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

http://textarossa.eu

WP4 Tool chain for heterogeneous multi-node HPC platform

D4.6 Task-based runtime systems

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

TEXTAROSSA

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

Grant Agreement No.: 956831

Deliverable: D4.6 Task-based runtime systems

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

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

Deliverable No	D4.6
WP No:	WP4
WP Leader:	BSC
Due date:	M30
Delivery date:	30/11/2023

Dissemination Level:

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

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

DOCUMENT SUMMARY INFORMATION

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

WP4 Tool chain for heterogeneous multi-node HPC platform

Deliverable number:	D4.6
Due date:	M30
Actual submission date:	02/12/2023
Editor:	Carlos Álvarez
Authors:	Carlos Alvarez, Antonio Filgueras, Daniel Jimenez, Xavier Martorell, JuanMiguel de Haro, Bérenger Bramas
Work package:	4
Dissemination Level:	Public
No. pages:	45
Authorized (date):	
Responsible person:	Carlos Alvarez
Status:	Plan Draft Working **[Final]** Submitted Approved

Revision history:

Version	Date	Author	Comment
0.1	2023-10-27	Carlos Álvarez	Draft structure
0.2	2023-11-03	JuanMiguel deHaro	Task-based for FPGAs
0.3	2023-11-07	Carlos Álvarez	First Final version

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Giuseppe Zummo	2023-11-29
Checked by Task Leader	Carlos Alvarez	2023-11-29
Checked by WP Leader	Berenger Bramas	2023-11-30
Checked by Project Coordinator	Massimo Celino	2023-12-01

COPYRIGHT

ACKNOWLEDGEMENTS

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

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

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

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

DISCLAIMER

Table of contents

Executive Summary

This deliverable provides the preliminary results on the adaptation of two programming models OmpSs (task-based) and StarPU (task-based) to the TEXTAROSSA computing nodes. It summarizes the objectives for both runtime systems and the current progress status. Both OmpSs and StarPU are task-based runtime system, but in the project OmpSs focuses more on the use of FPGA at a low level, whereas StarPU is used at higher level by considering the FPGA as any classical processing unit, such as GPU, and focuses on scheduling.
The work package also includes activities related to mixed precision technologies (tied to WP6), inter-FPGA communication SW stack (tied to WP2), and power modeling and use in runtime systems (tied to WP3). The current document describes the work related to facilitate the use of these technologies into the task-based programming models, but the core activities are performed in the other WPs, and WP4 is used to provide their software interface or study their inclusion in the runtime systems. Consequently, only activities that imply changes in the programming models' framework (compiler and runtime) are included in this deliverable.
As described in D1.1 and D1.2, WP4 uses IPs and technologies developed in other WPs and will be used by applications in WP6.

1 Introduction

WP4 focuses on the middle layer of the TEXTAROSSA project. It is tied to hardware and software, making the bridge between hardware features and software interfaces. Consequently, most WP4 activities are highly coupled with other WPs. The main activities of WP4 include the improvement of the streaming and task-based programming models to computing nodes with FPGAs. In this deliverable we focus on the task-based programming models: OmpSs and StarPU. Both runtime systems were mature and efficient before the beginning of the project. The current document summarizes the progress that has been done on both tools and provides their improvements. It also includes the additions to the programming models related to the other technologies addressed in WP4.

It should be noted that this WP does not provide a single unified system where all applications are expected to use all features. Instead, it provides building blocks that allow for efficient hardware utilization, which will be validated using micro benchmarks and WP6's applications.

1.1 Objectives

FPGAs are widely used as accelerator devices because they provide high levels of performance and energy efficiency. However, programming such devices involves the use of specific tools and techniques, and even hardware skills to develop a baseline application due to interfaces, data transfers, etc. This causes reluctance when using them by programmers, or completely make them not use them.

Task based programming models such as OmpSs and StarPU provide a good opportunity to abstract the underlying hardware complexity, so implementation effort is kept low while maintaining good levels of performance. The objectives of each programming model (further described in its own section) are related to the project objectives:

- Objective 1 - Energy efficiency. Executing in FPGA has been demonstrated to be competitive with other computing platforms in terms of energy efficiency. In addition of providing the support to executing in the IDV-E platform, the OmpSs task-based model is integrated with power measurement tools in order to be able to further control and improve the energy spent when executing in the platform.
- Objective 2 - Sustained application performance. As explained in the next sections, we aim to improve the performance obtained when executing applications over the IDV-E platform both by improving the framework and also by improving the task scheduling through the use of the Fast Task Scheduler developed in Task 2.5.
- Objective 3 - Fine-tuned thermal policies integrated with an innovative cooling technology. As explained in Objective 1, the power measurement tools integrated in the OmpSs framework for IDV-E provide the basis for integrating fine-tuned thermal policies developed in Task 4.5.

