial filtering layers, offering strong preprocessing. Independent CSP layers are sparsely connected with respect to the time axis, serving as independent processing pipelines that only connect at a deeper stage of the network.

For the EuroEXA project, EXA aims to port, optimize and evaluate the Neuromarketing code to distributed FPGAs in order to produce a low-power EEG processing algorithm.

4.1.1 Final Ported Application Software

The initial application, before the beginning of the project, had the form of a single kernel, high level prototype implemented in C++. During Task 2.1 (Deliverable 2.1), we moved to a complete synthesizable but unoptimized version using Vivado HLS. Specifically, we have presented the first reference implementation of the EEG core engine in custom hardware to demonstrate its versatility, in order to get the first performance and resource utilization results. In this first implementation, a single unoptimized synthesizable kernel was validated only through Vivado HLS co-simulation **without** using the OmpSs@FPGA programming model.

In addition, during Task 2.2 (Deliverable 2.2), we have extended the functionality of the application to support multiple distributed kernels by developing the functionality of a central node (parameter server) at SW level using OpenMP (on a single device) to validate that the algorithm achieves convergence.

Moreover, during the Task 2.2 (Deliverable 2.5), we have transformed the Neuromarketing code to synthesizable C++ kernel using OmpSs@FPGA programming model, while we have ported

the multi-kernel version on a single device parametrizing the pipeline & unroll stages in kernel code in order to full utilize any FPGA board independent of data size.

In this deliverable, along with the code, we report the following improvements/porting steps over the 3rd year of the project. Regarding the progress report of EXAPSYS Neuromarketing application we have concentrated in the following steps:

1. First of all, we have optimized the HLS kernel with a wide range of Vivado HLS pragmas to execute our application at 300MHz and approximately up to 2 orders of magnitude speedup from the initial HLS version.
2. We have optimized our code separating the channel from the time axis minimizing both the BRAM and LUT utilization, achieving better design utilization and 4 times higher performance.
3. In addition, we have ported and executed a multi-kernel version of our application on a single ZCU 102 board using the latest 2.5.2 version of OmpSs@FPGA.
4. We have ported and optimized the Neuromarketing application on a VU9 board using the Vivado HLS toolchain, while we extrapolated the numbers for the final testbeds, based on the measurements we got from one board and the INFN-provided per-hop latencies, in order to obtain more realistic numbers.
5. Moreover, we developed an MPI-enabled version of our application in S/W in order to efficiently exploit all FPGA boards on the final EuroEXA testbeds, minimizing the data transfers among different nodes during the training session.
6. We optimized the MPI-enabled version of our application so that the coordinator node can call the 1st FPGA accelerator (in addition to coordinating communication) minimizing the total number of MPI processes. Using this scenario, the application can exploit all EuroEXA's CRDBs.
7. We have compiled and executed successfully our application on the CRDB's at STFC and on the quad CRDB at TOPIC.
8. We successfully created and executed one large kernel on the CRDB's at STFC and on the quad CRDB at TOPIC (with 81% of DSPs utilization) @250MHz (the main limiting factor has been proven to be the inter-SLRs links).
9. We evaluated the MPI-enabled multi-node version of our application on the Mont-Blanc ThunderX ARM clusters at BSC as well as the PRACE JUWELS clusters and we profiled them in terms of both communication and computation.
10. Finally, we got application traces from PRACE JUWELS which are used in the extrapolation task.

Ported Application Software

The current application software, ported to the EuroEXA Single CRDB's FPGA at STFC and on the Quad CRDB at TOPIC, can be found in the EuroEXA repository (https://github.com/euroexa/deep-nn-eeg) and consists of the following files, along with a short description of their functionality. It should be noticed that we have implemented the MPI-enable

(multi-node) version of our application but we have not tested it yet in multiple CRDBs due to the delay of testbeds availability (and the porting of runtimes onto those parallel testbeds) but we have developed, tested and evaluated it on the Mont-Blanc ThunderX ARM clusters at BSC as well as PRACE JUWELS clusters.

Files	Functionality
dataset (folder)	the EEG dataset in csv format (minimal training dataset)
dataset_with_dealy (folder):	the EEG dataset in csv format (full training dataset)
csp_tb.cpp	multi core/fpga kernel testbench. Parameter server functionality
csp_tb_mpi.cpp	mpi-enable testbench. Parameter server functionality
MY_NNET.cpp	synthesizable kernel functionality
MY_NNET.fpga.h	header file of synthesizable kernel functionality
define_csp.fpga.h	Definitions about datasets & HLS parametrization
Makefile	OmpSs@FPGA Makefile for EUROEXA CRDB execution

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	X86, ARM, EuroEXA CRDB	yes
Operating system	Linux	yes
Languages	C++, Synthesizable C++	yes
Compilers	GCC	yes

Programming models	OmpSs@FPGA/MPI	yes
Low-level libraries	-	-
High-level libraries	-	-

License

Commercial license. Code will be distributed within the EuroEXA consortium.

4.1.2 Unique EuroEXA Features Supported and Exploited

The Neuromarketing application utilizes and exploits a number of unique features of the EuroEXA platform which heavily facilitated the conversion of the original C code to a fully synthesizable kernel and significantly increased the performance of the application as well.

First of all, the OmpSs@FPGA ***Task Execution Model*** allowed us to manage all the data transfers and task synchronization without being concerned with the device driver to control the data communication and synchronizations. In addition, we were able to implement a multi-kernel version of our application through the OmpSs@FPGA ***Task Execution Model*** which provides dynamic dataflow task-based execution support among tasks executed in the FPGA accelerators.

Moreover, concerning programmability and productivity, we can produce seamlessly different hardware designs in the same platform by configuring the runtime environment variables thus the EuroEXA tools and runtimes enable us to perform extensive design space explorations. In addition, through OmpSs@FPGA **toolchain** we can easily control specific data transfers where necessary in order to perform SMP-to-FPGA and vice versa data transfers using *memcpy* operations.

In addition, **"--interconnect_opt=performance" & "--simplify_interconnection"** OmpSs@FPGA interconnection features have been used in order to allow accelerators to concurrently access different banks, effectively increasing overall available bandwidth.

Finally, the OmpSs@FPGA ***Paraver/Extrae*** tracing toolchain has been used extensively so as to identify the bottlenecks of our application on both ZCU102 and CRDB boards in order to compare the experimental results on CPU-based and FPGA-based systems.

4.1.3 Porting Issues and lesson learned

During the 3rd year of the project, we faced a number of issues regarding the porting of our application onto different FPGA platforms using the OmpSs@FPGA environment (we are working tightly with BSC to address them). Specifically:

1) Trying to move from OmpSs@FPGA v1.4.2 to newest OmpSs@FPGA v2.1.1 we had some issues about the OmpSs@FPGA libraries which did not find the required node in the device tree for zcu102 board (new OmpSs@FPGA version requires Petalinux 2019.1).

- Specifically, we realize that there is an issue about the code in Petalinux 2019.1 .bbappend files (it is ignored). We manage it by creating a script to edit the system-user.dts file and adding the nodes that we need. In our case, we need to copy the contents pl_ompss_at_fpga.dtsi and ZCU102_boot.dtsi into system-user.dtsi

2) OmpSs@FPGA Kernel module is not loaded automatically on OmpSs@FPGA v2.x.x using petalinux 2019.1 on the ZCU102 board.

- After the above issue we need to execute the following command in order to load OmpSs@FPGA Kernel nodule manually after ZCU102 is booted: *sudo insmod /opt/install-arm64/modules/ompss_fpga-4.19.0-xilinx-v2019.1.ko*

3 When we used a large kernel size (>80% DSPs utilization) @300MHz on EuroEXA's CRDB, we got negative slack (due to the inter-SLRs links) and as a result our application was not working correctly.

- In order to resolve the issue, we tried to use 2 (or more) kernels in the same CRDB instead of one large kernel but we could not get speedups because the kernels need to be synchronized and exchange the weights periodically.

- Finally, we successfully create and execute one large kernel size on CRDB (with 81% of DSPs utilization) @250MHz using a custom *memcpy* function (from DRAM to BRAM and vice versa); this has been proved to be the optimal solution.

4) In case two MPI processes are executed in the same CRDB board (one MPI process calls the accelerator through OmpSs@FPGA, the other does not call any OmpSs@FPGA function, it coordinates the intercommunication), the CRDB either crashes or gives wrong results. In the 1st version of MPI-enabled Neuromarketing application, the MPI coordinator node (which is in pure SW) calls the N workers which call the accelerators; as a result, one CRDB is occupied only for the coordinator MPI process utilizing only N-1 of N CRDBs.

- In order to resolve the issue, we optimized the MPI-enabled version of our application so that the coordinator node can call the 1st FPGA accelerator (in addition to coordinating communication) minimizing the total MPI processes. Using this scenario, the application can exploit all EuroEXA CRDB's.

5 INFN Applications

5.1 Distributed simulator of Plastic Spiking Neural Networks (DPSNN)

The Distributed and Plastic Spiking Neural Network (DPSNN) application is an internal development of APE lab at INFN to help benchmarking HW/SW platforms at extreme scales; it is a C++, message-passing parallel code designed to run on clusters that provide an MPI framework. It simulates the spiking dynamics of the cortex by slicing it into a grid of cortical columns populated with neurons and their interconnecting synapses. The evolution of each column (or fraction thereof) is computed by an MPI process while the spikes emerging from synapses of neurons belonging to columns which are far apart become MPI messages between the owning MPI processes.

This mesh can be customized by many parameters such as the number of processes, the column size (number of neurons and synapses per column), axonal delays (how long it takes a generated spike to reach its destination synapse), synaptic connectivity (how many incoming and outgoing synapses belong to each neuron) and resulting topology (how the mesh they make up is shaped) just to name a few; the simulation can therefore be easily turned from a compute-intensive application to a latency-sensitive one to anything in between.

We were unable to complete the porting of the chosen kernels within the application that were synthesized on the FPGA using the Vivado HLS syntax to the OmpSs@FPGA one, so that the only whole application that can actually be run on the CRDBs is the C + MPI standard version; therefore the progress compared with D2.5 was retargeting the DPSNN as a test and debug tool for the networking stack and IPs for the CRDBs, with particular focus on heavy-duty stressing the novel EuroEXA components that were developed and integrated into the intra- and inter-QUADs switches.

5.1.1 Final Ported Application Software

DPSNN compute intensity is not great compared with in-memory data shuffling (for small to average cortical slice sizes) or with remote data exchange among processes (for large cortical slice sizes); for this reason, the porting activity onto the EuroEXA platform has been turned into identifying and excising from the complete application a mini-kernel representing a snapshot of high enough compute activity (one millisecond of purely neuronal dynamics, expunged of the time-consuming management of spike and synapse lists), together with a sample dataset, in order to assess either the incurring complexity and the gains that can be reaped by bringing the DPSNN into the available FPGA programming environment (initially Vivado HLS offered by the Xilinx SDSoC framework on a readily available ZCU development board) without being immediately forced to reckon with available memory size and bandwidth, which are constrained commodities on the FPGA platform.

This compute mini-kernel is made of three parts: a generator of spikes randomly distributed according to a Poisson distribution; a merger that chronologically sorts these spikes together with those incoming from synapses belonging to other cortical columns; an integrator for the Leaky-Integrate-Fire-with-Calcium-Adaptation dynamics of neurons belonging to the computed cortical column. The original merger was based on a recursive implementation of a merge sort; given that the spike sets to be ordered and merged are usually not larger than 20 units and recursive code is to be avoided on the FPGA environment, this section was redesigned to employ different sorter-mergers, namely sorting networks of different sizes – which, being branchless and with a fixed number of operations, are expected to be well-matched with the underlying FPGA architecture – and other trial implementations, to find the one that performs best

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86/Arm + Xilinx FPGA	yes
Operating system	GNU/Linux	yes
Languages	C / C++ (on x86/ARM)	yes
Compilers	GCC, Xilinx Vivado SDSoC SDK	
Programming models	HLS C/C++ on Xilinx SDSoC SDK	
Low-level libraries		
High-level libraries	MPI	yes

License

DPSNN is created by APE Group within the Istituto Nazionale di Fisica Nucleare (INFN) and is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

5.1.2Unique EuroEXA Features supported and exploited

The features of EuroEXA that are more relevant for the DPSNN are those underpinning the platform is built on, namely the promised power efficiency and the compactness of an FPGA-based system able to offer performances in the ballpark of those of a small to medium slice of a standard HPC cluster (from tens of cores to a few thousands which, for sensibly sized neural simulations, seems to be the scalability limit on clusters built upon ordinary computing architectures) at a fraction of the operating costs in power and encumbrance. We are aware that the size of the final testbed that can be conceivably achieved for the EuroEXA platform is not going to be large enough to allow for an incontrovertible and definitive result in this regard. After all, at the moment the porting effort managed to fit just a part of the compute kernel onto the FPGA. Nonetheless, we are hopeful that a reasonable extrapolation will be anyhow possible and a meaningful guess at the answer can be glimpsed once the testbed is available.

5.1.3Porting Issues and lesson learned

It was found that the heuristics that SDSoC employs to perform the synthesis of a C/C++ source into an FPGA bitstream often need non-trivial restructuring on the code to just be able to successfully complete; correctness of the code the user starts with is not by itself a guarantee for a working, albeit unoptimized, bitstream – this adjustment phase may be a time-consuming effort whose weight and scope is hard to assess beforehand and must be undertaken with care, especially in sight of the following stage of the porting work. This latter consists in annotating the source with the HLS directives (as *#pragma's*) that instruct the SDSoC tool to prefer one strategy or another when translating said source into an IP on the FPGA. For example, when pipelining some function – in order to let it be looped over with the highest throughput – significant, often non-trivial reshuffling in source code was found to be necessary to help the *#pragma* annotations in pushing the latency between the end of an iteration and the start of the following one (the initiation interval) to a reasonable value (1 clock cycle being the not always reachable optimum). The experience in this regard was found to be greatly improved by the GUI tools in the SDSoC environment that depict the dependencies between instructions as graphs that, when fiddling around the code, are an invaluable guide for the user in pushing these initiation intervals down so that a better pipelining and ultimately a better throughput can be achieved for the final IP.

5.2 Fluid Dynamics using Lattice Boltzmann Methods

This fluid dynamics simulation based on the D2Q37 Lattice Boltzmann model has been recently selected as one of the mini-app included in the SPEChpc 2021 benchmark suite. This is a bi-dimensional simulation of stacked fluids with different temperature and density, for which the computation of the collisional operator requires 37 elements per lattice site. The computational characteristics of this model makes the D2Q37 application very suitable to stress both the memory- and compute- subsystems of the hardware architecture on which it is run.

This application, from the computational point of view, applies a complex stencil operation on a lattice of data sites, in an arbitrarily long sequence of time steps. In particular, the stencil application consists of two main function kernels: the first, completely memory-bound, performing sparse memory accesses and named *Propagate,* and another one, named *Collide*, operating on the output of the first one, and executing 13 FLOPs/byte in double precision, making it to be compute-bound on most architectures.

Multi-process implementations commonly adopt a classical borders/halos exchange paradigm, splitting the lattice across different processes. Data exchange needs to be performed just before the *Propagate* kernel.

5.2.1 Summary of previous Results

At the beginning of the project, the D2Q37 application was implemented as a parallel code using MPI + plain C; several other versions were also available to target different hardware accelerators, such as GPUs and Xeon Phis, on which the execution of the two main kernels was offloaded.

In Deliverable 2.2, we reported about two new versions of this application, exploring new programming languages to start targeting FPGAs as accelerators, and also a mini-app based on GPI-2 reproducing the data movements of our application. In particular we developed:

1) an SDSoC implementation able to offload just the “propagate” function to the FPGA embedded in the MPSoC of a Trenz board, while running the main program and the “collide” function on a single Arm core;

2) an OmpSs implementation able to target x86 and Arm processors, running the “propagate” on all the available cores;

3) and also a GPI-2 mini-app which reproduces the communication patterns of the LBM application using the GPI2 library.

In Deliverable 2.5 we reported about the first code version that, using OmpSs@FPGA was able to run on the MPSoC Arm cores, while offloading both *Propagate* and *Collide* operations on the embedded FPGA.

5.2.2 Towards the final ported version

After the release of the Deliverable 2.5, a new code version has been implemented, using the Xilinx Vitis workflow, allowing to initially experiment with Alveo FPGA accelerators, while waiting for the EuroEXA CRDB hardware to be available to WP2 partners. Xilinx Alveo boards, in fact, embed FPGAs of the same UltraScale+ hardware family as the EuroEXA CRDBs, but at the time they were not yet supported by OmpSs@FPGA. This implementation adopting the Vitis workflow, allowed us to continue to work on kernel code optimizations, running on a local Alveo U250 FPGA, which embeds much more hardware resources than the Trenz board, used at the beginning of the project.

When OmpSs@FPGA became available also for Alveo cards, we moved back our Vitis implementation to OmpSs@FPGA (re-implementing just the host side of our code), allowing us to both run on the Alveo boards, as well as to be ready to run on the EuroEXA CRDBs, just changing the synthesis target. In parallel, we also included in the application code the message passing support, alternatively using MPI or GPI-2.

As soon as EuroEXA CRDBs became available we were able to deploy our application on them, as well as on the EuroEXA Quad boards, using MPI.

5.2.3Optimization of the Propagate kernel

In this final version, a major performance improvement has been achieved for the *Propagate* kernel. This kernel does not perform floating-point computations, but it rearranges data elements across the lattice; the performance of this kernel is commonly assessed estimating the bandwidth achieved with respect to the maximum theoretical peak of the hardware architecture. On commodity cache-based processors such as CPUs, as well as on GPUs, the main difficulty in

optimizing this kernel is given by the complex and sparse memory accesses the stencil requires to perform, impacting on the underlying memory sub-system. Optimal implementations on CPUs and GPUs commonly require to rearrange the lattice data structures, to adapt to the architecture cache lines and vector instruction widths. For these reasons the *Propagate* kernel can hardly commonly reach more than the 50% of the maximum theoretical bandwidth, and higher value can be achieved using specific programming optimizations, such as non-temporal instructions, threatening the code portability.

On FPGAs, on the other hand, thanks to their reconfigurability, the main stencil can be implemented in hardware, in order to operate only on data pre-loaded into on-chip memories, where multiple parallel accesses can be performed at the same clock cycle, at sparse memory addresses.

In fact, the *Propagate* kernel has been implemented exploiting a shift-register approach. It reads from the external DDR memory a stream of lattice sites at the full memory bandwidth, reading each site just once, storing them on a buffer allocated in BRAMs/URAMs on-chip memories. The buffer is large enough to contain all of the sites required to apply the stencil operator and as soon as one new sites is added to the buffer, an old one, not needed any more is discarded. This approach allows to exploit the maximum available peak memory bandwidth and we reached indeed almost 90% of the maximum experimental bandwidth, measured with a synthetic benchmark developed using OmpSs@FPGA on the EuroEXA CRDB.

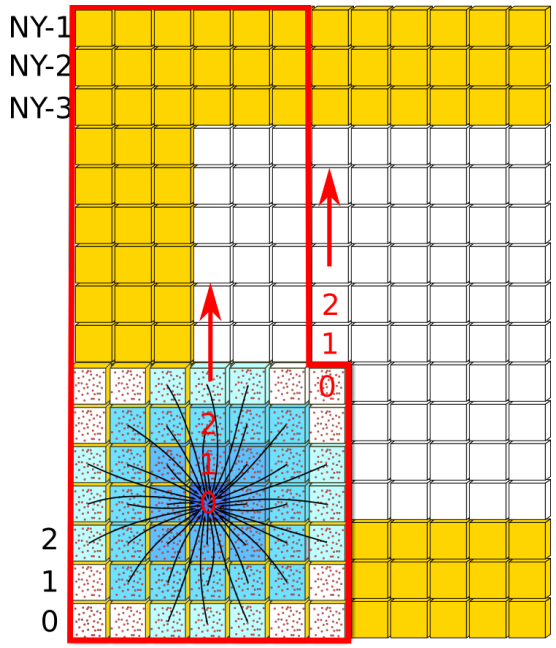


Figure 5.1 Sketch of the Propagate stencil being applied on the lattice sites. The area into the red lines represents the buffer on on-chip memories, highlighting the sites initially stored locally. The Propagate is applied on the site in the middle of the stencil, before applying it on the next one, a new site is loaded into the buffer, and an old one get discarded. Sites marked in yellow represent the halos of the lattice, which get exchanged with nearby processes.

5.2.4 Final ported application software

Our final version of the code is the merge of the latest OmpSs@FPGA implementation and the multi-process one, which allow us to run on the single EuroEXA CRDB (without message-passing libraries support), as well as on the EuroEXA Quad boards, using MPI, and also using GPI as soon as it will be supported.

In summary, in this deliverable, along with the code, we report the following improvements/porting steps over the 3rd year of the project:

- Ported the full application to GPI-2 and tested it on a local x86 cluster, comparing results with a previously existing MPI implementation.
- Ported the full application to the Xilinx Vitis workflow to target Alveo boards. We ran experimental tests on an Alveo U250, allowing us to validate the produced results correctness and also to collect information guiding the kernels optimization work.
- We optimized the *collide* compute-bound function, increasing the amount of lattice sites computed in parallel. We also realized that the Kolmogorov complexity of this function, as well as its high floating-point intensity, is not making this part of the code a good candidate to obtain high performance on the FPGA. At the same time it will allow us to study the limits of this architecture and also to understand how this function could be modified to better fit on this architecture.
- We have optimized the *propagate* memory-bound function, using unique features of FPGAs, such as the BRAM and URAM on-chip memories, making possible to implement this stencil kernel using a shift-register and thus allowing it to reach more than 80% of the maximum theoretical bandwidth of the EuroEXA CRDB architecture.
- We have ported the full application to OmpSs@FPGA to target the EuroEXA CRDB, benefiting from the initial kernels optimizations included in the Vitis version tested on the Alveo U250.
- We collected baseline performance metrics on the PRACE JUWELS cluster, for CPU only and GPU accelerated versions of our application, as well as energy related metrics on the INFN Ferrara COKA cluster, to be later used for evaluation purposes. We also collected Extrae traces for extrapolation purposes.

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86/Arm + Xilinx FPGA	yes
Operating system	GNU/Linux	yes
Languages	C	yes
Compilers	gcc, v++, mercurium	yes

Programming models	OmpSs@FPGA, Vitis/HLS	yes
Low-level libraries	--	--
High-level libraries	MPI, GPI-2	yes

License

The software is copyrighted by the University of Ferrara, University of Rome and INFN under a standard 3 clause BSD license.

Portability

The final OmpSs@FPGA implementation, can target the EuroEXA single CRDB board, as well as the EuroEXA Quad; and theoretically any architecture supported by OmpSs@FPGA.

Ported Application software

This is the application repository where all the related software versions have been uploaded: https://github.com/euroexa/fluid-dynamics

In particular, the multi-process enabled OmpSs@FPGA final ported implementation, targeting the EuroEXA architecture is:

https://github.com/euroexa/fluid-dynamics/tree/master/d2q37-MPI-OmpSs

5.2.5 Unique EuroEXA Features supported and exploited

This application utilizes and exploits a number of unique features of the EuroEXA platform.

From the performance point of view, the most important unique feature of FPGA accelerators, exploited specifically by the P*ropagate* kernel function, is given by the presence of on-chip memories, such as BRAM and URAM. These memories allow to implement as a shift register the complex stencil operation which this kernel needs to apply on the whole lattice at each timestep. This grants the possibility to process lattice sites as fast as they can be read from the external DDR memory. On other architectures such as CPUs and GPUs, proper data layouts have to be used to mitigate the sparse memory accesses adverse impact on memory subsystems and anyway, just a fraction of the theoretical off-chip memory bandwidth can be reached. On such devices, on-chip memories are commonly organized as caches with predefined cache lines, which do not allow to efficiently implement the propagate kernel function using a shift register.

Concerning the *collide* kernel function, its high Kolmogorov complexity as well as its high floating-point throughput do not make it a good candidate to reach a high performance on FPGA devices. Despite this, given that the performance on FPGAs can be highly increased lowering the compute precision, a possible further optimization of this kernel could be obtained investigating in the

direction of lowering the precision of its computations, or at least part of them, studying its numerical stability. This could possibly allow optimizations which may not be possible on ordinary processors and accelerators, for example exploiting the use of fixed-point computations.

5.2.6 Porting Issues and lesson learned

The main porting issue we had, concerning the new implementations reported in this deliverable, has been related to the optimization of the *collide* kernel function. This function is in fact heavily compute-bound, requiring approximately 6600 floating-point operations per lattice site, and to obtain a high performance on the VU9 FPGA accelerator in a CRDB board, is required to be able to use most of the available DSPs to implement floating-point cores. This function is made of several stages, combining the values (called populations) stored at each site with a high number (*i.e.* 666) of floating-point constant parameters, requiring several loops performing different reductions. This translates to a high Kolmogorov complexity, which impacts on the routability of the design of this kernel.

A lesson we learnt, optimizing this complex kernel function, is that it is essential to divide it in several compute blocks, to implement them as separated independent functions, able to communicate with one another thanks to data streams implemented as FIFOs in the FPGA. This is the only way we identified as a solution to implement a routable design able to process multiple lattice sites at the same time, thanks to the use of the DATAFLOW HLS pragma, which implements a function level pipeline. Despite this, less than half of the DSPs can be exploited at this time, since the routing resources are still the main performance limit for this design.

Other minor issues we faced were related to the porting from the first implementation using the Vitis workflow to the OmpSs@FPGA implementation:

1. The use of HLS streams requires Xilinx libraries, making the code less portable, requiring the use of *#ifdef* guards, using the OmpSs defined variable: *__HLS_AUTOMATIC_MCXX__* to hide from the Mercurium compiler all of the code parts where C++ objects from Xilinx libraries are used.
2. While in the Vitis workflow host and device parts of the code are commonly separated, in the case of OmpSs@FPGA we had to reorganize the code in a single file containing the host part and the main kernel functions, moving all of the sub-task functions, using HLS streams in a separate include file.
3. Using OmpSs@FPGA users can not yet select in which Xilinx Super Logic Region (SLR) to place the implemented kernels, and which memory banks to use. This issue is being addressed by other partners and should be solved, granting further possible optimizations.

One interesting lesson learned in the porting of the *propagate* function is the possibility to use FPGA BRAM/URAM to implement complex stencils as shift registers, while this could not be easily accomplished using cache memories on other more common architectures.

.

6 UNIMAN applications

6.1 LFRic

LFRic is an effort to increase the computational scalability of the Unified Model, a numerical model of the atmosphere used for both weather and climate applications. It is being developed by the UK Met Office in close collaboration with NERC, STFC and some universities around the UK, including The University of Manchester.

LFRic is a software infrastructure which since its inception prioritized a “separation of concerns”, an approach aiming at performance portability and scalability without sacrificing scientific productivity. On the one hand, the natural science aspects, *e.g.* how the fluid dynamics equations are solved, are directed by GungHo, a new dynamical core for the Unified Model, responsible for numerically solving the differential equations governing fluid motion over the cubed-sphere discretizing mesh. On the other hand, the technical implementation, *e.g.* how data is moved about within a computer's memory, is determined by PSyclone, a code generation and transformation system that optimizes and parallelizes the science code.

Although PSyclone can target a variety of parallel hardware by leveraging MPI, OpenMP and OpenACC, full support for FPGAs remains a work in progress. Our objective is to demonstrate its potential benefits in the race for Exa-scale computing.

6.1.1 Final ported application software

LRic can be run in many configurations representing a range of weather and climate scenarios across a broad range of spatial and temporal scales. The selected scenario is a baroclinic test case (a baroclinic atmosphere is one for which the density depends on both the temperature and the pressure) which has been developed by the UK Met Office as part of their performance evaluation procedure.

Ashworth et al. showed that LFRic’s computational requirements come mainly from basic linear algebra operations. For the baroclinic benchmark, time is spent chiefly in the Helmholtz solver, which is used to compute the pressure. Among the invoked kernels, the two responsible for most of the execution time, which we will name hx and mv, perform double-precision matrix-vector multiplications within an outer loop which runs over the (thirty in our case) vertical levels of each mesh cell.

We are currently finishing an initial port of the target kernels for acceleration using the FPGAs of the EuroEXA architecture with neighbouring cell data interchanges (also halo exchanges) mediated by the host CPU. Work is ongoing to enable direct memory accesses between sibling FPGAs.

PLATFORM REQUIREMENTS	Required	Achieved implemented

Hardware	Any single- or multi-processor supporting the compilers and programming models below	yes
Operating system	Any OS supporting the compilers and programming models below.	yes
Languages	Fortran 2003, C	yes
Compilers	GCC (gcc/gfortran), PSyclone (domain-specific code generator), Vivado HLS, Vivado Design Studio	yes
Programming models	MPI/Fortran, OpenMP/Fortran, Vivado HLS/C	yes
Low-level libraries	MPI, OpenMP, Vivado HLS	yes
High-level libraries	HDF5, NetCDF, XIOS, YAXT, pFUnit	yes

6.1.2 Unique EuroEXA Features supported and exploited

Both the hx and the mv kernels have been previously ported to the ZU9EG MPSoCs of Testbed 0 using industry standard Xilinx tools, namely Vivado HLS and Vivado Design Studio. The performance of LFRic will benefit significantly from the use of the multiple CRDB nodes in Testbed 2 and the faster, higher-density logic VU9P FPGAs they include.

Furthermore, the two ported FPGA kernels and the PSyclone-generated CPU code that invokes them are currently being modified to allow the halo exchange interleaving them to happen entirely on the former. The objective is to use the on-chip UNIMEM infrastructure to directly address sibling FPGA memory, avoiding unnecessary copies to host memory and back.

Halo exchanges interleaved between unmodified kernels will still happen via host-mediated MPI transfers but using the on-chip Ethernet infrastructure.

6.1.3 Porting Issues and Lesson Learned

The rationale behind the possibilities for computational acceleration cannot ignore the flow of data through the various LFRic kernels and algorithms. The success of FPGA acceleration relies on the data fields being able to reside in FPGA memory for long enough periods to offset the cost of the data transfers between the CPUs and the FPGAs. Therefore, the more tasks are ported to the FPGAs, the more unnecessary data transfers are avoided and the better the potential

performance. This is the motivation behind our most recent efforts to use UNIMEM to directly funnel data and, ideally, one would like to port enough kernels to execute an entire LFRic algorithm without host CPU intervention. However, besides the porting effort, the designer is often limited by the available programmable logic, particularly when as much as 50% of the available resources are allocated for network interfaces. In order to cover a larger portion of LFRic's computational code, future work could instead try an asymmetric approach, whereby different FPGAs implement kernels of different nature.

Finally, we note that the programming model used is rather programmer unfriendly, requiring some low-level considerations, including address manipulation, setting and examining kernel start and stop bits and setting the widths of data paths. However, as described earlier, LFRic uses a "separation of concerns" approach to ensure that scientists need not concern themselves with parallel, platform-dependent coding. In the future, in order to fully support FPGA ports using this methodology, PSyclone should be modified to support automatic FPGA kernel generation.

7 FRAUN applications

7.1 FRTM

Seismic depth migration algorithms calculate images of the Earth's subsurface from the measured and pre-processed seismic reflection data. These images deliver important pieces of information to the geoscientists for discovering oil and gas reservoirs. The method of reverse time migration (RTM) realistically simulates the propagation of waves through the subsurface by solving the full wave equation. Thus, RTM allows the exact imaging of structures with strongly contrasting seismic velocities as they occur, *e.g.* for salt bodies.

Porting to the CRDB prototype includes a simplified version of the seismic kernel implemented as hardware (VU9) and a test application executed on the ARM CPU (ZU9). This kernel consists of a 25-point stencil computation followed by a time integration step. The FPGA ported version of the kernel has been validated by comparing the results with reference data computed on the ARM CPU. Primary changes introduced since D2.5 include a streaming architecture in favour of the double buffering design and the use of multiple HLS kernels to account for the VU9 SLR. The number of stencil computations per clock cycle has been increased to fit the theoretical peak bandwidth of the DDR memory modules attached to the VU9. The seismic data, which is stored in the DDR memory, is organised in tiles to separate halo/boundary from inner volume blocks and the FPGA design is complemented by respective data rearrangement buffers. Finally, GPI communication has been included in the application to support the Quad CRDB.

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	CRDB	yes
Operating system	Linux	Yes
Languages	C++ + Vivado HLS directives	Yes
Compilers	Vivado HLS, Vivado, C++ compiler	Yes
Programming models	Vivado HLS	--
Low-level libraries	--	--
High-level libraries	Vivado design template (BSC)	--

License

Proprietary software owned by Fraunhofer ITWM. Authors: Daniel Grünewald, Matthias Balzer, Leo Nesemann, Norman Ettrich

7.1.1 Unique EuroEXA Features supported and exploited

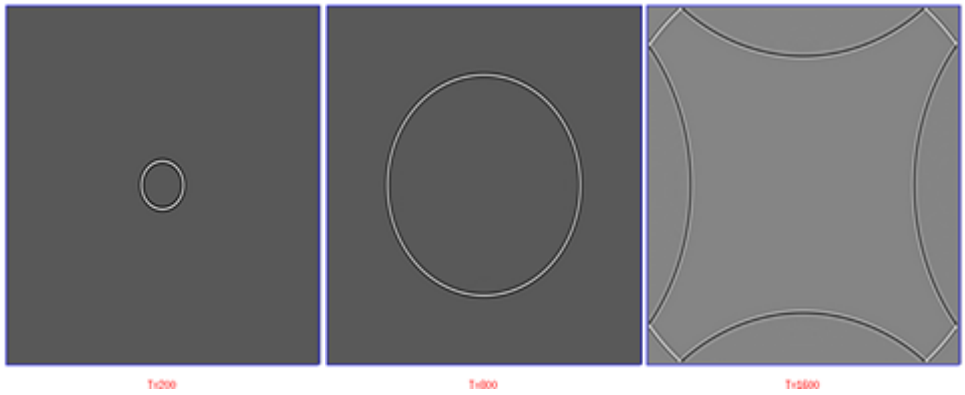
The EuroEXA CRDB combines a ZU9 and VU9 on a single board. Both FPGAs are directly connected via serial links. In addition, 1x16GB and 3x16GB DDR modules are attached to the ZU and VU9, respectively.

The FRTM stencil kernel is implemented on the VU9 accelerator while the controlling application runs on the ZU9 ARM CPU. Test data is loaded to the DDR memory attached to the VU9 before processing. As explained in the following section, a streaming approach is implemented to transfer data between the VU9 and the DDR memory while computing the wave propagation results on the FPGA. With a Linux environment set up on the CRDB, the controlling application can conveniently be executed on the ZU9 ARM CPU. Status and control registers of the kernel hardware implemented on the VU9 accelerator are mapped into virtual address space, transparent to the application.

The GPI communication library enables multi-CRDB support and is used with the FRTM kernel for boundary data exchange. Details on the communication pattern are described at the end of the FRTM section.

7.1.2Porting Issues and lesson learned

FPGA data transfer rates, on-chip memory and compute units are limited by the particular hardware. Within these limitations data compression and reduced precision arithmetic can help to improve performance if applicable to the application. To estimate the feasibility of data compression and reduced precision arithmetic, with respect to the FRTM kernel we ran some tests with the “zfp” and “half” libraries (described below) prior to further porting to the CRDB.



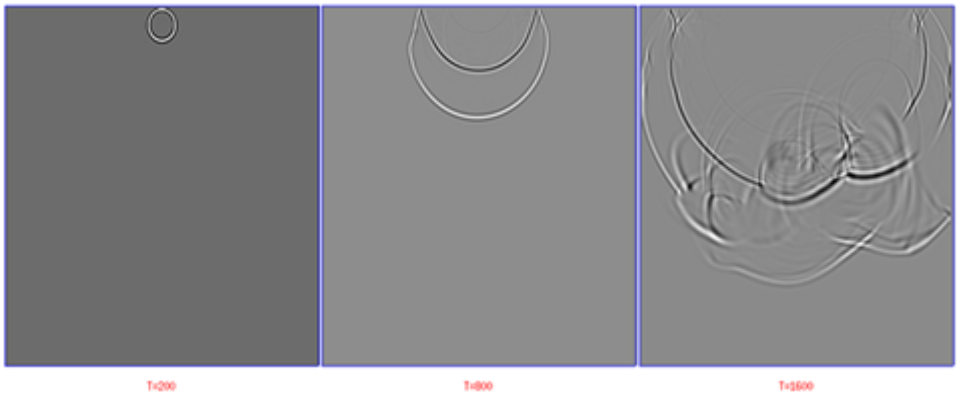


Figure 7.1 Seismic propagation of Ricker wavelet with a realistic velocity model

As test setup for the seismic wave propagation we started with volumes including 500x500x500 grid points and up to 3000 time iteration steps. We considered a constant as well as a realistic velocity model with a representative choice of propagation parameters. Our department's seismic group analysed the effects of compression and reduced precision arithmetic on the propagation results by comparison with reference computations.