- Objective 4 - Seamless integration of reconfigurable accelerators. The task-based runtimes allow for seamless integration of reconfigurable accelerators as can be seen in their respective sections.
- Objective 5 - Development of new IPs. The Fast Task Scheduler IP is a key part of the OmpSs@FPGA framework. OmpSs@FPGA contributes to the IP development as a primary tool to test the IP functionality. It also provides design requisites that must be incorporated in the IP for the whole framework to work as expected.
- Objective 6 - Integrated Development Platform. Task based runtimes will be used in applications executing on the project platforms. It is important to highlight that IDV-E features a host CPU (ARM based) that has never before been used to drive computation in a PCIe attached FPGA. Developing the system in a way that is compatible with new different CPUs helps ensuring new host CPUs (like EPI CPUs) will be able to drive this kind of computations in the future.

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

- Strategic Goal #1: Alignment with the European Processor Initiative (EPI). As shown in this deliverable the OmpSs@FPGA task-based programming model provides a system that can use an EPI processor to drive computations in a cluster of FPGA PCIe attached accelerators. Also, as described deliverable 2.11, the programming model allows to manage a manycore RISC-V processor with significant performance improvement over other state-of-the-art approaches.
- Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The OmpSs@FPGA framework is developed following the open-source model and is freely available in its github repository.
- Strategic Goal #3: Opening of new usage domains. The task-based frameworks address the problem of simplifying the task of executing applications over FPGA-based computing platforms. In this sense, we expect that through the improvement of the tool, it will open the possibility of executing efficiently new applications on the objective platforms.

These objectives are further discussed in the next sections.

2 StarPU - Task-based scheduling and use of FPGA

Leader: Inria

2.1 Presentation

StarPU is a task-based runtime system designed from the beginning to support heterogeneous architectures. Since then, it has been improved to support distributed memory parallelization (on top of MPI). Its internal structure allows for the development of new schedulers very easily. Several scientific computing applications use StarPU, such as an FMM solver [SCALFMM], a dense linear algebra solver [CHAMELEON], a sparse linear algebra solver [PASTIX], an H-Matrix solver [HMAT], a machine learning framework for climate/weather prediction [ExaGeoStat], a quantum Monte Carlo kernel library [QMCkl, a Navier-Stokes solver [FLUSEPA], and others.

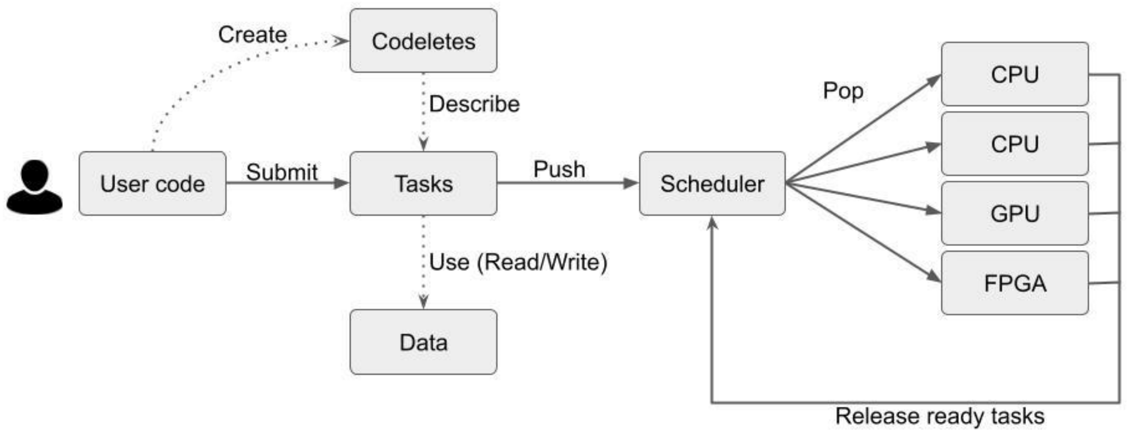


Figure 2.1: StarPU internal structure. The scheduler has access to all the ready tasks and decides how they should be distributed.

The scheduler is a critical component in any dynamic runtime system because it decides the order of execution of the ready tasks and on which processing unit the tasks are executed. Therefore, these decisions impact the execution duration and the amount of memory transfer between memory nodes.