Floating-point compression

zfp is a BSD licensed open-source C/C++ library for compressed floating-point arrays (https://computing.llnl.gov/projects/zfp). An FPGA implementation of zfp is available as "zwh". zfp supports lossless and lossy compression and allows for fixed and variable compression rates. To keep our existing memory layout of the volume (grid) data, we focused on the fixed rate compression mode. As the test results suggest, a compression ratio not less than 0.5 seems acceptable. We also tried to compile the hardware version zhw with the Xilinx toolchain, but experienced issues with SystemC support.

Half-precision floating-point

In contrast to *e.g.* fixed-point arithmetic the half-precision floating-point type is conveniently supported by Vivado HLS. For testing half-precision floating-point arithmetic on x86 hardware we used the half C++ library (https://sourceforge.net/projects/half/).

The half-precision format includes 1 sign bit, a 5 bit exponent and a 10 bit significand. The limited range of the exponent requires repeated rescaling of the data during wave propagation to maintain sufficient precision. Rescaling with powers of two leaves the significant unmodified. While the resulting seismic images still reproduce the subsurface structures, also frequency dependent errors are introduced as compared to the reference computations. Though FPGA accelerators are expected to benefit from (custom) reduced-precision arithmetic, further detailed analysis is needed to provide a reliable implementation of the FRTM kernel.

Data streaming architecture

As reported in D2.5 we ported the seismic wave propagation kernel to the ZU9 following a double buffering approach. The approach allowed to overlap computations on the programmable logic with data transfer from/to the processing system and worked well with our test configuration to validate the implementation. For the validation of the design as a first step we only exploited a small part of the FPGA on-chip memory resources to speed up the development cycle. However, increasing the buffer sizes for optimal resource utilization introduced routing/timing issues. Judging from the error messages, the issues mainly originated from unfavorable on-chip memory access pattern.

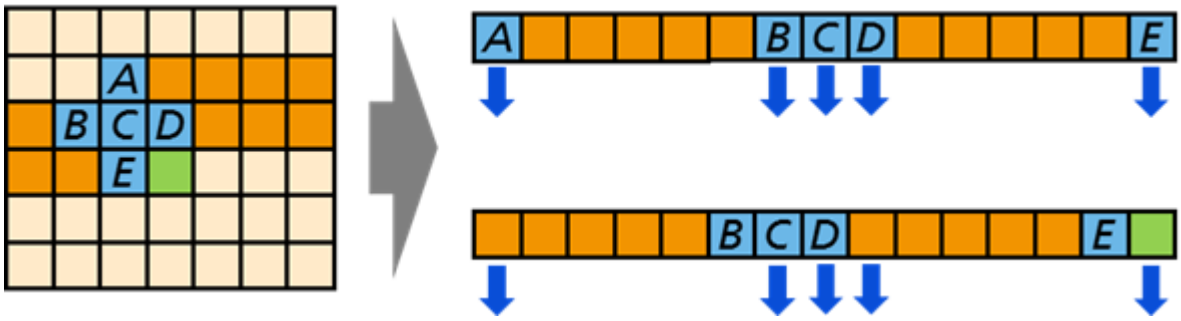


Figure 7.2 Stencil data access for grid volume mapped to shift buffer

We thus replaced the double buffering design in favor of a streaming architecture. This removes the need for random access to the local memory when iterating over the three-dimensional seismic volume. Within the streaming architecture the volume is mapped to a (one-dimensional) shift buffer and the stencil data is read from predefined addresses only (see Figure 7.2). The stencil data for the subsequent grid point is available by adding the next grid value (green) to the buffer and shifting all stored values by one position. For a three-dimensional model with stencil radius *r* the shift buffer needs to store at least *2r* planes. This streaming architecture can be implemented with Vivado HLS by combining FIFOs and shift registers.

The streaming architecture naturally allows for time skewing by duplicating stencil compute units and forwarding stream data from one compute unit to the next. Halo regions of the respective input streams are removed before forwarding the data. Additionally the introduction of time skewing requires source injection to be handled by the hardware module rather than the processing system, since intermediate time steps are not available to the processing system. We use the velocity model input stream to add the source signal amplitudes as well as required meta data (*e.g.* source signal coordinates).

The test application running on the ARM CPU of the processing system has been modified accordingly to support the new hardware design. Validation of the hardware module can still be done using binary comparison with a reference calculation executed on the CPU.

Porting to CRDB

Compared to the ZU9, more hardware resources (LUTs, DSPs, on-chip memory, …) are available with the VU9 as part of the CRDB. However, the VU9 organizes its resources in SLR, which introduces some constraints on how to utilize them. In particular, an HLS IP is limited to a single SLR. Thus, we need to split our HLS kernel implementation to optimally work with the VU9 platform.

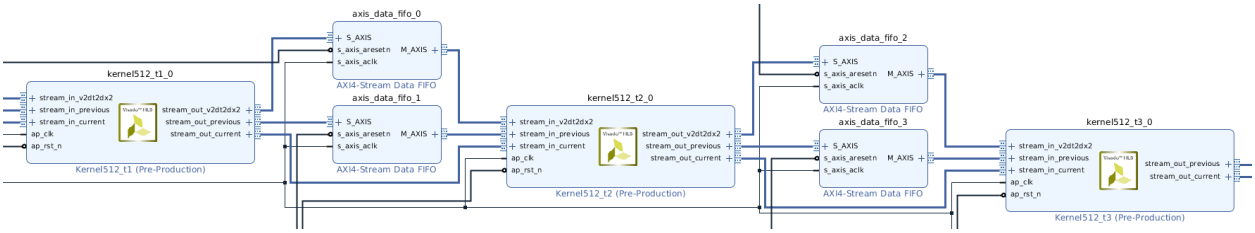


Figure 7.3 Combining three separate HLS IPs to implement stencil kernel on VU9 SLR structure

An obvious choice to split the HLS kernel module is within the time skewing part. This introduces a separate HLS IP for each time step. The internal forwarding of streams from one compute unit to the next is now made external, which also introduces some changes in the kernel interfaces. The split kernel IPs are then connected using AXI Stream Data FIFOs (see figure). The blocking nature of the AXI streams implicitly controls the compute units. Hence, we switched the implementation of the hardware module to a free-running kernel. This simplifies the hardware interfaces, since block control ports of the kernel modules are dropped.

The DDR modules attached to the VU9 support a theoretical bidirectional peak bandwidth of about 36GB/s. With each stencil update requiring 3x4 bytes input data, 15 stencil computations per clock cycle at 100MHz are necessary to fully utilize the available DDR memory bandwidth. Hence, we define the I/O stream width as 512 bits and use separate streams for each of the 3 input data fields. This renders 16 stencil updates per clock cycle possible. To respect the dimensions of the halo regions, we implement the streaming approach with 2D plaquettes of size 4x4.

The stencil kernel input streams, output streams and GPI communication pattern demand different data storage schemes for optimal access. To address the various requirements we organise the seismic data, that is stored in DDR memory, in tiles of different sizes and define halo/boundary and inner partitions accordingly. This allows to transfer data between DDR memory and VU9 as larger contiguous blocks. The resulting data stream, however, is not ready to be used by the compute kernel. Thus, we add rearrange buffers to the design to reorder the stream data as needed. We implement these rearrange buffers with double buffering in Vivado HLS and add the IPs to the Vivado design (see Figure 7.4). The velocity field is not modified during wave propagation and can thus be already stored with duplicated halo regions as needed by the compute kernel. No additional data rearrangement is required here.

The FRTM kernel, in addition to the velocity data, requires wave field data for the previous and current time steps as input. The time skewing approach demands output of both the updated previous and current wave field data to continue the iteration. Using double buffering for the previous and current wave fields, 5 volume buffers have to be maintained during propagation.

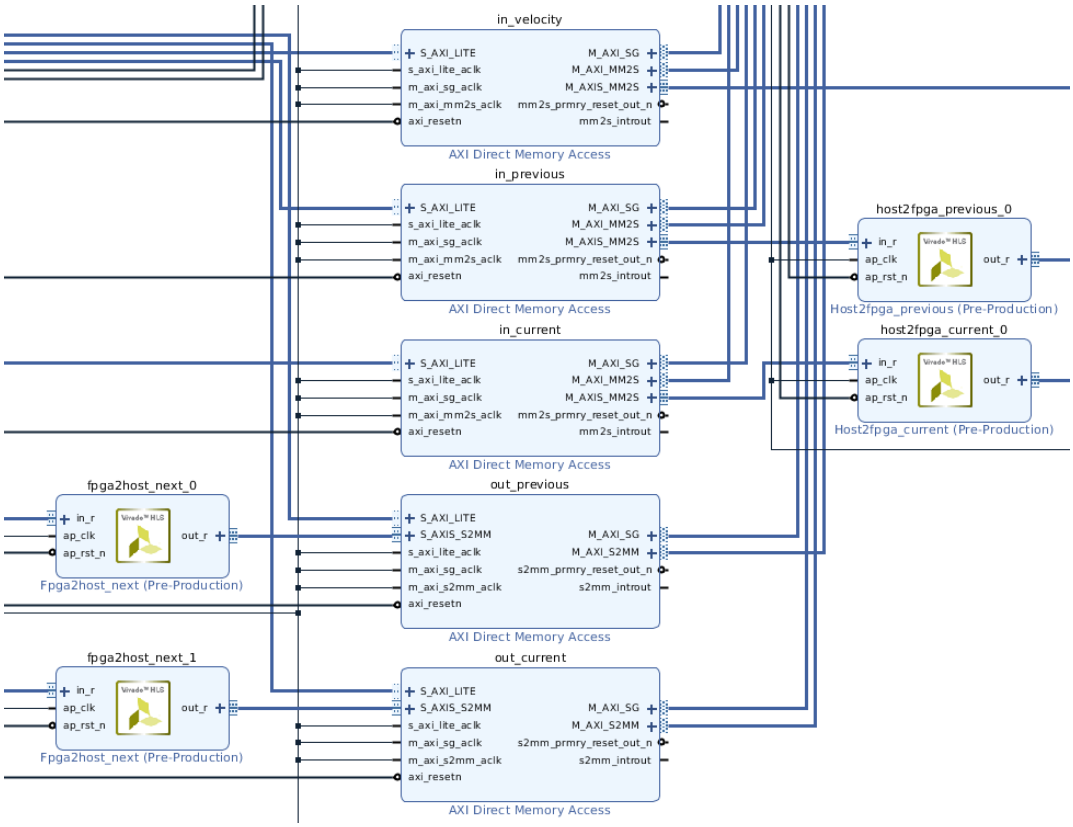


Figure 7.4 DDR-VU9 data transfer with DMA controllers and rearrange buffers.

To more conveniently make the kernel parameters available to the test application we store the particular kernel configuration on the hardware and let the application fetch the parameters to initialize the test scenario. As environment for the CRDB we use the OmpSs setup provided by BSC. This includes the ZU9 firmware with an AXI Chip2Chip bridge to connect to the VU9. The Vivado design template for the VU9 already configures peripheral hardware, e.g. the attached DDR memory.

Keeping the default configuration we add the FRTM kernel IPs to the provided design. The test application, which runs on the processing system of the ZU9, requires access to the DDR memory attached to the VU9 and the registers of the DMA controllers, which handle the data transfer to/from the HLS kernels. With the AXI Chip2Chip bridge this access is transparent to the application. To keep the Linux setup as is, we make the registers and DDR memory available to the application by directly mapping the physical address ranges using the *mmap* utility. On the CRDB root privileges are thus required for the application to run properly.

Multi-CRDB support

The data organisation as described above already defines halo/boundary partitions that can be transferred as contiguous blocks with GPI. Additional packing and unpacking can thus be avoided.

The double buffering introduced for the data transfer with the FPGA naturally supports the GPI inter-FPGA boundary data exchange pattern. As for the compute kernel, data exchange between the CRDBs needs to include not only data for the latest time iteration step, but also that for the previous one. GPI's notification mechanism is used to check for the halo data available when needed for the kernel computation. The communication pattern on the Quad Board is illustrated in Figure 7.5

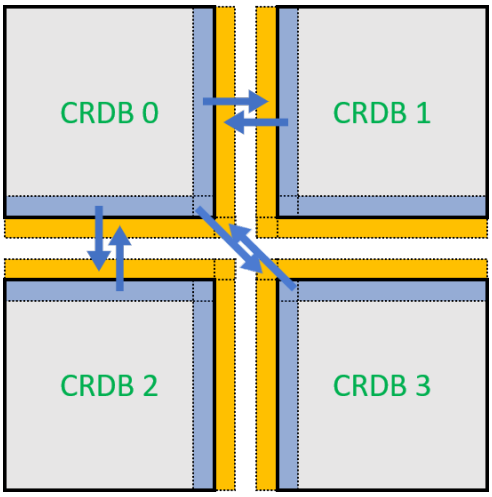


Figure 7.5 Boundary data exchange on multi-CRDB setup

We have checked the GPI communication on our x86 cluster prior to running the application on the Quad CRDB by replacing the FPGA kernel with computation on the CPU. To validate the multi-CRDB setup we run the wave propagation on the Quad Board including all four FPGAs and then compare the results with a reference propagation carried out on a single ARM CPU. As for the single CRDB setup switching off compiler optimisations allows for binary comparison of the computed data.

Since GPI with FPGA support could not be set up on the Quad Board, we use CPU-driven communication with the FRTM application instead. The halo/boundary blocks that we store in the VU9 DDR memory are duplicated to be available for GPI communication on the ZU9. While this workaround does not modify the inter-FPGA communication pattern, it introduces required data copies from/to VU9 to/from ZU9 DDR memory for every GPI data transfer. The additional data management is handled by the application.

Vivado toolchain

When working with the Xilinx toolchain (Vivado HLS, Vivado design suite, Vivado SDK) it seems advisable from our experience to restrict the development to a single version of the tools. While source code compiles with different versions of Vivado HLS, the results of the compilation may still differ in required resources or achieved runtimes. The Vivado design suite on the other hand is pretty sensitive to a specific version. Included Vivado IPs may have changed and introduce incompatibilities within the design. This is in particular crucial when working with design templates from different partners.

8 IMEC applications

8.1 SMURFF framework for Bayesian probabilistic matrix factorization

In the first porting deliverable we explained how we split the SMURFF application into two parts. The first one is the training part of the application. This part has been mapped on the ARM processor of the EuroEXA platform and results of the mapping were presented in a previous deliverable. The second part of SMURFF is the inference part. We have extracted this part in a separate mini-application called Virtual Molecule Screening (VMS). This VMS mini-application has been mapped on the FPGA. Results of the FPGA mapping have been presented in the second intermediate porting deliverable. In this deliverable we present multi-node porting using MPI, hardware verification using an Alveo Datacenter card and OpenCL and also kernel design space optimization.

8.1.1 Final ported application software

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	• Xilinx Zynq UltraScale+ MPSoC ZCU102 • Xilinx Alveo U200 PCIe accelerator card • EuroEXA CRDB single node • Multi-node x86 cluster	Yes Yes Yes Yes
Operating system	Ubuntu Linux 16.04 or later	yes
Languages	C++	yes
Compilers	Mercurium Compiler	yes
Programming models	OmpSs@FPGA	
Low-level libraries	Fixed-point template library (internal)	
High-level libraries	None	

License

The FPGA version of the VMS mini application has not been publicly released and is currently only available to the EuroEXA consortium.

Description of the Ported Application

The application has been ported to do inference using multiple nodes of the EuroEXA platform using MPI. The main MPI primitives are *MPI_Isend/MPI_Irecv*. We also use some collectives

(*MPI_Reduce*). We instantiate multiple instances of the same compute kernel, one instance per accelerator (FPGAs or GPUs) where the output of the accelerated kernel is combined using OmpSs (on a single node) and using MPI across nodes. We have verified the MPI porting on a commodity-hardware cluster with GPUs and CPUs, where each device (FPGA, GPU, CPU) has an optimized implementation of the VMS kernel.

To mitigate the delay of the EuroEXA testbed we have done hardware verification on an Alveo U200 accelerator card. We ported our application to this platform using Vitis OpenCL + Vitis HLS version.

This porting and verification on Alveo has allowed us to do a design space exploration of the optimal kernel parameters that yields best FPGA performance. A single kernel invocation processes one 5D block, and we explored all dimensions of the block. Namely:

- **Number of chemical compounds per block** - Since we process one block per FPGA kernel and since starting a kernel on the FPGA incurs some overhead, we want to make the block size as large as possible. The downsize of a large block size is that we always need to allocate memory for the complete block, even if we do not fill it completely.
- **Number of molecule features** - We use a word embedding to represent each molecule as a vector of 512 elements. This size allows us to accurately represent each molecule we want to generate and evaluate. Although possible, it was not needed to split this vector into smaller chunks for processing on the FPGA.
- **Number of protein targets per block** - Together with the number of samples, the number of protein targets determines the output size and hence the required write-bandwidth to the DRAM on the EuroEXA. We have chosen to pick 128 of the most interesting protein targets. Fixing this number allows us to explore FPGA utilization by varying the number of samples.
- **Number of latent dimensions -** This is the main indication of the model complexity. For our application to get good prediction a number between 30 and 100 is needed. We have opted for 32 giving us a kernel size that fits comfortably on the FPGA.
- **Number of samples of the posterior distribution** – We need around 100 samples in the posterior distribution to obtain good results. We opted to make a kernel that processes only 16 samples and instantiate multiple kernels on the FPGA. We combine the samples from those different kernels on the CPU. This allows us to better utilize the multiple DRAM memories and the multiple SLRs, since we can connect one kernel to one SLR and one DRAM bank.

The results will be presented in the exploitation deliverable (D2.7).

8.1.2 Unique EuroEXA Features supported and exploited

SMURFF heavily relies on the following unique features developed in the project:

- Kernel off-loading to FPGA using OmpSs@FPGA: the VMS application relies on OmpSs@FPGA for FPGA acceleration.

- Task-based programming using OmpSs in general: the main SMURFF application takes advantage of task-based programming for shared-memory parallelization and load-balancing.
- Dataflow programming and data flow graph generation using MaxJ: we also evaluated the Maxeler programming model and found it to be equally potent for efficient kernel FPGA acceleration.
- Fast low-latency asynchronous communication using GASPI: we have a GASPI/GPI implementation of the main SMURFF distributed implementation (called BPMF). This GASPI version outperforms the MPI version.
- Excellent energy vs performance: our first evaluation of the FPGA version shows a performance on par with a GPU implementation at a much smaller energy consumption.

The results will be presented in the exploitation deliverable (D2.7).

8.1.3 Porting Issues and lesson learned

Porting the code on the platform poses some specific challenge:

- The code is written in C++ and we use OmpSs to offload relevant parts to the FPGAs, however OmpSs compiler presents some limitations on the use of C++.
  Solution: we isolated relevant FPGA code.
- FPGA compilation and bitstream assembly is slow, however the design space exploration of code and its performance is mandatory. It is crucial to explore the best options and parameters for loop blocking, loop unrolling, fixed-point refinement, data-layout optimizations, and so on.
  Solution: we automate the space exploration using scripting and code generation in Python.
- The FPGA toolchain (both EuroEXA internal as well as external tools) is not always mature.
  Solution: The toolchain became much more mature during the project.

9 INAF Applications

9.1 GAlaxies with Dark matter and Gas intEracT (GADGET)

As explained in deliverable D2.2, due to the complexity of the GADGET code (*i.e.* the considerable diversity of algorithms and data structures) we focused on several specific kernels that are crucial in well-defined tasks which are the building blocks of the GADGET architecture.

Namely, we have identified the following kernels:

- the direct N-Body force evaluation (referred in the following as kernel 1)
- the domain-decomposition based on space-filling curves (referred in the following as kernel 3)
- the tree building (referred in the following as kernel 3)

Moreover, to enlarge the spectrum of test application, algorithm and computational/communication pattern included in our testing, we considered the possibility of using a mini-app extracted from the Pinocchio code (referred as Kernel 2) that was developed to test the reshuffle of computational domains on a grid (and is then memory- and communication-intensive).

In the final months of the project (since D2.5), we focused mainly on Kernels 1 and 3, which are our main targets in this project. In section 9.1.1, we discuss the improvements respect to the D2.5, in particular the optimization of the CRDB FPGA acceleration and the overall code improvements necessary to adapt the application to the Exa-scale platforms which have the same high-level feature of EuroEXA project. In fact, enhancing the adaptability, the memory-hierarchy awareness while lowering the amount of data travelling over the network are key features the code must be equipped with if we want to really benefit of Exa-scale supercomputers, as the EuroEXA prototype. However, we are aware that the size of the final testbed that can be conceivably achieved for the EuroEXA platform is not going to be large enough to allow for an incontrovertible and definitive result in this regard.

Kernel 1 was the more suited to be ported on FPGA, while Kernel 3 has been used to implement a re-engineering of the code following the general guidelines described in the previous deliverable D2.2 and D2.5 in order to evolve it to exploit the EuroEXA testbed 2 and 3 and future and emerging architecture sharing similar properties.

9.1.1 Final Ported application Software

Kernel 1: N-Body for direct force evaluation.
It relies on a Hermite 6th-order integration scheme, as described in D2.2.

We implemented a direct N-body code in order to exploit FPGA as accelerators.

Currently three versions of the code are available:

1. the version, presented in D2.2, in which the host code is parallelized with hybrid MPI+OpenMP programming, while the computational kernels (one per each stage required by the Hermite solver) are parallelized with OpenCL in order to exploit GPUs;
2. one version written in Standard C, optimized for CPUs, parallelized using MPI+OpenMP;
3. the final code implementation, derived from the version presented in D2.5, fully OmpSs@FPGA compliant, which is able to run on the MPSoC Arm cores, exploiting the FPGA embedded in the EuroEXA CRDB. Moreover, this final ported code allows us to run on the single EuroEXA CRDB (without message-passing libraries support), as well as on the multi-FPGA EuroEXA system, as soon as it will be made available to WP2 partners.

In summary, in this deliverable, we report the following improvements/porting steps, along with the code:

1. we have ported the full application to OmpSs@FPGA to target the EuroEXA CRDB, benefitting from the previous kernel optimizations tested on both Zynq-US+ and Zynq-7000;
2. we have optimized our code using Vivado HLS toolchain, in order to efficiently exploit all FPGA boards on the EuroEXA testbeds;
3. we further optimized the kernel using unique features of EuroEXA FPGAs, such as URAM, along with BRAM, on-chip memories, allowing the implementation of local arrays partition on flip-flop registers. We also rely on DATAFLOW HLS pragma to divide kernel functions in several compute blocks, and so to implement them as separated independent functions, able to perform in parallel different tasks;
4. we designed the final application to target any architecture supported by OmpSs, allowing the programmer to easily fine-tune the FPGA parameters before starting the time-consuming process of the bitstream creation;
5. we collected baseline performance metrics (*e.g.* time-to-solution, GFLOPS) on the PRACE JUWELS cluster, for CPU only and GPU accelerated version of our application;
6. we collected energy related metrics on the INFN Ferrara COKA cluster, to be later used for comparison/evaluation purposes.

We have compiled and executed successfully our application on EuroEXA CRDB@STFC using a BURST_LEN=1024, FACTOR_UNROLL=8, running at 200 MHz, using the following resources on the VU9.

Table 5 Resource Utilization for INAF GADGET on CRDB

	Used	Available	Utilization
BRAM_18K	784	4320	18.15%
DSP48E	3344	6840	48.89%
FF	350848	2364480	14.84%
LUT	348694	1182240	29.49%
URAM	96	960	10.0%

BRAM and URAM occupancy mainly depends upon the number of particles loaded in chunk (*i.e.* BURST_LEN), while DPS and LUT on the unrolling factor used in the loops (*i.e* FACTOR_UNROLL).

Code release through GitHub

This is the application repository where all the related software versions have been uploaded: https://github.com/euroexa/Exa-HiGPUs Each release corresponds to the status at a specific deliverable, thus v0.2.2 contains all the software released for D2.2, v0.2.5 contains the updates associated to the D2.5, while v0.2.6 contains the updated associated to this deliverable D2.6.

Kernel 3: domain-decomposition and tree.

We performed several developments on some fundamental GADGET's kernels during this activity. Namely, we used the mini-app DDT (Domain-Decomposition and Tree), which we extracted from GADGET, to develop, experiment and test some advancements and re-engineering of pillar algorithms and tasks. The goal is to exploit large Exa-scale platforms with a large number of cores and complex memory hierarchy and accelerators (*e.g.* GPUs and FPGAs on the same nodes). The target platform is the TB2 with VU9 accelerators keeping code portability thanks to the use of OmpSs runtime.

While the work discussed in the previous section is directly linked to the specific FPGA feature of the EuroEXA target machine, the activity presented in this section has a more general perspective in the frame of re-engineering and adapting the scientific HPC code to the future Exa-scale platforms which share the same high-level features of the EuroEXA project. In fact, enhancing the adaptability, the memory-hierarchy awareness while lowering the amount of data travelling over the network are key features the code must be equipped with. Moreover, while what the future Exa-scale platform will exactly look like is still to be established, what seems to be assured is that it will present a strong computational heterogeneity; hence, a good design that allows to

"swap" fundamental building blocks and/or algorithms and their implementation is also of fundamental importance.

In this document we report specifically about the following activities:

1. improvements in the domain-decomposition and tree-building strategies;
2. the threadization of the tree construction;
3. a proof-of-concept C++-ization of the fundamental data structures towards a modularization of the entire code;
4. preliminary works on the explicit NUMA-awareness;
5. Improvements in the acceleration of the code.

Improvements in the domain-decomposition and tree-building strategies. The GADGET's domain-decomposition is based on the construction of a top-oct-tree (top-tree in the following) that represents the distribution in space of bunches of particles (while the complete oct-tree is the representation of the same distribution down to the level of each single particle). The leaves of the top-tree are the smallest units of the domain decomposition that is defined as the assignment of sets of those bunches to each MPI process. The relative weight of a "bunch" of particles is a run-time parameter: namely, it is given as a (small) fraction *f* of the average number of particles per MPI process $weight = N_{total}/N_{MPI} * f$. The larger it is, the faster will be the top-tree construction and the coarser will be the attainable domain decomposition; the smaller it is, the better the domain decomposition, at the cost of a larger time spent in the construction.

The particles are sorted by their one-dimensional id along a space-filling curve (SF-curve; namely, a Peano-Hilbert curve), so that each top-leaf, which is by construction a parallelepiped, could be seen as a segment of the SF-curve and the ensemble of top-leaves as subsequent segments along the same SF-curve.

Then, once the top-leaves are individuated following the constraint of *f*, they are grouped in compact clusters (the "sub-domains) with a "split" operation along the SF-curve so that each cluster is approximately equal in both its computational- and memory- loads. Those sub-domains are then assigned to the different MPI processes in order to minimize the work imbalance within the limit of the maximum allowed memory imbalance. The sub-domains assigned to the MPI processes are not necessarily adjacent and in general they are not.

In this process there are 3 sensitive factors: (a) the precision of the particles' distribution inside the leaves of the top-tree, (b) the quality of the split and (c) the assignment function of the domains to the MPI processes.

(a) The top-tree construction
Once each process has built its own top-tree, all the top-trees must be merged to achieve a

global top-tree whose top-leaves will be the units of the upcoming domain decomposition. Any process is unaware of the detailed particles' distribution within each top-leaf in any other process, and if it is the case to split a top-leaf in 8 sub-leaves its overall load is equally subdivided in the 8 new top-leaves. The initial algorithm implemented by GADGET has a major flaw: that assumption leads to an "instability" in the domain decomposition because the top-tree representation does not strictly depend only on the particles' position but also on their distribution among processes. As a result, time is spent in shuffling the particles among processes and exchanging particles between MPI tasks.

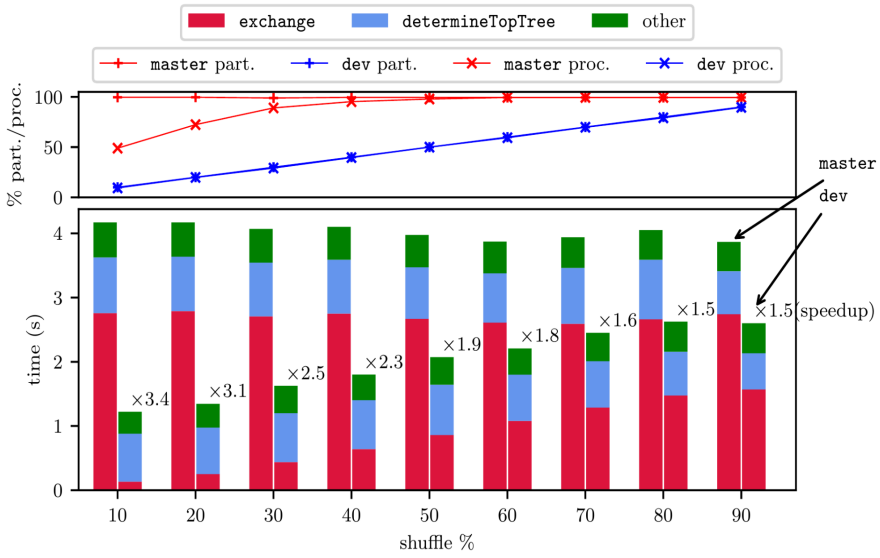


Figure 9.1 The effect of dd in shuffling particles across MPI processes. The x-axis represents the percentage of particles and processes participating the shuffling. The bar plots come in pairs: the *master* branch to the left and the *dev* branch to the right, for each value of shuffle. The speed up of master is also shown on top of dev bars. The graph on top shows the percentage of particles exchanged and processes communications. These instances were run on 4 X 48 = 192 processors.

Figure 9.1 shows this effect (see the caption for details: in the following plots "master" will label the original GADGET's code and "dev" will label our re-engineered code). In the same figure we show the effect of our modifications, which amount in using a larger amount of memory to store exactly the same representation of the top-tree on all the processes. While this algorithm involves a larger communication among processes the actual time spent on it is not larger (blue fraction of bars in Figure 9.1), since the reduction operation between processes is now associative it could be handled in a *log(p)* time (where *p* is the number of MPI processes).

The decomposition is now "stable", *i.e.* it depends only on the particles' position P. This stability determines the fact that a significantly smaller fraction of particles is exchanged due to the change of particles position if that change is small enough with respect to the computational domain. The tree construction is at least as efficient as before in terms of time taken while the number of exchanged particles and the exchange time are very significantly reduced (the speed-up factors are indicated on the top of the dev-related bars in Figure 9.1).

(*b), (c) The split and assign strategy.*
We improved the Split-and-Assign algorithm that individuates the sub-domains along the SF-curve which represents the top-leaves, and that then groups them into domains to be assigned to the MPI processes.

As first, we enhanced the accuracy of the split, which now is guaranteed to be optimal.

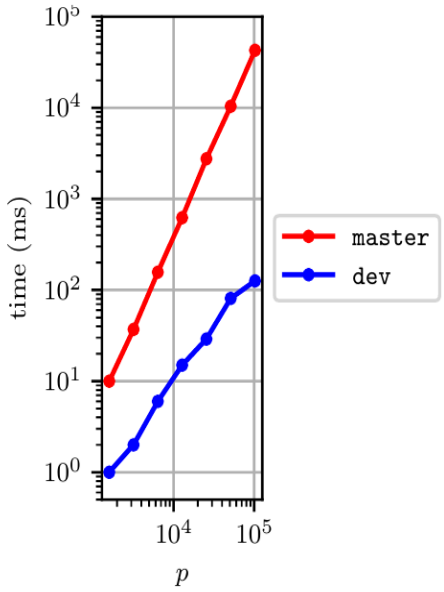


Figure 9.2 Assign routines' time to solution, with multiple-domains $M = 4$. As result of these algorithms pM domains are assigned to p processes.

Secondly, we enhanced both the efficiency and the accuracy of the Assign part, which now is also guaranteed to find a nearly-optimal clustering of the sub-domain.

Taking into account that, obviously, no parallel run could be faster than the slowest of its processes and that the quality of the work-imbalance largely determines the impairing among the processes between two domain decompositions, also these developments may contribute significantly not only to reduce the run-time but also to a better locality in the domain decomposition and hence to a better exploitation of shared memory and a smaller data exchange through the network.

Figure 9.2 shows that our new Assign routine is considerably faster than the original one; although the absolute timings are relatively small (notice that the timescale on y-axis is in ms), the reader should consider that the domain-decomposition, being the responsible of the work-imbalance, should be performed as often as possible and, hence, possibly tens of thousands of times along a simulation.

2 - Threadization of the tree construction;

Having a multi-threaded code is of paramount importance on modern architecture because (i) it adds the mandatory flexibility in the allocation of computational resources and shared memory

resources and (ii) it allows to control the “effective communication surface” among the MPI processes and then the network usage. In the original GADGET code both the tree construction and the tree walking were not threaded (while most of the code is). We underwent the threadization of this utterly important part of the code starting from the top-tree construction.

The changes in the algorithms that have been discussed in the previous section and, consequently, on the related data structures, and the shift from recursive to iterative implementations, allowed a quite simple treatment of the loops that depend linearly on the number of particles.

We show in the following figures the comparison between the OpenMP-ized version of the dev code, labelled as “omp”, and the original code, labelled as “master”. All the test runs were performed on a test-case with 10^8 particles on 8 nodes of a machine having 4 sockets per node and 12 cores per socket.

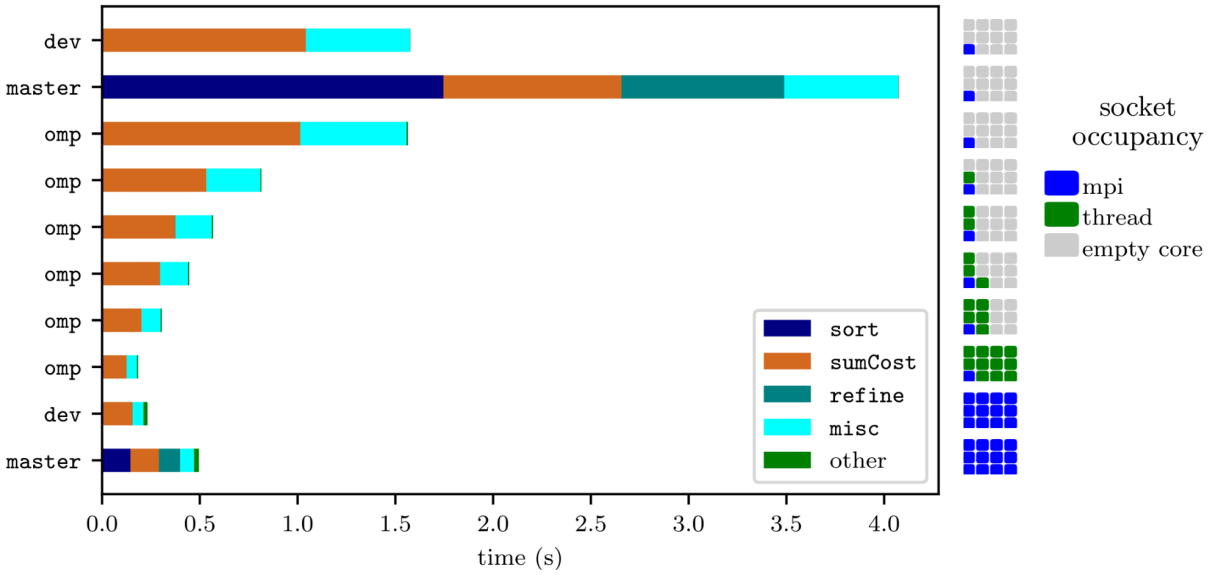


Figure 9.3 Profile of the top-tree constructions for the master, dev, and OMP versions, using 1 or 12 MPI processes per socket and different number of threads. These instances were run on 8 computing nodes using an initial condition file consisting of 2 x 10^8 particles.

Figure 9.3 shows the performance of the threaded version of the top-tree construction. As it is self-evident, (i) our dev version is in general more efficient than the master version both using a single MPI process (first two lines) and 12 MPI processes per node; (ii) the OpenMP threadization is quite efficient and greatly reduces the run-time of all the code sections as the number of threads increases.