2.2 Objectives

The use of FPGA in task-based parallelization is still an open problem and has not been investigated in depth in StarPU. This is why, in Textarossa, we will attempt to better understand when FPGA can be beneficial, how to schedule the tasks, and if we can save energy.

2.3 Status

In the context of the project, we have created a new scheduler called multreeprio and we have performed the initial test in using FPGA with StarPU.

Multreeprio is a dynamic task scheduler that aims to minimize the overall completion time of parallelized task-based applications. The goal is to find a trade-off between resource affinity, task criticality, and workload balancing on the resources. To this end, we compute priority scores for each task and manage the available tasks in the system with a data structure based on a set of binary trees. Tasks are assigned to available resources according to these scores, which are dynamically computed by heuristics based on task affinity and criticality. We also consider workload balancing across resources and data locality awareness. To evaluate the scheduler, we study the performance of dense and sparse linear algebra task-based applications using the StarPU runtime system on heterogeneous nodes. Our scheduler shows interesting results compared to other state-of-the-art schedulers in StarPU.

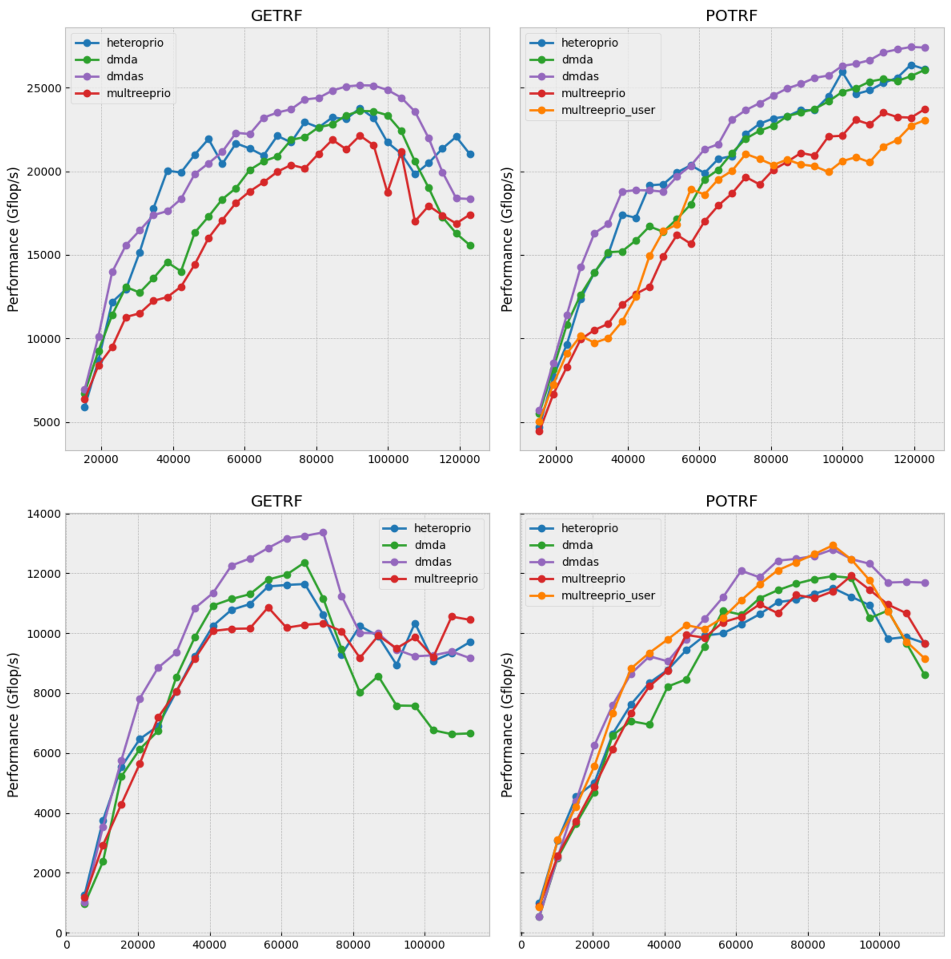


Figure 2.2: Performance results of the multreeprio scheduler against state-of-the-art scheduler for classic linear algebra kernels (Chameleon) on A100 (top) and V100 (bottom).

To utilize FPGAs within StarPU, we employ the OpenCL interface to access and manage Xilinx devices. Prior to the start of the project, StarPU already offered support for OpenCL.

However, several modifications were required to facilitate its use. Foremost, the OpenCL kernel should only be allocated once on the device. This approach contrasts with GPUs/CPUs, where OpenCL kernels exist as binary code for execution. In the context of FPGAs, allocating a kernel involves configuring the FPGA, which ideally should be done once.