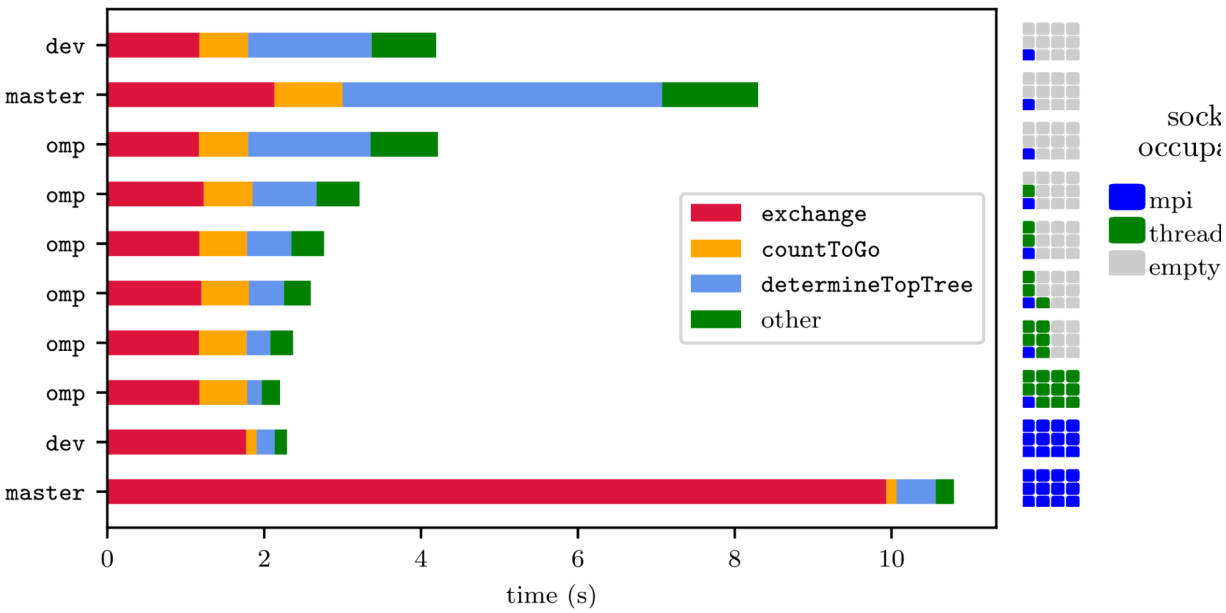


Figure 9.4 Profile of the domain decomposition for the master, dev and omp versions using 1 or 12 MPI processes per socket and different number of threads. These instances were run on 8 computing nodes using an initial condition file consisting of 2 x 10^8 particles

Figure 9.4 shows the effect of the threadization in the top-tree construction on the overall domain-decomposition. Notice that the run-time of the exchange part (the red fraction of the bars) is affected by the re-engineering described in the previous section (less particles being exchanged and a lower communication surface).

The *CountToGo* section is yet to be threaded and as such is not affected.

However, both the top-tree construction (the blue fraction) and other minor operations (green fraction) exhibit a substantial improvement.

Proof-of-concept C++-ization of the fundamental data structures

It is obviously important for a scientific code to be easily modifiable and maintainable and this is where GADGET currently comes short. The details of the tree data structure are deeply intertwined in all parts of the code that use the tree for their calculations. Whenever one needs to introduce a new physical module, it is needed to modify the core parts of the code. This leads to a complicated and barely human-readable “#ifdef forest”. The ideal situation, however, should not require every developer to ever touch core functionality and instead let them concentrate on their particular physics problem, implementing new functions in a separate file while following clear instructions on the APIs.

The main goal of the work discussed in this section is *to develop a proof-of-concept framework to make the modularization straightforward and provide the future developers a convenient set of tools for adding new features to GADGET*.

To accomplish this goal, we re-designed completely new data structures using the advanced features available in the standard C++17; most of the complicated additions needed to realize our version do not come with runtime penalty, as we use templates and compile time loops/conditions thus resolving everything “complicated” at compile time.

Our main change is to shift to a mixed property-based workflow, so that the processes a particle participates in are driven by its “properties”. The main rationales behind this strategy are: (i) to be able to individuate well-defined clusters of properties, *i.e.* data structures, of limited size; for instance, the “kinetic” property that could be listed as mass, id, 3d-position and 3d-velocity, the “hydro” property which is made of density, temperature, hydrodynamic acceleration, …, and so on.

To test the fully functional proof-of-concept, we re-engineered the DDT mini-app, including the domain-decomposition, the tree-construction and the tree-building, also developing an example function which mimics the density loop of the GADGET code.

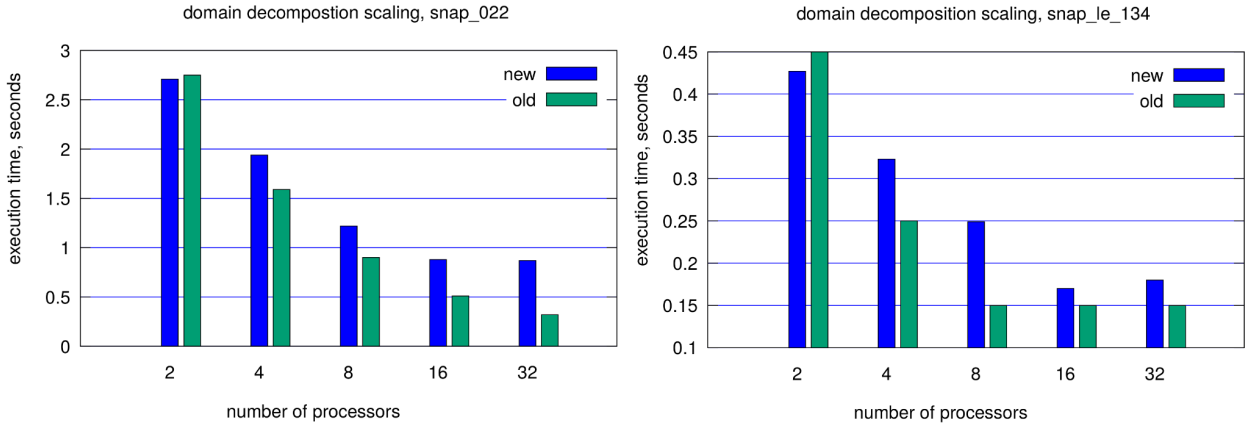


Figure 9.5 C++ versus original DDT implementation.

Figure 9.5 presents the timing of our C++ version of the code compared with the original one. The timings are reported for 2 different test cases, *i.e.* 2 snapshots from real simulations: *snap_le_134* contains 2 million particles whereas *snap_022* contains about 10 million particles. Note: both of the examples represent the evolution of a galaxy cluster at redshift $z \sim 0$ and $z \sim 4.1$ respectively, so that the former refers to a much later time and hence to a much more clustered matter distribution.

That is why *snap_le_134*, even though it contains about 1/5 of particles, requires a larger tree-building time and a comparable look-around time.

4 - Explicit NUMA-awareness

At the moment of writing, we are still developing the best strategy to implement explicit NUMA awareness in the DDT mini-app. We detail our first experience of it in the “Issues and lessons learned” section.

We coupled our NUMA-exploration library, that we already mentioned in previous documents, with the DDT app and as such the app is now aware of the machine topology. The MPI processes are grouped in a hierarchy of communicators as shown in the table here below

hierarchy level	description
WORLD	collects only the master processes of all the hosts
HOST	collects all the processes that run on the same host
ISLAND	*possible intermediate level between NUMA regions and HOST; it may account for an internal connection different than the network among hosts*
REGION	collects all the processes that run on the same NUMA region

Each Level is “collapsed” in the higher one whenever the two levels overlaps: for instance, a common situation is that the “ISLAND” level does not exist - typically, a computational node is also a HOST and has a unique NUMA region - and then the levels “REGION”, “ISLAND” and “HOST” coincide and are a unique one.

At each level at which the memory is shared, the MPI processes create MPI memory windows explicitly shared and non-contiguous.

The rationale that drives our development is (i) to minimize the data exchanged through the network, (ii) to minimize the “effective communication surface”, *i.e.* the number of MPI processes that take part in a data exchange given the same amount of data, (iii) to enhance the exploitation of shared memory resources and (iv) to avoid the synchronization through message passing within the largest possible subsets of MPI processes: for instance, both neighbours search and tree building could be handled more effectively at host-level if the tree structure was held in a shared memory region and accessible to all the processes

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86/Arm + Xilinx FPGA	yes
Operating system	GNU/Linux	yes
Languages	C / C++ (on x86/ARM)	yes
Compilers	GCC, Xilinx Vivado SDSoC SDK	yes
Programming models	C/C++ OmpSx Xilinx SDSoC SDK	yes
Low-level libraries		

High-level libraries	MPI	yes

9.1.2 Unique EuroEXA Features supported and exploited

From the performance point of view, the most important unique feature of FPGA accelerators, which we exploited is the presence of on-chip memories, allowing the implementation of local arrays partition on flip-flop registers. On the other side, we largely based our porting on OmpSs@FPGA runtime that has been developed specifically for our platform.

On the other side optimized communication and memory access is crucial in order to benefit from extreme Exa-scale platforms; this requires excellent network capabilities and optimized MPI implementations. Subject to the availability of an MPI stack for the UNIMEM capabilities of the EuroEXA architecture, this could yield significant improvements to performance of our codes.

9.1.3 Porting Issues and Lesson Learned

During the 3rd year of the project, we faced a number of issues regarding the porting of the application on different FPGA boards supported by OmpSs@FPGA.

- *OmpSs@FPGA v2.1.1*:
    - the main issue was related to the input/output structure data type, on which our code relies, where the application did not work properly despite the bitstream being successfully created Working closely with the development team of OmpSs@FPGA at BSC, we figured out that the issue was related to how HLS was accessing the structure elements, generating a separate port for each struct element. More in detail, it generates accesses to non-valid memory locations as we are generating memory addresses assuming the *sizeof* of the full struct. The solution was to add the extra HLS directive (*#pragma HLS data_pack variable=input_struct_variable*) in the acceleration wrapper managed by OmpSs@FPGA. This issue has been fixed on the latest release of OmpSs@FPGA v2.5.2-euroexa and now the programmer is no longer forced to keep the Mercurium intermediate files and edit them as desired, before starting the time consuming process of creation of the bitstream.
- *CRDB@STFC*:
    - when we use a large kernel size (>80% of DSPs usage) on EuroEXA CRDB FPGA, the application hangs. As suggested by other WP2 partners, a possible solution is to split the large kernel into 2 (or even more) kernels in the same CRDB. Nevertheless, we claim that we could not get any benefit in terms of performance because the kernels use the same memory channels, so the data access is mutually exclusive;

- the biggest obstacle in the porting process is mapping the kernel using >80% of DSPs on VU9, requiring the design to span all three SLRs. This, in turn, requires signals to be passed to all three SLRs and makes it hard for the compiler to perform the place and routing processes during the bitstream creation. The latest OmpSs@FPGA v3.2.1-euroexa allows the programmer to select in which SLR to place the kernel and which memory banks to use. We split the large kernel into 2 or 3 kernels targeting different SLRs but we could not get speedups because the access to DDR banks is mutually exclusive;

- the design space exploration of code and performance in large FPGAs like the VU9 is crucial. Analysis performance with Vivado HLS and scripting can allow to automate the process. However, the most time-consuming FPGA compilation is due to the place and routing processes during the bitstream creation (many core-hours using a x86 server targeting the VU9). To create a valid bitstream, it may be necessary to repeat the process several times. This is due to the fact that during the bitstream creation the programmer may discover that it is necessary to reduce the FPGA resource-usage (e.g. lowering the unrolling factor in loops through Vivado HLS directives) or the clock frequency (so reducing the kernel performance), or in the worst case scenario, to redesign the algorithm targeting the FPGA. This is a very time consuming process;

- we tested the usage of the shared memory using MPI windows using both normal (through `MPI_Win_allocate`) and explicitly shared-memory (through `MPI_Win_allocate_shared`) allocations. Shared-memory has been called having the parameter `alloc_shared_noncontig` set to `true` in order to let the memory to be locally allocated for each MPI process;

- we found - which appears to be undocumented in general, since we did not find any mention of that - that many different MPI implementation (OpenMPI, IntelMPI and SpectrumMPI) proceed via allocating a page of memory in the memory bank of the parent process. See Figure 9.6 in which we print out the memory allocation of 12 MPI tasks running on 3 sockets (4 per socket). All of them allocate 97658 pages; however, tasks 4-7, which run on NUMA-node 1, and tasks 8-11, which run on NUMA-node 2, allocate all but 1 page in their own memory and 1 still in the NUMA-node 0's bank. We think that the mentioned single page is needed to provide a sort of allocation table, like the TLB for the operating system. The result is that the shared-memory needs to be warmed-up before accessing it to be efficient; otherwise, accessing it, either in reading or in writing, results to be extremely slow. Hence, that means that any strategy must take into account that the shared-memory windows must be allocated as resident service area and not as on-the-fly spots to be allocated and de-allocated for single operations. We think that this may be helpful in general and also, eventually, in the adaptation of the MPI library to the specific target machine.

```
Task 0 (proc 73615) :: 0x7f904cf74208 is mapped onto segment 0x7f904cf74000 in numa-node   N0 [97657 pages]
Task 1 (proc 73616) :: 0x7f7ebd745208 is mapped onto segment 0x7f7ea59cc000 in numa-node   N0 [97658 pages]
Task 2 (proc 73617) :: 0x7f09fb169208 is mapped onto segment 0x7f09cb677000 in numa-node   N0 [97658 pages]
Task 3 (proc 73618) :: 0x7feed5349208 is mapped onto segment 0x7fee8dade000 in numa-node   N0 [97658 pages]
Task 4 (proc 73619) :: 0x7fcf1c4d8208 is mapped onto segment 0x7fcebcef4000 in numa-nodes  N0 [1 pages], N1 [97657 pages]
Task 5 (proc 73621) :: 0x7f833d8c5208 is mapped onto segment 0x7f82c6568000 in numa-nodes  N0 [1 pages], N1 [97657 pages]
Task 6 (proc 73625) :: 0x7f38d0820208 is mapped onto segment 0x7f384174a000 in numa-nodes  N0 [1 pages], N1 [97657 pages]
Task 7 (proc 73629) :: 0x7fa1c0e3a208 is mapped onto segment 0x7fa119feb000 in numa-nodes  N0 [1 pages], N1 [97657 pages]
Task 8 (proc 73631) :: 0x7f43c32dc208 is mapped onto segment 0x7f4304714000 in numa-nodes  N0 [1 pages], N2 [97657 pages]
Task 9 (proc 73635) :: 0x7f32519f3208 is mapped onto segment 0x7f317b0b2000 in numa-nodes  N0 [1 pages], N2 [97657 pages]
Task 10 (proc 73638) :: 0x7fae7fe84208 is mapped onto segment 0x7fad917ca000 in numa-nodes : N0 [1 pages], N2 [97657 pages]
Task 11 (proc 73641) :: 0x7f5fdcdab208 is mapped onto segment 0x7f5ed6978000 in numa-nodes : N0 [1 pages], N2 [97657 pages]
```

Figure 9.6 The memory allocation of 12 MPI tasks running on 3 sockets (4 per sockets)

9.2 GaiaGsr

The goal of AVU-GaiaGsr code is to produce a Global Sphere Reconstruction using a subset of the Gaia ESA Satellite observations. We employed a modified version of the PPN-RAMOD model where:

- space-time is represented by the PPN approximation of the Schwarzschild metric of the Sun;
- observations are the Gaia-like abscissae along the satellite scanning direction, computed with respect to the satellite's reference frame.

Unknowns are:

- the astrometric unknowns, represented by the spatial coordinates of the stars along with their proper motions
- the attitude unknowns, given by an appropriate B-spline representation of the Rodrigues parameters of the satellite covering the whole mission duration
- the instrumental parameters;
- the global parameter of the PPN formalism, used to test General Relativity against other alternative theories of gravity.

In general, an astrometric model like the above results in a non-linear equation

$$\cos\psi_i \equiv F_i(\underbrace{\alpha_*, \delta_*, \varpi_*, \mu_{\alpha*}, \mu_{\delta*}}_{Astrometric\ Parameters}, \underbrace{a_1, a_2, ...}_{Attitude}\underbrace{c_1, c_2, ...}_{Instrument}, \underbrace{\gamma, ...}_{Global})$$

To solve such a system, we use a hybrid implementation of PC-LSQR, an iterative method for solving large and sparse linear equations, with the aid of some parallelization techniques and of an ad-hoc compression algorithm of the sparse system matrix A. The LSQR method consists of a conjugate-gradient type algorithm which is equivalent to compute, at each iteration *i*, an approximate solution

$$(1)\quad x^{(i)}=(A^TA)^{-1}A^Tb^{(i-1)}$$

and then evaluates the vector of residuals

$$(2)\quad r^{(i)}=b-Ax^{(i)}$$

which has to be minimized in the least-squares sense, according to suitable convergence conditions defined by the algorithm itself. To solve the system, an additional number of constraints equations are set. The A matrix is represented in Figure 9.7 and is organized as shown.

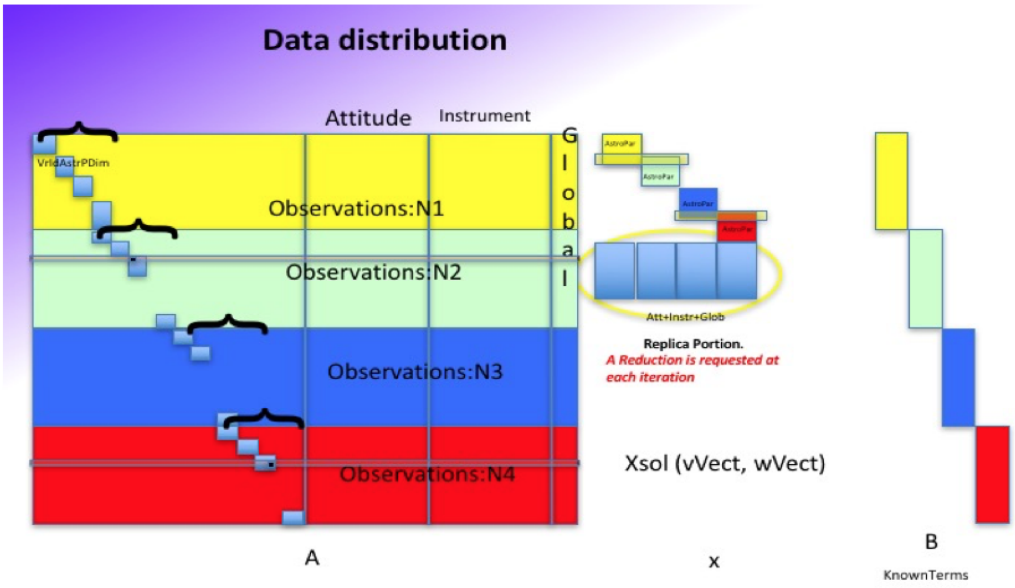


Figure 9.7 Data distribution in the parallel environment: different colors represent the system portion assigned to a single MPI task, with very few replica vector portions.

The code was designed to solve the system equation with 10^8 primary stars observed an average of $7 \cdot 10^2$ times at the end of the 5 years of the mission lifetime. The system *matrix A* is represented as a very large sparse matrix. Figure 9.7 represents the data distribution in the parallel environment: different colors represent the system portion assigned to a single MPI task, with

very few replica vector portions. On each MPI task, each iteration (above described) is executed using the OpenMP paradigm.

The parallel procedure is organized in different modules that are necessary to complete the process: data are provided by DPC (Data Processing Center) pipeline in FITS formats describing the system and must be re-organized in binary format to fill the system as above described. When the main run is complete and a solution is obtained, the final results must be converted into FITS datafile format and organized to be processed on the DPC complete pipeline. Sometimes also intermediate results must be converted into FITS files.

9.2.1 Final ported application Software

The core app has been extracted from the pipeline of the ESA Gaia mission (currently in production) which is a complex application based on a large set of codes and algorithms employing MPI+OpenMP.

In the context of EuroEXA the kernel has been exploited in two different versions: one using a parallel paradigm with MPI and OmpSs and a second one with MPI + OpenACC.

The hardest task to accomplish in the porting of the code was the positioning of the directives managing the data transfers between the host and the device: an inconvenient placing of these directives might make the computation dominated by the data movements, deleting the potential gain in performance. In our porting, we move ~95% of the data at the beginning of the entire cycle of iterations of the LSQR algorithm, making the code compute-bound rather than memory bound. The quantities that we transfer at the beginning of all the iterations are not modified during the computation and they are:

1. The 1D vector containing the compactified coefficient matrix A (the matrix A with the only non-zero elements for each observation) (systemMatrix)
2. The 1D vector of indices pointing to the position that each element of the compactified coefficient matrix had in the original total matrix A for the astrometric and the attitude sections (matrixIndex)
3. The 1D vector of indices pointing to the position that each element of the compactified coefficient matrix had in the original total matrix A for the instrumental section (instrCol)
4. The number of observations assigned to each MPI process (mapNoss)
5. The number of coefficients assigned to each MPI process (mapNcoeff)
6. The 1D vector containing the number of non-zero elements for each constraint equation (instrConstrIlung).

The only quantities that need to be copied from the host to the device and from the device to the host at each iteration of the LSQR algorithm are the 1D vectors of the known terms, b (knownTerms), and of the unknowns, x (vVect), that are updated at each time step. Precisely, the

LSQR algorithm iterates the solution, up to convergence, through these two steps, respectively called “aprod mode 1” and “aprod mode 2”:

1. $b = A \times x$
2. $x = A^T \times b$

that perform the calculations expressed in Eqs. (1) and (2) in Section 9.2.

In the original MPI + OpenMP code, the computation related to modes 1 and 2 was performed in a separate routine, the *aprod* function, called at each iteration of the *lsqr* algorithm. To better handle the data dependencies and to optimize the code, we refactored the application, integrating the *aprod* mode 1 and 2 functions in the *lsqr* function. Specifically, these two new functions computing the *aprod* modes 1 and 2 take as input all the variables necessary for the computation

SMP Acceleration: MPI + OmpSs

The original AVU-GSR code from which we extracted the mini-app is written using a parallel paradigm employing MPI and OpenMP. Each MPI task opens a set of OMP threads so that all the specific threads of a task run on a single node. The code is “thread-based”: each thread has a unique identifier and, depending on the specific input data and the matrix geometry, the algorithm has been designed to optimize the load balancing between threads of each MPI task.

In order to use OmpSs the kernel was rewritten changing the paradigm from “thread-based” approach to “task-based”. We have ported and tested the application on the quad-CRDB at TOPIC exploiting the SMPs available in all nodes. The FPGA exploitation would require a (i) a deep refactoring of the host-accelerator communications and (ii) re-engineering of pillar algorithms of the application to benefit from FPGA acceleration. We start to tackle these items using GPUs, as described in the following.

GPU Acceleration: MPI + OpenACC

The application parallelized with MPI+OpenMP was also ported on GPU accelerators by replacing OpenMP with the OpenACC paradigm. With this porting, the computation of the equations (observations) assigned to each MPI process (coloured stripes in Figure 9.7) are parallelized on the GPU threads rather than on the CPU threads. Specifically, we run the MPI+OpenACC application on a node of CINECA Marconi100, a cluster computer having 980 computing nodes with two sockets per node, 16 physical cores per socket, 4 virtual cores per physical core and a memory of 256 GB. For each node, Marconi100 has 4 NVIDIA Volta V100 GPUs with a memory of 16GB each.

The code ported with MPI+OpenACC runs on multiple GPUs, one per MPI task. On a node of Marconi100, with 4 GPUs, the optimal configuration to run the code is on 4 MPI processes, where the computation related to a single MPI task is assigned to a single GPU device. To compare the performances between the code written in MPI+OpenACC and in MPI+OpenMP we refer to a computation run on 4 MPI tasks.

We show below a schematic version of an extract of the code where these data transfers are highlighted.

```
#pragma acc enter data
copyin(systemMatrix,matrixIndex,instrCol,mapNoss,mapNcoeff,instrConstrIlung)

...

while(convergence condition)

{

...

#pragma acc enter data copyin(vVect,knownTerms)
aprod_mode1(vVect, knownTerms, systemMatrix, matrixIndex, instrCol, instrConstrIlung, mapNoss,
mapNcoeff);
#pragma acc exit data copyout(vVect,knownTerms)

...

#pragma acc enter data copyin(vVect,knownTerms)
aprod_mode2(vVect, knownTerms, systemMatrix, matrixIndex, instrCol, instrConstrIlung, mapNoss,
mapNcoeff);
#pragma acc exit data copyout(vVect,knownTerms)

...

}
```

This porting provides a speedup of ~1.3 per *LSQR* iteration with respect to the code parallelized with OpenMP, but further optimizations are in progress to obtain higher gains. This result can be visualized in the left-hand panel of the Figure 9.8, that shows the computation time of each *LSQR* iteration against the number of MPI tasks per node (bottom horizontal axis) on an entire node of Marconi100 for the code parallelized with MPI+OpenACC (red line) and for the code parallelized with OpenMP (blue line). The top horizontal axis represents the number of OpenMP threads. The middle panel shows the computation of the only mode 1 (speedup of 0.7) and the right panel of the only mode 2 (speedup of 2.5).

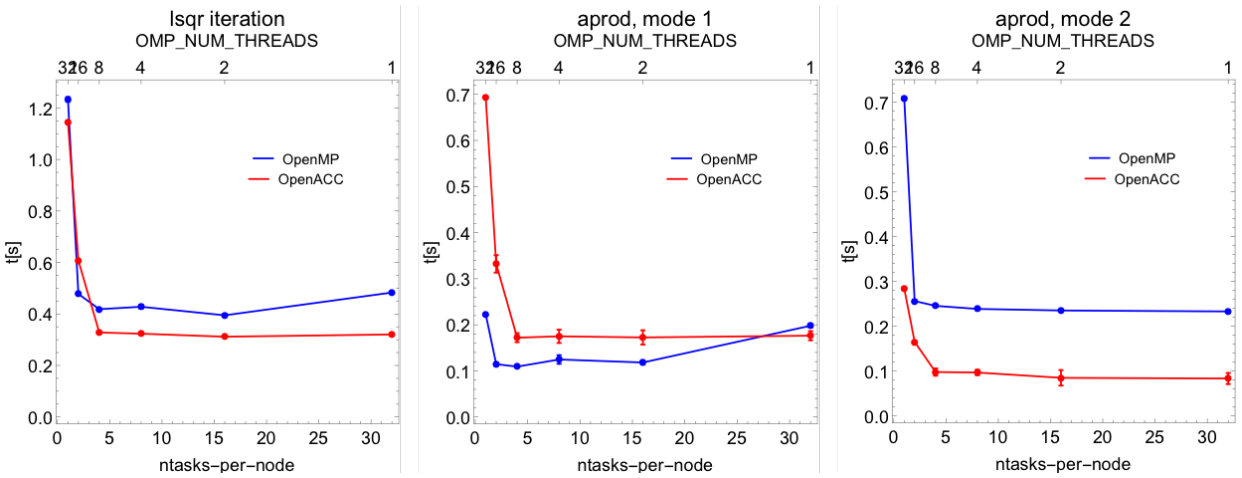


Figure 9.8 GaiaGsr OpenACC porting results at CINECA Marconi 100

9.2.2 Unique EuroEXA Features supported and exploited

The GaiaGsr application aims to utilize and exploit a number of unique features of the EuroEXA platform which heavily facilitated the conversion of the original code to a fully synthesizable one.

In particular we are using OmpSs runtime to offload computations in a SMP manner.

9.2.3 Porting Issues and Lesson Learned

The porting of the code to the FPGA employing OmpSs has required us to change the paradigm from “thread-based” approach to “task-based”, isolating the most computationally intensive portion to run on the accelerator.

The substantial gain in performance resulting from the parallelization over the Accelerators, might be lost due to the time employed in the data transfers between the host and the device. The porting of the code implied a careful study of the strategical points where to transfer data such that they did not dominate the overall computation and a deeper analysis for the insertion of the *pragma* instructions in the different code sections.

9.3 LOFAR Gridding Code

LOFAR (Low-Frequency Array) is an interferometric array of radio telescopes made of small antennas organized in several stations distributed in Europe. It makes observations of the sky in the 10MHz to 240MHz frequency range and exploits the aperture synthesis technique, which consists of a sophisticated data processing methodologies to obtain sky images from the collected signals, delivering the energy flux coming from a given region of the sky, at a given frequency.

Overall, the aperture synthesis methodology is performed in various steps (often repeated in multiple iterations), namely the calibration step, estimating and correcting for time, frequency and direction-dependent instrumental errors, followed by imaging, i.e. the process of converting the corrected visibilities into images through an inverse FFT calculation, then deconvolution corrects the resulting images for the incomplete sampling of the Fourier plane. Finally, denoising and source detection and characterisation are performed.

A full analysis of all these steps of the LOFAR pipeline is beyond the scope of the project. We have hence focused on the most computationally demanding step when data grows to large volumes, as expected for the coming surveys of LOFAR. This is represented by the gridding algorithm of the imaging step, which consists in convolving the visibilities in the complex space to a cartesian mesh that can be then FFT-transformed to get the fluxes. This step, although simple from an algorithmic point of view, is challenging for HPC, due, in particular, to problems posed by efficient memory usage (in terms of distribution of data, locality of memory access, race conditions). Any solution that can optimize its execution has to be carefully explored. Furthermore, the application is data intensive and stresses the I/O phase which has to be properly parallelised and supported by the employ of fast I/O devices and specialised software solutions.

We have developed a code based on the w-stacking method by Offringa et al. (2014, MNRAS, 444, 606) using the C programming language, with some C++ extensions required by the CUDA GPU implementation, adopting a procedural programming approach. Schematically, the resulting code is shown in Figure 9.9, in which we highlight the parts that have been subject to a distributed parallel implementation and those that have been also accelerated through multi/many-threads approaches.

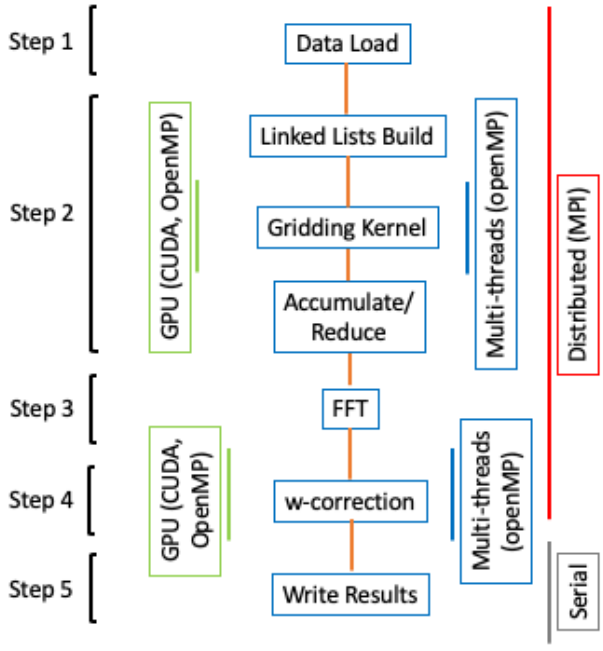


Figure 9.9 Schematic Imagingvcode architecture. Different kind of HPC enabling are highlighted.

9.3.1 Final ported application software

In the project we aim at experimenting the gridding code on different HPC architectures. The initial implementation has been enabled to HPC by using MPI for the distributed parallelism, OpenMP for multithreading and CUDA for the GPU.

The I/O phase, originally based on the adoption of the CASACORE library, has been reimplemented in order to avoid any kind of dependence from external libraries and get rid of the Python driver. Parallel data reading has been implemented adopting basic POSIX solutions. However, further investigation regarding the adoption of MPI-I/O is in progress.

The resulting code has been ported to the ARM architecture and is in the process of being enabled to FPGA adopting the OmpSs programming model.

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86/Arm + Xilinx FPGA	yes
Operating system	Unix-like	yes
Languages	C	yes
Compilers	C++ NVCC OpenMP >= 4.5	yes
Programming models	MPI, OpenMP, CUDA, OmpSs@FPGA	yes
Low-level libraries	MPI, OpenMP, CUDA, OmpSs	yes
High-level libraries	None	

License

The code is open-source, currently unlicensed.

9.3.2 Unique EuroEXA Features supported and exploited

The LOFAR Gridding application can exploit both FPGA (using OmpSs@FPGA) accelerated computing and efficient I/O through the usage of the parallel distributed filesystem deployed on the testbed.

9.3.3 Porting Issues and lesson learned

The porting on the ARM platform (4 CPUs Cortex-A53) has been accomplished using the gcc compiler, version 9.3.0. At the time of these tests the MPI environment was not available, hence

only the OpenMP version could be tested. Compilation and linking were successful with no particular issues. Standard "-O3" optimization has been adopted, together with the specific options *"-march=armv8-a -mtune=cortex-a53 -mcpu=cortex-a53* ".

A number of benchmarks have been performed on the Power9 (P9) architecture available on the Marconi100 supercomputing system at CINECA (Italian national HPC centre - https://www.hpc.cineca.it/hardware/marconi100). The benchmarks measure both the computing capabilities of the ARM architecture and those of the available filesystem. In the following table we present the results obtained using a small dataset, made of around 600000 points in the u-v plane and corresponding 10 million visibilities (the actual measurements), for a total amount of about 100 MB.

In the Table 6 "Input" refers to efficient (and MPI parallel, not exploited in these tests) binary data load, "Processing" refers to the time needed for the gridding and "Output" is the time to write the results in a ASCII file (much slower than the input). Times are all in seconds.

Table 6 Gridding Code performance comparison. We compare the results obtained in Marconi100 supercomputer at CINECA with the ones of the platform. Times are in seconds.

	Input		Processing		Output	
Nthreads	ARM	P9	ARM	P9	ARM	P9
1	0.44	0.02	6.48	1.78	5.40	0.40
2	0.38	0.02	3.38	1.16	5.44	0.40
4	0.43	0.02	1.87	0.82	5.21	0.41

The processing time shows ARM results to be about 4 times slower than Power9 on a single core/thread. Multithreading, however, appears to be more efficient scaling almost linearly with the number of threads. On 4 threads the ARM architecture results to be slightly more than twice slower than the Power9.

The processing time has also been compared to that obtained using the V100 GPU available on the Marconi100 system, which runs in 0.24sec (faster than the CPU, but not so much, due to small dataset size which leads to an inefficient exploitation of the accelerator).

It is clear, however, how the component that has to be highly improved is the I/O subsystem, which leads to a strong slow-down of the application compared to that available for the Power9 processor (GPFS filesystem). This is even more crucial when datasets of realistic size (>, >> GB) are considered.

10 BSC Application

10.1 Alya

Alya is the multi-physics simulation code developed at Barcelona Supercomputing Center (BSC).[1]

10.1.1 Final ported application software

Alya's MicroPP particle code is highly imbalanced due to the mix of linear and non-linear finite elements. BSC ported MicroPP to hybrid MPI + OmpSs-2@Cluster, which combines the MPI programming model with task offloading to address the load balance. The actual translation to MPI + OmpSs-2@Cluster was relatively straightforward, starting from the pure MPI version. The key change was to identify the main kernel using the #pragma oss task annotation, as shown in Figure 10.1. OmpSs-2@Cluster does not have strong support for C++, meaning, for example that task annotations cannot reference C++ members (*e.g.* `material_list`). Also there can be no dependencies among the dependencies to an offloaded task, *e.g.* `gp_ptr->u_k` depends on *`gp_ptr`. In both cases, the fix is to capture the values in a local variable, as shown at the beginning of Figure 10.1.

In addition, some changes were needed related to memory management: (a) use the OmpSs-2 memory allocation primitives, *lmalloc* and *dmalloc*, and (b) pack the *tpu* and *tpvars* arrays into a single allocation to allow them to be described using a single dependency. Finally, the application was modified to not call *MPI_Init* and *MPI_Finalize*, which are in fact called by the runtime, and to use the application communicator provided by the runtime via *nanos6_app_communicator()*, in place of MPI_COMM_WORLD. In total, 630 source code lines were added, modified or removed.