Consequently, our first modification involved adding two extra steps: initializing and releasing the FPGA devices. To accomplish this, a straightforward call to a function that facilitates execution on the workers is sufficient.

The second modification aims to ensure a coherent execution of OpenCL tasks. This can be achieved either by the user, who can verify the code to be executed within the task, or by StarPU itself, which can restrict execution to certain devices only. The version of StarPU that is provided in this project includes a filter module to select only some of the devices out of all the devices that are present on the computing node.

Currently, we do not have performance or energy results available. However, we aim to obtain some for Chameleon and Scalfmm by the end of the project. These will be included either in an updated version of the current deliverable or in the deliverables produced at the end.

The software stack for utilizing OpenCL with Xilinx FPGAs has proven to be less robust than anticipated. For instance, HWLOC was generating illegal instructions, and using CL_DEVICE_TYPE_ACCELERATOR instead of CL_DEVICE_TYPE_ALL resulted in a segmentation fault by the Xilinx library. This has delayed our benchmark campaign.

3 OmpSs@FPGA - Task-based runtime for FPGAs

Leader: BSC

3.1 Presentation

OmpSs is a task-based programming model. OmpSs has been designed to be non-invasive so that minimal changes are needed to port and parallelize an application. In particular, pragma directives annotations are used to allow productive parallel programming, even for heterogeneous architectures, such as GPUs and FPGA.

In the last part of the project development presented in this deliverable we have completely overhauled the framework to update it. Now the OmpSs@FPGA framework leverages the new state-of-the-art programming model OmpSs-2. Also, the compiler tool Mercurium used in previous versions has been deprecated in favor of an in-house developed LLVM/Clang fork compatible with newer C++ standards. Finally, the new system is compatible with the new Vitis HLS flow that has substituted the previous Vivado HLS system. In addition, our carefully implemented framework maintains backward compatibility with the old tools. More details about the new implementation can also be found in deliverable 4.7 HLS Flow, section HLS in the Task-based model and in deliverable 4.8 Framework for efficient CNNs inference on a TEXTAROSSA node, section 6.2 Compiler modifications for CNN mixed precision. For the sake of clarity, here a brief reference introduction to the new updated framework follows.

3.1.1 OmpSs@FPGA code example

In the OmpSs-2 programming model (that for simplicity from now on will be referred as OmpSs), the programming model directives allow the user to specify the target device of any given task as well as the input that it requires and the output data it produces. This input and output data specification allows the runtime to detect dependencies among tasks to ensure a correct task execution scheduling. Listing 3.1 shows a function defined as a task, that will be submitted as task for any `vecSum` function call in listings 3.2 and 3.3. The pragma "`oss taskwait`" at the end of the code is necessary to guarantee completeness of all the tasks. However, it is important to highlight that the OmpSs programming model in this new iteration advocates using as few `taskwaits` as possible. As a result of the code shown, tasks "`vecSum`" will be offloaded into an FPGA accelerator (as indicated by the "`device`" clause). Those tasks will require input data "`a`", which is a vector with 16 elements and a parameter of the function, and input data "`b`", also a vector with 16 elements. In addition, each task produces (updates) vector a. All input and output data requirements are transferred to/from the device automatically and transparently to the programmer.

```
#pragma oss task device(fpga) inout([16]a) in([16]b)
void vecSum(float a[16] , float b[16]) {
for (int i = 0; i < 16; ++i)
```

```
        a[i] += b[i];
    }
```

Listing 3.1: Sample FPGA accelerated task

The compiler also uses local memory in order to optimize memory accesses. This feature can be declared with the `copy_in`, `copy_out`, `copy_inout`, `copy_deps` clauses and is activated by default with dependence (`in`, `out` or `inout`) clauses. A variable stored in local FPGA memory is automatically declared, and data will be copied to/from this variable by the wrapper prior to/after the kernel execution. Accesses to the variable will reach the local memory instead of the global one. Without the mentioned clauses to enable this optimization, pointer dereferences generate an access to off-chip memory.

In order to execute multiple instances of the same task call in parallel, resources, i.e. accelerators, can be replicated. The clause `num_instances(N)` allows specifying the number of times that a task accelerator is instantiated in the FPGA. The OmpSs@FPGA user guide [OMPSSGUIDE] defines all specific OmpSs@FPGA clauses and their usage.

```
float vec_a[16], vec_b[1024];
for (int i = 0; i < 1024; i+=16)
     vecSum(vec_a, &vec_b[i]);
#pragma oss taskwait
```

Listing 3.2: Minimal example of C code invoking sequential tasks

Listing 3.2 shows how a code invoking the task defined in listing 1 works. As can be seen in line 5 of the code, invoking a task is as simple as calling the function that the task performs. With the simple code shown in listings 3.2 and 3.1 the user will be able to program an FPGA IP that receives the data from the CPU host, computes the vector addition and sends the output back to the host without taking care of the cumbersome data movements and/or IP managing details.