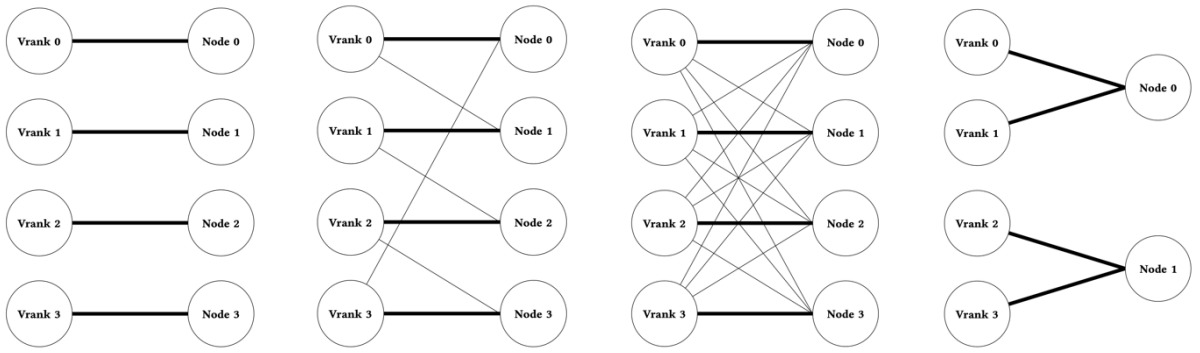
By far the majority of the work in this task since D2.5 was in the Nanos6 runtime system, and the work has been documented in D3.3. In brief, we developed the initial implementation of the programming model and runtime to support hybrid MPI + OmpSs-2@Cluster, which was previously not supported. The hybrid support was integrated with BSC's Dynamic Load Balancing (DLB) library's DROM and LeWI support, in order to manage the sharing of resources within a single node (local vs offloaded tasks from different application MPI ranks). We also implemented a work-stealing load balancing scheduler and two approaches for multi-node load balancing (a local approach that converges to the solution and a global approach using a linear program formulation). Finally, many of the performance improvements reported in D3.3 were in response to performance issues exposed by MicroPP.

```
material_t *tpmaterial0 = material_list[0];
material_t *tpmaterial1 = material_list[1];
material_t *tpmaterial2 = material_list[2];
// [...]
```

[1] Alya – High Performance Computational Mechanics. https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational

```
double *tpu_k = gp_ptr->u_k;
double *tpu_n = gp_ptr->u_n;
double *tpvars_k = gp_ptr->vars_k;
double *tpvars_n = gp_ptr->vars_n;
// [...]


if (gp_ptr->coupling == FE_LINEAR ||
            gp_ptr->coupling == MIX_RULE_CHAMIS) {

             /*
              * Computational cheap calculation
              * stress = ctan_lin * strain
              *
              * All mixture rules are linear in Micropp
              * so the homogenization of the stress tensor
              * is this simple and cheap procedure.
              */

             homogenize_linear(gp_ptr, tpell_cols, ell_cols_size);

       } else if (gp_ptr->coupling == FE_ONE_WAY) {

              #pragma oss task in(this[0])                          \
                               in(tpell_cols[0; tell_cols_size])    \
                               in(tpmaterial0[0])                   \
                               in(tpmaterial1[0])                   \
                               in(tpmaterial2[0])                   \
                               in(tpelem_type[0; tnelem])           \
                               inout(gp_ptr[0])                     \
                               out(tpu_k[0; tnndim])                \
                               in(tpu_n[0; tnndim])                 \
                               out(tpvars_k[0; tnvars])             \
                               in(tpvars_n[0; tnvars])
             homogenize_fe_one_way(gp_ptr, tpell_cols, tell_cols_size);

       } else if (gp_ptr->coupling == FE_FULL) {
             homogenize_fe_full(gp_ptr, tpell_cols, tell_cols_size);

       }
```

Figure 10.1 Identifying the non-linear finite-element kernel as an OmpSs-2 task



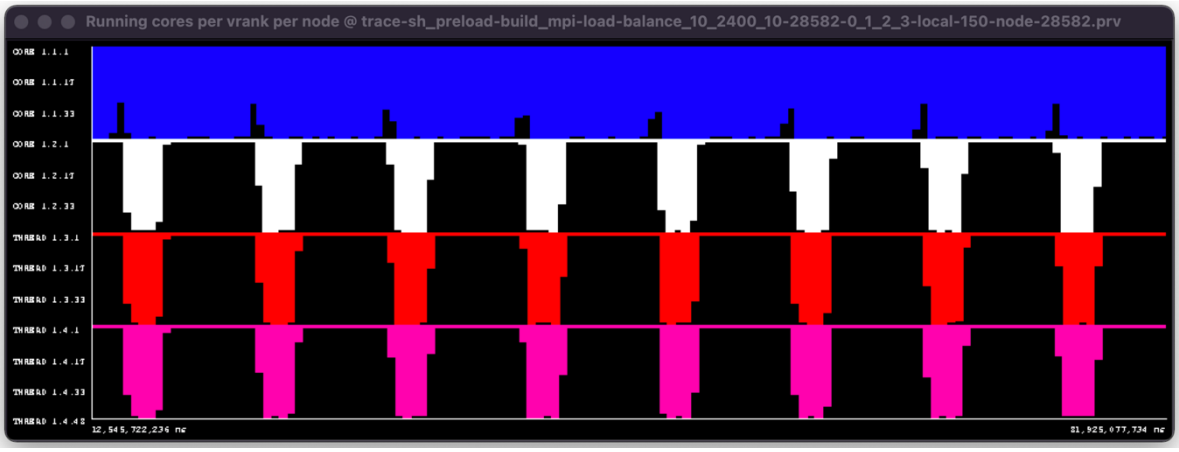
(a) 4 Vranks on 4 nodes without offload (b) 4 V ranks on 4 nodes with offload to next (c) 4 Vranks on 4 nodes with offload to any (d) 4 Vranks on 2 nodes without offload

Figure 10.2: Representations of different executions of hybrid MPI + OmpSs@Cluster

The execution of a hybrid MPI + OmpSs@cluster program can be represented as a graph, as shown by several examples in Figure 10.2. The MPI ranks visible to the application program are shown at the left-hand side of the figures and are labelled by the rank in the application's communicator, which is known as the vrank (virtual rank). Figure 10(a) is an MPI-only execution of 4 vranks on 4 nodes and the bold lines indicate the node on which the main function, and therefore in this case the whole application rank, is executed. Figure 10(b) shows a scenario where OmpSs@Cluster is configured to enable offloading of tasks from each node to the next node sequentially, in case of load balancing issues. Figure 10(c) is the case in which all tasks from every vrank can be executed on any node. Finally Figure 10(d) shows a case where 4 vranks are executed on 2 nodes, without task offloading. Each edge in these graphs corresponds to an MPI process visible to the Operating System and MPI library.

Early in the design, we decided to allow arbitrary graphs such as these, which are determined automatically given the user's selection of the number of vranks, number of nodes and the degree (number of nodes on which the tasks of each vrank can be executed). Limiting the degree controls the overhead in the runtime system of managing processes on multiple nodes that can execute tasks, and it also limits the number of Operating System processes created on each node, which would otherwise grow excessively in proportion to the number of vranks in the whole program.

Figure 10.3 shows a timeline of MicroPP without task offloading, on four nodes, corresponding to the graph in Figure 10.2(a). The timeline is created as a Paraver trace by enabling Extrae instrumentation in the Nanos6.conf configuration file and then running a script to post-process the trace to create the stacked bar-graphs. The four colours correspond to the four vranks in this execution and the nodes are shown in sequence, from the top to the bottom of the window. We see in Figure 10.3(a) that each node executes tasks from the vrank running on that node and that while node 0 is almost fully busy, the other nodes are busy only about one third of the time. Figure 10.3(b) shows the coarse-grained allocation of cores to vranks using DROM, which is fixed throughout the execution.

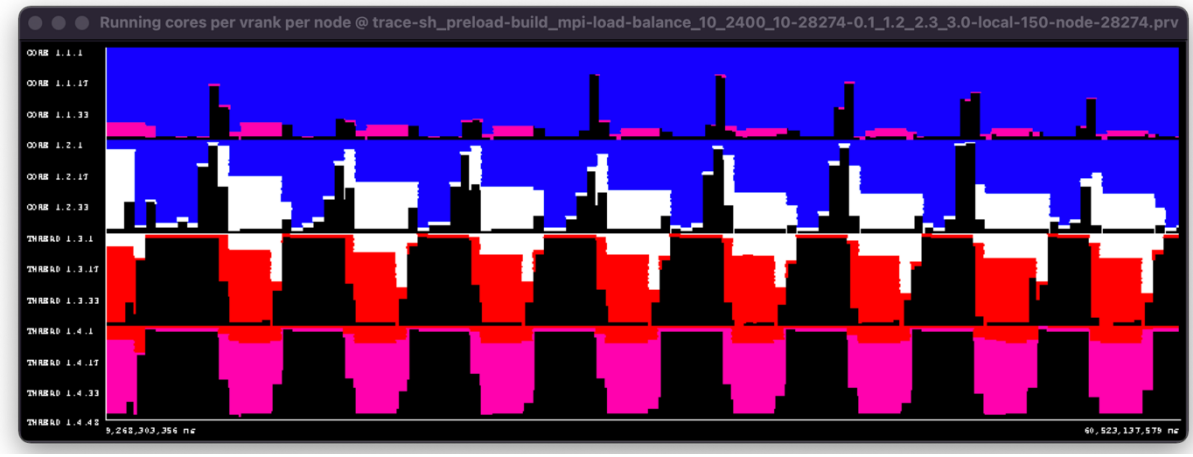


(a) Timeline of number of busy cores per node

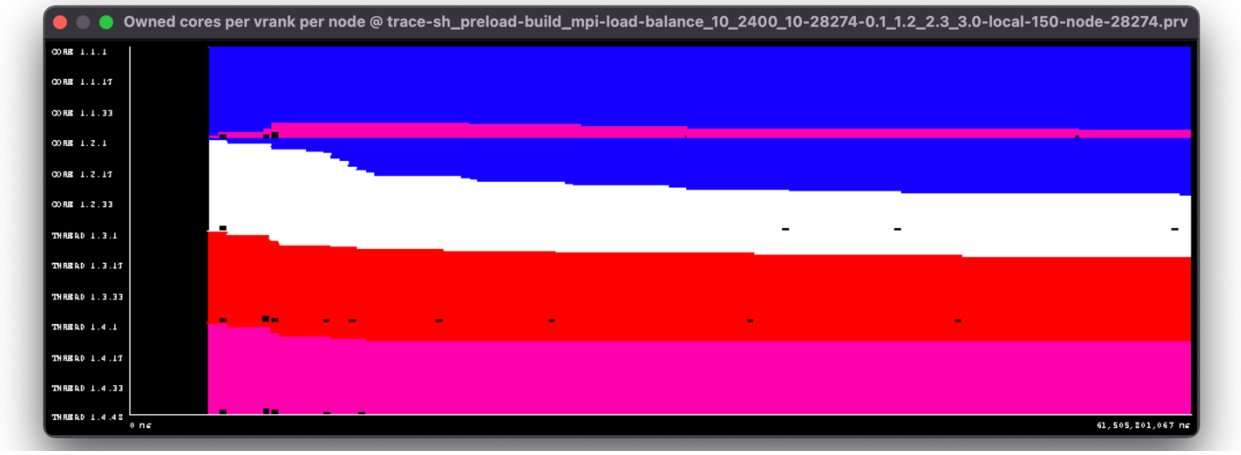
(b) Timeline of number of owned cores per node

Figure 10.3: Timeline of numbers of busy and owned cores per node without task offloading

Figure 10.4 shows a timeline of MicroPP on four nodes, with task offloading enabled to the next node sequentially, which corresponds to the graph in Figure 10.2(b). In this case, the first two nodes are busy most of the time. In addition, much of the work from vrank 1 has been moved from Node 1 to Node 2 to leave more compute resources for vrank 2. Figure 10.4(b) shows that most of the cores on nodes 0 and 1 have been allocated to vrank 0. Note that this plot shows the ownership of cores according to DROM. It is possible for temporarily unused cores to be given to a different rank, using LeWI, but the ownership allows them to be rapidly returned once they are needed again. While the execution is not perfect, the time to completion has been reduced by about 36%.



(a) Timeline of number of busy cores per node



(b) Timeline of number of owned cores per node

Figure 10.4: Timeline of numbers of busy and owned cores per node with task offloading to one more node

Finally, Figure 10.5 shows a summary of the complete set of results. The *y*-axis is the execution time per timestep, in milliseconds. Each group of bars is labelled by the number of vranks and the number of nodes on which they are executed. For example, “4 on 2” means that 4 vranks are executed on 2 nodes. We see that in all cases task offloading improves the execution time, in some cases significantly, for example, as mentioned above, by 36% when running 4 vranks on 4 nodes. The low improvement for 8 vranks on 8 nodes when offloading to one other process is because Vrank 0’s secondary node, Node 1, also has a high load. For this example, it is necessary to enable task offloading to 2 secondary processes, which brings the improvement in execution time to 30%. These are significant figures given the small investment in porting effort needed to enable improved load balancing in the application.

Figure 10.5: MicroPP results showing reduced execution time due to task offloading

PLATFORM REQUIREMENTS	Required	Achieved implemented
Hardware	x86/ARM cluster, e.g. MareNostrum 4	yes
Operating system	Linux	yes
Languages	Fortran 90, some CUDA and OpenACC	Yes/no
Compilers	-	
Programming models	MPI, OpenMP and OmpSs	yes
Low-level libraries	Only MPI	yes
High-level libraries	METIS	

License

Alya can be licensed free of cost under a collaboration agreement. The MicroPP implementation is available open-source and freely downloadable (https://github.com/Ergus/micropp) and usable under GPL-3 license.

10.1.2 Unique EuroEXA Features supported and exploited

MicroPP and OmpSs-2@Cluster are compatible with Arm CPUs as used by EuroEXA TB2. Task offloading requires low-latency communication of small MPI messages (up to hundreds of bytes). Subject to the availability of an MPI stack for the UNIMEM capabilities of the EuroEXA architecture, this could yield significant improvements to performance by improving the ability to load balance.

10.1.3 Porting Issues and lesson learned

The main porting issues encountered were due to the initially low performance of the OmpSs-2@Cluster runtime, which was optimized concurrently with the application porting. As discussed in D3.3, several performance and correctness issues were encountered and corrected in the runtime. Together more than 10x performance was made, with only minor changes in the application code. This work was shared with the porting of InfOli to OmpSs-2@Cluster. Optimization of the runtime and fixing bugs took much longer than anticipated. Diagnosing the issues required extensive performance debugging, which was done with Extrae/Paraver tracing.

Most of the changes, apart from the task annotation in Figure 10.1 and similar annotations on tasks related on initialization, are related to memory management, since OmpSs-2@Cluster requires the application to use a custom allocator, and memory allocated by tasks cannot be used by sibling tasks or the parent. We are working on ways to relax this requirement.

11 ECMWF applications

The IFS, or Integrated Forecast System, is the global weather forecasting and climate research atmospheric model co-developed by ECMWF and Meteo-France and used operationally at both ECMWF and Meteo-France. Results from ECMWF operations are disseminated to Member State weather services. It has been in continuous development and operational use since the 1980s, and has been run on various hardware architectures, such as shared and distributed memory machines, vector and scalar multi-core processors.

The IFS is a so-called spectral model, as its dynamical core is based on the spectral transform method. One part of the computations is performed in spherical-harmonics space (dynamics), while the rest are performed in grid-point space (advection, physical processes), and spectral transforms are used to move from one space to the other at every model time step. The advection scheme is semi-Lagrangian, while time advancement is semi-implicit. The combination of these algorithmic choices allows very long time steps to be used, leading to efficient medium-range forecast computations.

For the EuroEXA project, ECMWF demonstrates a key component of an atmospheric forecast, the cloud microphysics kernel, running on an FPGA. This kernel is representative of kernels in the physical parametrisations that constitute a significant fraction of the overall cost of a forecast. The IFS as a whole is run on the ARM host CPUs.

11.1 ESCAPE dwarves - CLOUDSC

11.1.1 Final ported application software

The porting of meaningful amounts of the IFS physics code to FPGAs is a daunting task. The code language, style and scope mean that hand porting is unrealistic. For this reason, infrastructure targeted at allowing the automatic porting of physics kernels has been worked on.

Currently, the source-to-source translation toolchain developed in-house includes different Fortran frontends that can parse source code into a high-level internal representation allowing composition of tailored code modification passes. From that, backends of varying degrees of maturity allow to produce Fortran, C, Python or Maxeler Java (MaxJ) source code. Since deliverable D2.5, significant progress has been made in developing data flow analysis and loop transformation passes including variable promotion and demotion that serve as building blocks in the transformation process of rewriting a physical parameterization. The current state is that this allows to automatically generate MaxJ source code, including necessary C and Fortran wrappers to call them from Fortran host code, capable of running in the simulation environment for algorithmic patterns commonly found in physical parameterizations.

A crucial step on the road towards batch processing of physics kernels is the development of recipes that describe transformations required to translate the legacy code to the target programming model and achieve good performance. CLOUDSC, the operational cloud

microphysics parameterisation was chosen as a representative kernel for this task. Using the source-to-source translation toolchain, this was translated from Fortran 90 to C, yielding a version that produces bit-identical results. This version of the dwarf was then ported by hand to MaxJ. Additionally, an MPI-parallel version was developed that can work on independent streams of data in parallel, thus allowing the use of multiple devices to speed-up processing of larger data sets and emulating the use of the parameterisation in the full forecast model.

PLATFORM REQUIREMENTS	Required	Achieved/Implemented
Hardware	Max5 DFE (DataFlow Engine) ALVEO U200	yes
Operating system	-	
Languages	MaxJ	yes
Compilers	MaxCompiler	yes
Programming models	-	
Low-level libraries	-	
High-level libraries	-	

License

Source code for ESCAPE dwarves is governed by a specific ESCAPE license. CLOUDSC has been made available open source under Apache 2.0 in January 2022. Source code is available from https://github.com/ecmwf-ifs/dwarf-p-cloudsc .

11.1.2 Unique EuroEXA Features supported and exploited

The ESCAPE software has been ported on the platforms using the Maxeler MaxJ.

11.1.3 Porting Issues and Lesson Learned

One of the most important pieces in the porting process was the design of the algorithmic iteration space, as described in detail in the previous deliverable D2.5. Conceptually, this corresponds to defining a single loop that processes a set of input data streams and generates a stream of outputs which are modified exclusively in the current iteration of the loop. In practice this meant that loops had to be fused and reordered, impacting the scope of local variables and subsequently incurring the necessity to promote certain arrays. Doing so is a tedious task, in particular for long kernels with many local variables and loop-carried dependencies such as CLOUDSC, and can end in an error-prone trial-and-error style approach. The realisation that automatic dataflow analysis tools to identify cuts across variable scopes and dependencies might be able to aid this process significantly sparked the development of such analysis capabilities in the source-to-source translation toolchain.

Another pattern commonly found in the CLOUDSC kernel is the coexistence of similar streams of computation dependent on an inner loop condition. Identifying and sharing resources among such

mutually exclusive blocks poses the opportunity to save a considerable amount of resources. However, we have not yet developed strategies that would allow us to discover and resolve these situations reliably in all circumstances.