Another important feature of the OmpSs model is that it allows the Nanos6 runtime to implicitly extract parallelism. From the previous listing 3.2, Nanos will be able to know that invocations of task `vecSum` should be executed sequentially as the output of each invocation is the input of the next invocation. Consider now listing 3.3 that executes a slightly modified main program.

```
float vec_a[1024], vec_b[1024];
for (int i = 0; i < 1024; i+=16)
     vecSum(&vec_a[i], &vec_b[i]);
#pragma oss taskwait
```

Listing 3.3: Minimal example of C code invoking parallel tasks

As can be observed in listing 3.3, all the invocations of `vecSum` can be executed in parallel because their input and output dependencies do not overlap in memory. The Nanos6 runtime will be able to detect this situation and execute the code in listing 3.3 using all the available hardware resources in parallel. As an example, adding clause `num_instances(2)` at the line 1 pragma of listing 3.1 will result in nearly doubling the performance of the program without any other programmer involvement.

3.1.2 Nested FPGA tasks

One of the most innovative features of the OmpSs@FPGA model is the capacity of creating and managing tasks inside the FPGA (i.e. without CPU host intervention). The feature intends to break the master-slave model for FPGA devices in task based parallel programming models. The main goal is to allow interaction of the FPGA device with the parallel programming runtime to make the FPGA cooperate in the application execution beyond the current offload model. The new interaction capabilities include the creation of nested tasks inside an FPGA task and its synchronization.

Listing 3.4 shows the same code of listing 3.3 but with a function `largevecSum` also annotated as an FPGA task. To the best of our knowledge, the call to `vecSum` of line 7 is not possible in on other programming models aside OmpSs@FPGA. Indeed, in the programming model side, our proposal adds support for task calls inside code regions annotated with the `device(fpga)` clause. Note that those task calls are converted into runtime API calls by the compiler, and therefore, a new design and implementation of the model is needed to support such runtime APIs inside the FPGA devices.

```c
float largevec[SIZE*16];
float anothervec[SIZE*16];

#pragma oss task device(fpga) inout([SIZE*16]a) in([SIZE*16]b)
void largevecSum(float a[SIZE*16] , float b[SIZE*16]) {
     for (int i=0; i<(SIZE*16); i+=16)
          vecSum(&largevec[i],&anothervec[i]);
     #pragma oss taskwait
}

int main() {
     largevecSum(&largevec[0],&anothervec[0]);
     #pragma oss taskwait
}
```

Listing 3.4: Minimal example of C code invoking parallel tasks

All runtime API calls are supported inside the FPGA by using queues, where the task accelerators write requests to the runtime. Then, the runtime reads these requests and makes the needed actions. Some interactions are optimized and directly read and handled inside the FPGA, avoiding the latency between the host and the device. For this direct FPGA task execution, a hardware runtime support inside the FPGA has been designed and implemented. The hardware runtime part coordinates with the host runtime when needed to correctly execute the application.

3.1.3 Compilation process

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

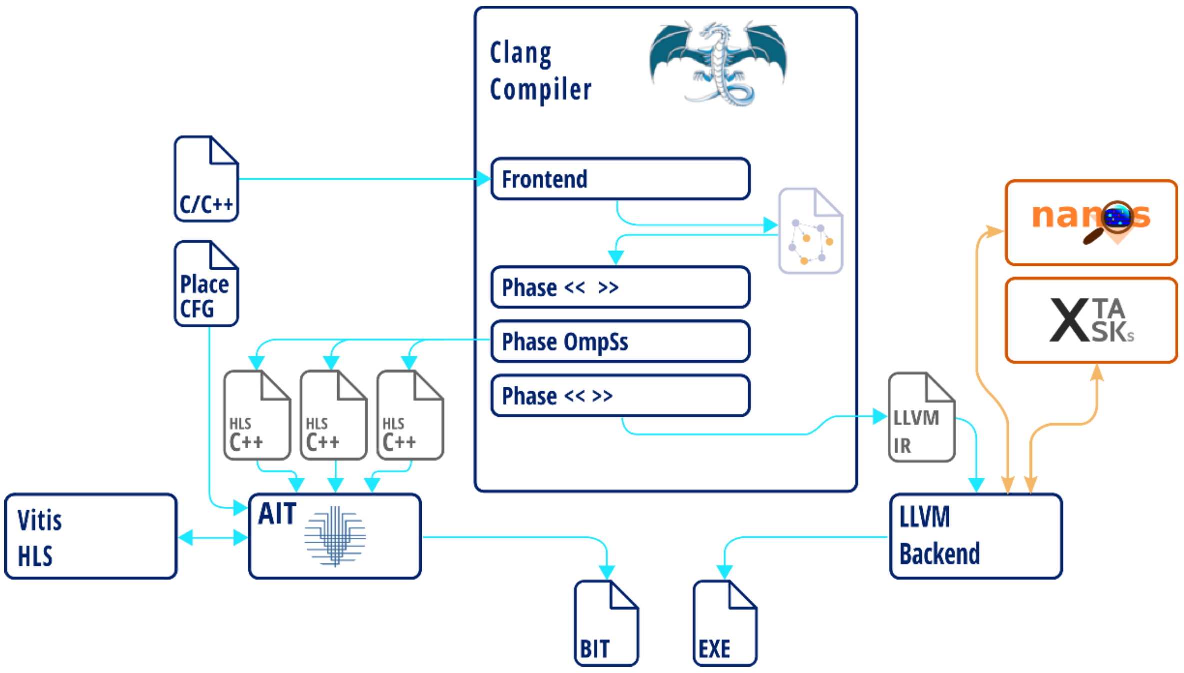


Figure 3.1: OmpSs@FPGA compilation flow

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

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

3.2 Objectives

FPGAs are widely used as accelerator devices because they provide high levels of performance and energy efficiency. However, programming such devices involves the use of specific tools and techniques, and even hardware skills to develop a baseline application due to interfaces, data transfers, etc. This causes reluctance when using them by programmers, or completely make them not use them.

Task based programming models such as OmpSs provide a good opportunity to abstract the underlying hardware complexity, so implementation effort is kept low while maintaining good levels of performance. The objectives of the OmpSs programming model in the Textarossa project can be summarized as follows:

- Provide support to the execution of task-based parallelized programs (with the OmpSs programming model) on Textarossa IDV-E platform.
- Improve performance of such task-based parallelized programs so it can be competitive with alternative current state-of-the-art programming models.
- Explore how task-based and stream programming models could be mixed to obtain systems that get the best of both worlds in terms of programmability, performance and energy efficiency.
- Support the Fast Task Scheduling Hardware developed in Task 2.5 and leverage it to accomplish the previous objectives.

The last objective is in fact accomplished by fulfilling the three previous objectives. All the objectives have been accomplished as described mainly in the next sections and to some extension in deliverable 4.7 HLS Flow, section HLS in the Task-based model and in deliverable 4.8 Framework for efficient CNNs inference on a TEXTAROSSA node, section 6.2 Compiler modifications for CNN mixed precision.

The results of the research reported in this deliverable have been published in three conference papers. A first research paper at the 9th BSC Doctoral Symposium [FILBSCDS] covered the initial implementations of the task-based programming model for PCIe attached large FPGA boards and how to improve the applications resource usage. The second research paper published at the 32nd IEEE International Symposium on Field-Programmable Custom Computing Machines [FILFCCM] included how a set of memory and placement improvements to the framework allowed different precission applications to take advantage of multi-SLR FPGAs (as the one used in IDV-E). Finally, a third paper [FILFPT] has been accepted for presentation at the upcoming International Conference on Field-Programmable Technology (FPT'23). This last conference paper evaluates how different applications benefit from the improvements developed in this work package. Also, a journal research paper titled "Automated parallel execution of distributed task-graphs with FPGA clusters" covering the joined stream and task-based programing models' proposal is under review by the Future Generation Computing Systems journal.

3.3 Accelerator Placement, Interleave and Priorities

In modern FPGA devices, place and route has become an increasingly difficult task due to an increase in resources and device complexity. This results in an exponential increase in implementation possibilities. Such a huge search space causes tools to have a hard time providing a good solution. This is even more challenging in chiplet-based devices due to their topology. In the same way, off-chip memory resources have grown both in size and number of modules. These resources are presented to the user as raw memory interfaces requiring the user to manage how accelerator kernels access off-chip memory to make effective use of the available bandwidth. Efficient usage of memory resources becomes a critical challenge as more computational resources are added to a design imposing more pressure on the memory subsystem. This section describes a set of new features added to the OmpSs@FPGA programming model and runtime in order to mitigate these issues in a transparent way for programmers.