The biggest obstacle in the manual porting process was mapping a kernel of this size to the limited real estate offered by the FPGA chip. The double precision version of the kernel used more than 90% of some of the available resources on the Xilinx Virtex UltraScale+ VU9P FPGA used for development, requiring the design to span all three Super Logic Regions (SLR). This, in turn, requires signals to be passed to all three SLRs to keep them in lockstep and makes it harder for the compiler to get timings right and produce a bitstream successfully. To mitigate this problem to some extent, the so-called pipelining factor was lowered to reduce congestion in the pipeline without having to lower the clock frequency too much and the kernel was split into three parts: setup, solve and diagnostics with the second one being by far the biggest and requiring the most DSPs, even after moving as many additional computations into the setup phase for later use as possible. In the end, only after applying these modifications a double precision design was successfully generated. As a consequence, capabilities to move code sections were integrated into the source-to-source translation toolchain and can be triggered by in-source pragma annotations.

The fact that CLOUDSC, like many other physical parameterisations, is very sensitive to numerical accuracy proved to be an obstacle in obtaining bit-identical results with the FPGA version and original CPU versions of the code. Tracking down the source of differences and proving correctness required in the end porting some of the Maxeler math library employed for frequently used exponential functions to CPU to be able to produce identical results, although different from the original CPU version.

In preparation for TB3, the EuroEXA system supporting the Maxeler programming environment, bitstreams had to be generated for Alveo U200 FPGA accelerators. The FPGA chip on that card is virtually identical to the one used on Maxeler MAX5 dataflow engines, which had been used for developing the FPGA port. This allowed re-using the existing implementation of the kernel without modifications and required only minor changes to the accompanying so-called manager that specifies build configuration and data streams.

Finally, a major enabler of successful development work and porting efforts was the Maxeler simulator. Considering that generating a bitstream for the double precision version takes in the order of days due to some of the resources offered by the FPGA being used almost entirely, this tool allowed testing correctness before undertaking the lengthy build procedure.

11.2 Integrated Forecast System (IFS)

The IFS as a whole has been tested on ARM platforms. The GNU compiler environment used to target EuroEXA host processors is capable of building the IFS and all its dependencies. The IFS in its entirety has not been ported to FPGAs, as this is well beyond the scope of the EuroEXA project. However, once the toolchain currently under development is able to target FPGA

execution from Fortran code and the software infrastructure for offloading parameterizations is in place in the full IFS, real tests can be carried out with specific physics components being offloaded to FPGA.

The horizontal discretisation in IFS is based on the spectral transform methods and requires transformation from spectral to grid point space and back in every time step. This involves transposing data across nodes and results in global communication patterns that could potentially benefit from the UNIMEM capabilities of the EuroEXA architecture.

Licence

Source code for IFS is proprietary to ECMWF.

11.2.1 Final ported application software

The atmospheric forecast model of the IFS, including physical parameterizations and wave model, was compiled for Aarch64 instruction set architecture. The MPI communication features used in the IFS, in particular in the communication-intensive spectral transforms, are all contained in the list of features supported by the UNIMEM communication layer, promising compatibility with the interconnect features.

11.2.2 Unique EuroEXA Features supported and exploited

The IFS can run on ARM CPUs similar to those used in the EuroEXA platform and make extensive use of the interconnect between nodes. Subject to the availability of an MPI stack for the UNIMEM capabilities of the EuroEXA architecture, this could yield significant improvements to the parallel performance, in particular in the communication-intensive spectral transforms.

11.2.3 Porting Issues and Lesson Learned

Generating binaries for applications to run on the CRDB nodes of test bed 2 requires compiling for the Aarch64 instruction set architecture. The default development environment for these nodes consists of a x86 Docker container that contains the necessary cross-compilation infrastructure to create binaries for the target system. Unfortunately, cross-compiling a complex application like the IFS is not a trivial endeavour. In particular some external dependencies, such as the widely-used HDF5 library, do not support cross-compilation. This is due to certain checks during the configure phase that are carried out by compiling and running small test programs to determine data type sizes. Naturally, this approach does not work when cross-compiling and, unfortunately, there is no officially supported way of providing this information in a different form. Instead of applying in-depth modifications to the build process, a qemu emulation environment was used with an Aarch64 Debian 9.6 image that comes with the same build toolchain as in the Docker container to produce binaries.

12 FORTH Applications

12.1 Satellite Image Processing

12.1.1 Final ported application software

In D2.2 we presented our early efforts on the hardware implementation of the inference stage of a Neural Network (NN); the Neural Network was developed by researchers at FORTH and it expects as input signals of 1800 features.

In D2.2 we created an early prototype of the architecture and we verified that our neural network is working properly both as a standalone design and as part of TB0. In this deliverable, we report the output of the next milestone, which involves the scaling of the architecture, both horizontally and vertically. We can achieve horizontal scaling with the use of multiple FPGAs and QFDBs.

For the vertical scaling of the architecture, we have moved in two dimensions. First, we increased the operating batch size in a sustainable way. In other words, we optimized the architecture so that it can support more than one query in parallel without a proportional increase of resources. Secondly, we mitigated the design in a larger platform, such as the CRDB platform, which features a VU9 Xilinx FPGA which provides approximately four times more resources than the one used on the QFDB

In this deliverable, the porting procedure provides post-synthesis results on the actual CRDB FPGA, combined with memory hierarchy design considerations from fully implemented and downloaded designs on the previous platform, the QFDB

PLATFORM REQUIREMENTS	Required	Achieved/Implemented
Hardware	CRDB – ported architecture from QFDB	Yes (achieved for QFDB – synthesized for CRDB)
Operating system	--	
Languages	C++	yes
Compilers	Vivado	yes
Programming models	OmpSs	Planned – the CRDB porting of our architecture supports a clock rate of 200MHz, which is necessary for the integration with OmpSs, and leaves enough FPGA resources for OmpSs.
Low-level libraries	-	
High-level libraries	-	

License

Proprietary to FORTH

The architecture of the Neural Network used (and the data that have been used for training) are proprietary.

12.1.2 Unique EuroEXA Features supported and exploited

The application runs on ARM CPUs in the EuroEXA platform and makes extensive use of the interconnect between nodes. It uses the ARM CPU to store model parameters and load them to FPGAs when necessary. ARM CPUs also handle inputs and route them to proper available devices. The goal is to achieve in the end an elastic application, which in the presence of high load can scale to multiple boards.

The CRDB hardware platform offers a large enough FPGA to provide enough parallelism for batches of up to 8-13 different inputs to be served in parallel without any latency or energy overheads

12.1.3 Porting Issues and Lessons Learned

Migrating the CNN design to the CRDB presented certain challenges. First of all, the custom IPs of the design have to be repackaged for the targeted part and we need to ascertain whether all Xilinx IPs used in the design are also available for the new target platform. We would then have to integrate the network into the CRDB architecture, which could potentially create problems or lower the performance of the network, due to sharing of processors resources. Problems with bus sizes are not anticipated and the design should pass Synthesis and Implementation with relative ease since the VU9 is a much larger FPGA. However, depending on whether or not an OS will be used, a driver may need to be written in place of the Vivado SDK program. It has to be noted here that the overabundance of resources makes batching most useful, but applying it would require a bottom up redesign of the architecture. Another important problem we found is that, in QFDB, our architecture depends on the ARM processor to bring input and state data in-and-out of FPGAs. In CRDB, we had to refactor our design to replace ARM cores. The solution that we gave was to replace the ARM processor with a soft-core MicroBlaze CPU. Anyway a non-trivial platform migration from QFDB to CRDB has been done.

13 Summary

This deliverable described in details final porting, optimization and adaptations of the applications on the EuroEXA platform based on available system software and runtime environments. The project is providing a set of runtimes and software and libraries designed optimized for EuroEXA testbeds (*e.g.* OmpSs, MaxJ, GAPSI, MPI etc). In the D2.4 we described their main features utilized by the applications as discussed in the sections above. Those runtimes and optimized libraries not only result into more efficient applications (*e.g.* in terms of performance or power consumption), but also impact positively on the porting activities simplifying the adaptation and also minimizing the hardware “lock in” that has a negative impact on applications maintainability beyond the project lifetime.

For each of the tools and applications, we present the work done and the relevance for the project, with a general view towards Exa-scale. The applications have been used also to test the platform, during the deployment phases. The results of the optimizations and the performance studies will be presented in the evaluation deliverable.

Respect to D2.5, we move the HPCG development and implementation discussion to WP3 deliverable D3.5.

Our work was extremely successful, and the results demonstrate the importance of a co-design that must involve all the layers of an HPC ecosystem: real application developers, system software and runtime developers and hardware developers.

13.1 Porting considerations and Lesson Learned

Beside the technical achievements, this work also allowed us to develop a good knowledge of the problems (and solutions) on porting and developing applications for Exa-scale platforms. Other communities and researchers could benefit of our experience, learnt lessons and failures.

Porting applications to the FPGA is a more complex process than porting it to other accelerators. This is a time-consuming task due to the long compilation times that leads to a slow feedback loop between the code modifications and the production of performance results.

Additionally, even though high level languages (*e.g.* C/C++ OpenCL) is supported by the Xilinx vendor, highly reducing the programming effort required, allowing a much higher software portability, we found that the ecosystem, on which OmpSs@FPGA and Maxeler compilers rely on, is not as mature as other traditional FPGA low level programming languages. On the other hand, the development of high-level frameworks, available in the context of the EuroEXA project, are fundamental towards a wide adoption of FPGAs in the HPC context. During the project also the high-level programming abstractions for FPGAs evolved (as discussed in WP3 deliverables), libraries are being developed, and leading to better programmability and simplifying code porting.

In the following, we summarize the main cross-cutting issues faced by all WP2 partners:

- all WP2 partners faced the challenge of code refactoring in order to exploit FPGAs on the CRDB. Most of the changes to the program are related to memory management between host and device. How the kernels communicate with the memory interface or among them is fundamental to achieve the desired performance from the FPGA devices;;
- at the same time, many applications benefit from the unique feature of FPGA accelerators given by the presence of on-chip memories, such as BRAM and URAM. These kinds of memories allow, for instance, to easily implement as a shift register the complex stencil operations on which many applications rely on. On other architectures, such as CPUs and GPUs, custom data layouts have to be implemented to mitigate the sparse memory accesses, avoiding the possibility of reaching the theoretical maximum memory bandwidth;
- another fundamental stage in the porting process was the design of the algorithmic iteration space. In practice, this means that loops had to be fused and/or reordered, heavily impacting the scope of local variables. Doing so is a tricky task, in particular for complex kernels with many loop-carried dependencies, and can be error-prone imposing an iterative trial-and-error style approach. The latest issue might be partially mitigated by automatic dataflow analysis tools to identify dependencies across variable scopes;
- scientifical codes are very sensitive to numerical accuracy, which requires most of the time double precision arithmetic, which is resource-hungry and (occasionally) performance-poor on FPGAs. Some WP2 partners experimented with data compression and reduced precision arithmetic if applicable as a feasible workaround to the limited data transfer rate, on-chip memory and DSPs. This imposed a further detailed analysis to provide a reliable implementation of the kernels;
- the design space exploration of code and performance in large FPGA like the VU9 on the CRDB is crucial. Analysis performance with HLS tools and scripting can allow to automate the process. However the most time consuming FPGA compilation is due to the place and routing processes during the bitstream creation. High Kolmogorov kernel complexity heavily impacts on the routability of the design of such a kernel. A lesson we learnt is that it is fundamental to split complex kernels in several compute blocks, to implement them as separated independent functions (function level pipeline), able to communicate between them thanks to data streams implemented as FIFOs in the FPGA.;
- latest OmpSs@FPGA advanced hardware features allow the programmer to constrain kernels to a particular Super Logic Region (SLR). Programmers benefit from these features, preventing the possibility that a kernel is automatically placed by Vivado across 2 SLRs, usually negatively impacting timing. Additionally, users can apply register slices in between the SLR crossings to further help timing at the cost of using additional fpga resources. The downside is that kernels have to be split into smaller kernels that fit single-SLR resource capability. The latest process cannot be performed on complex kernels, so the biggest obstacle in the porting process is mapping the kernels using >80% of DSPs on the EuroEXA platform, requiring the design to span all three SLRs;

- Applications are using either OmpSs@Cluster or a combination of MPI and OmpSs to use more nodes at the same time, and the results in terms of performance has been described in Deliverable 2.7

APPENDIX A Applications Technical Annex.

In this appendix we collect the technical aspects of two applications: Neuromarketing code and GADGET. In particular we summarize how to compile and run the codes on different architectures.

How to compile the Neuromarketing code

For the multi-node version (OpenMP in a single machine)

You need comment the lines 24 & 25 of *define_csp.fpga.h* in order to execute the code in host executing the following command: *g++ -fopenmp csp_tb.cpp MY_NNET.cpp*

You can select the number of kernels in line 105 of *define_csp.fpga.h*

For the MPI-enabled version (MPI installed required)

You need comment the lines 24 & 26 of *define_csp.fpga.h* in order to execute the code in host executing the following command:

* S/W compilation with MPI: *mpicxx -O3 csp_tb_mpi.cpp MY_NNET.cpp -mcmodel=large*

* S/W running: *mpiexec -np 2 ./a.out (total_nodes + 1)*

You can select the number of MPI-nodes in line 105 of *define_csp.fpga.h*

For the FPGA version

You need comment out the line 24 of define_csp.fpga.h in order to execute the code in FPGA executing the following command (from ompss_at_fpga_2.5.2-euroexa docker):

* make CROSS_COMPILE=aarch64-linux-gnu- BOARD=euroexa_crdb FPGA_HWRUNTIME=som FPGA_CLOCK=200 FPGA_MEMORY_PORT_WIDTH=512 bitstream-p

You can select the number of kernels in line 105 of *define_csp.fpga.h* and line 754 of MY_NNET.cpp

How to compile GADGET code

This is the application repository where all the related software versions have been uploaded: https://github.com/euroexa/Exa-HiGPUs.

The current application software, ported to EuroEXA CRDB, consists of the following files, along with a brief description of their functionality:

- *make.def*. It defines how the application is built:
    - to compile to x86, x86_64 architecture:
        - using the standard GCC selecting the CPU to generate code for at compilation time by determining the processor type of the compiling machine (*-march=native* flag). Moreover, the user is allowed to build different application versions, *i.e*. enabling/disabling MPI, enabling/disabling source code for automatic profiling (namely, time-to-solution of stages of the application, MPI overhead), enabling/disabling compiler optimizations, enabling/disabling debugging support, enabling/disabling the application to perform a single time step of integration for the aim of benchmarking architectures;
    - to compile using OmpSs targeting any supported FPGA by OmpSs@FPGA framework:
        - we take advantage of the feature of the source-to-source *OmpSs* (*Mercurium*) compiler of splitting the compilation stage of host code and FPGA kernels. The part of the code to be run on the host SMP can be compiled as stated above, while the user is allowed to build only the FPGA *IP block,* or the *bitstream* enabling/disabling the hardware instrumentation. In the first case the *OmpSs@FPGA* toolchain is stopped after Vivado HLS is issued, allowing the programmer to analyze in detail the throughput/area trade-off and experimenting with different *#pragma HLS* directives;
- *make.fpga*. It selects the FPGA model, the frequency and the memory port width of the bitstream. The file must be edited by the user before issuing the OmpSs@FPGA toolchain;
- *Makefile*. It orchestrates all the compilation options described above;
- *include/parameters.fpga.h*: this header file contains the macros to select at compile time the FPGA resources, namely the number of particle to load in *BRAM/URAM* per block (*BURST_LEN*) and the factor of unrolling (*FACTOR_UNROLL*) of the main loop of the algorithm. This header file must be edited by the user before issuing the compilation. The impact of such macros can be checked stopping the *OmpSs@FPGA* toolchain at the HLS step (see above);
- *include/constant.fpga.h*. This header file automatically places the local arrays on *BRAM/URAM* with the right array partitioning based on the unrolling factor set in *include/parameters.fpga.h*;
- *src/hw_evaluation_kernel.c*. It contains the kernel exploiting the FPGA. Our implementation guarantees that all the loops are performed in pipelines with an initiation

interval of one clock cycle. We rely on custom memory management (Nanos++ APIs) for memory allocation in kernel space and for memory movement from host DRAM and BRAM/URAM and vice versa; we select it as the most optimal solution to manage data movements.

LFRic Installation Procedure

Install/get_lfric.sh is an installation script for GNU/Linux machines. It tries to minimise the amount of external, globally installed packages by compiling a large part of LFRic's dependencies from source. Evidently, a basic building system must be in place to compile these dependencies and to avoid a chicken-or-egg conundrum, this is prepared using an available package manager, thus requiring root privileges. It has been successfully tested on:

- Ubuntu Server 18.04.5, 21.04 and 21.10;
- CentOS Linux 8.4.2105;
- Fedora Server 34.

Going forward, a Docker container image is probably a better solution than this potentially hard to maintain script. For convenience, *install/get_lfric.sh* creates *env_lfric.sh* which defines a few environment variables, including:

- the executable path, compiler, linker and loader flags;
- both build, namely optimisation and target platform, and execution settings:
    - variable PROFILE expands to production, for "risky optimisation"; the other options are full-debug and fast-debug for no and safe optimisations only, respectively;
    - variable LFRIC_TARGET_PLATFORM expands to meto-spice, enabling Psyclone's optimisations for the UK Met Office's SPICE system; the other options include meto-xc40, meto-xcs and monsoon-xc40 for their Cray XC40, Cray XCS and MONSooN systems, respectively;
    - variable OMP_NUM_THREADS expands to 1, for single-thread executions by default;
- some useful shortcuts:
    - variable TOPLEVEL expands to the directory where install/get_lfric.sh is executed from;
    - variable LFRIC_DIR expands to LFRic's installation directory, $TOPLEVEL/lfric-fallowdeer.1.16068 by default;
    - variable GUNGHO expands to the path of LFRic's main executable, $LFRIC_DIR/gungho/bin/gungho;
    - command *cds* is aliased to LFRic's source directory, cd *$LFRIC_DIR/gungho/source*;
    - command *cdm* is aliased to LFRic's build directory, *cd $LFRIC_DIR/gungho*.

Modified Source Files

Under modded you will find all source files Ashworth et al. originally created or modified. We, Nobre et al., have since enabled the execution without configured FPGA communication device ports, simplified makefiles, fixed some bugs, cleaned and refactored the code and performed some optimisations as described below.

The contents of this folder may be copied over LFRic's original sources, e.g. *cp -R modded/. $LFRIC_DIR*. Compile just as exemplified in *install/get_lfric.sh*, i.e. *cdm*; *make clean build*. Next, to identify execution hotspots and gauge our progress, we've been using the following benchmark.