3.3.1 Accelerator Placement

Acceleration in multi-SLR FPGAs may result in a below expected resource usage by the model due to the larger costs of propagating signals. To mitigate these larger costs of propagating signals across large regions, specially between different SLR, the accelerator placement feature allows users to assign specific computation kernels to different SLR.

This serves two main purposes. First, by constraining an IP core to a single SLR region, it prevents the implementation tool from placing it across different SLR. When this happens, the design can fail to implement because it needs more SLR crossing points than are available on the device as region crossing is a very limited resource [17, 18]. Even if design can be implemented, design quality of results (QoR) can be dramatically affected because the delay introduced in SLR crossing limits maximum clock frequency.

Also, by letting the user assign kernels to specific regions, it defines which signals have to cross an SLR boundary. This allows you to automatically place register slices on accelerator interfaces. These register slices are configured and constrained to specifically perform region crossing. They are placed around the boundary of the region and pipelined to propagate signals to points that are far away from the boundary without degrading design QoR.

Code 3.3.1.1 shows a placement configuration file used in the Alveo U200 FPGA. In this file, users specify the target SLR for each accelerator instance. This file is read by AIT to build and constrain the design. It is represented as the Place CFG input file in listing 3.5

```
{
    "first_acc"  : [ 0, 0, 2 ],
    "second_acc" : [ 1 ]
}
```
.

Listing 3.5: Placement Configuration File

In this case, there are three instances of first_acc, two of them are placed in SLR 0 and one of them in SLR 2. The only instance of second_acc is placed into SLR 1.

Figure 3.2 shows a schematic view of a design generated using the OmpSs@FPGA framework. Figure 3.2a shows a design that's generated without considering the different SLR, treating the device as if it was a monolithic design. When enabling placement feature, the resulting design is represented in 3.2b.

Note that in top of prevent accelerators from being placed among two SLRs, we also insert slices in elements belonging to the static part of the design, sometimes referred as the shell, in addition to accelerator data interfaces.

It is also worth noting that inside the same region, no slices are inserted. This is because interconnect IPs have all their interfaces registered, therefore, no extra register slices are needed inorder to keep good QoR. Furthermore, adding extra registers can worsen congestion, which can lead to lower operating frequencies.

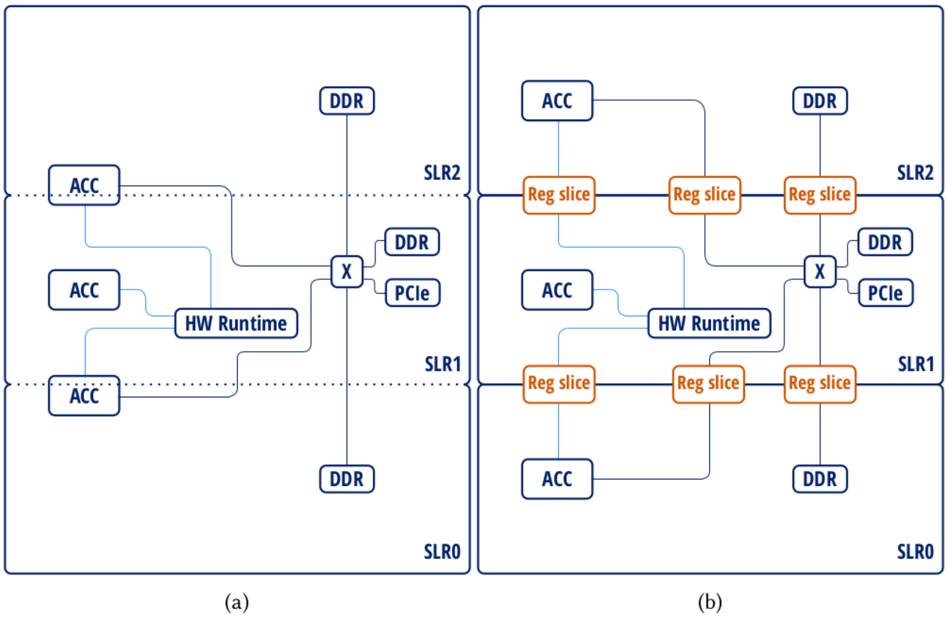
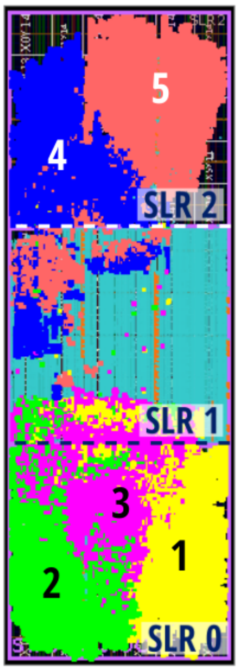


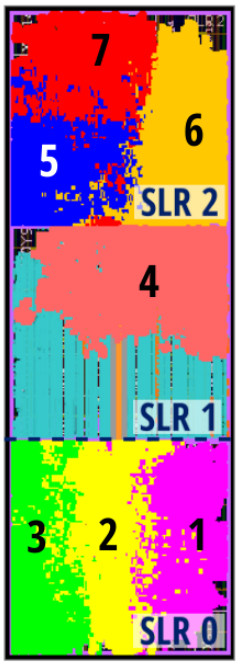
Figure 3.2: Design diagrams when not specifying placement (a) and with placement enabled (b)

Figure 3.3 shows a physical representation of the device. They show how components are laid out in the target device after the implementation phase. Highlighted and numbered areas belong to the computing accelerators, non-highlighted parts represent the shell and dark areas represent unused resources.

Figure 3.3a shows how a design without placement would be implemented. This would be the implementation of a design like the one represented in represented in figure 3.2a, using 5 accelerators. Applying the proposed placement improvement, results in figure 3.3b, which would implement a design that follows the scheme shown in figure 3.2b, increasing the number of accelerators up to 7. In this case, we were able to fit two more accelerators and better exploit device resources. This can be seen as a reduction of the dark areas in figure 3.3b.



(a)



(b)

Figure 3.3: Physical representation of implemented designs without (a) and with placement constraints (b)

In addition, even though we have implemented those features specifically for SLR, arbitrary placement regions can be defined and treated the same way. This is especially interesting for devices such as the Alveo U280. In this device, input/output pins are placed in a narrow column at the center of the device where user logic cannot be placed. This acts like an SLR boundary as crossing this gap, introduces some delay in signal propagation.

3.3.2 Memory Interleaving for DDR channels Accelerator Placement

We have implemented a general and transparent way to efficiently place application data into separate memory modules. Even though this can be achieved by manually placing application data in a specific layout to use all memory modules, this would result in increased development effort as it may not be a trivial task.

The goal is that accelerator accesses can be scattered across multiple memory interfaces in order to reduce access conflicts when several accelerators need to access data that otherwise would be stored in the same memory module. This is implemented by inserting an “interleaver” module between accelerator memory interface and the memory interconnection and between any other IP that access off-chip memory, such as the PCIe block, they are shown in figure 3.4, labeled as IL.

Memory interleaving modules are inserted between master and slave memory interfaces. They overwrite the transaction memory address in order to scatter access among all available interfaces. The address rewrite process is implemented by performing a series of bit selections and shifts, in a similar fashion as how chip, bank or row are selected in DRAM memory modules. In our case, however, no individual select signals are generated, instead, the address is reassembled and sent through the memory interconnection. External memory interfaces are mapped contiguously, therefore, rewriting the address is enough to send the transaction through the proper memory interface. This also provides a modular solution as no other part of the design, such as memory interface configuration or interconnection must be modified in the design to enable interleaving.

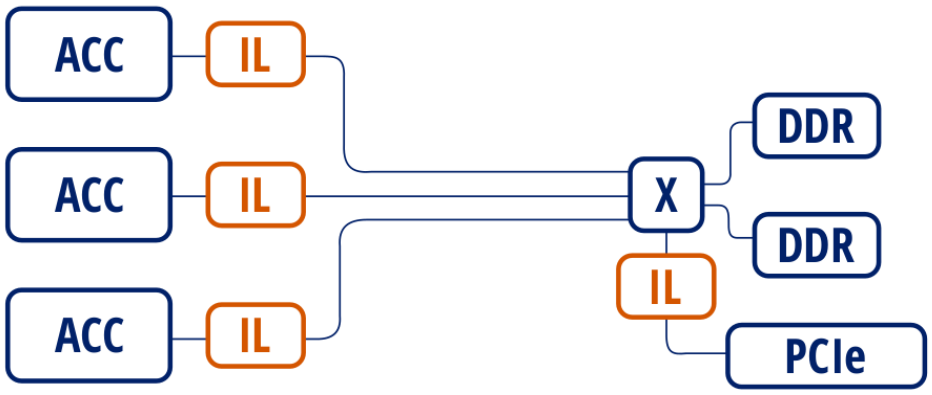


Figure 3.4: Memory interconnection diagram with interleaver modules (IL)

However, there are a number of requirements in this solution. One of them is that memory module size, as well as the number of modules must be power of 2 so address rewrite can be implemented using bitwise operations. This happens to be the case for most available FPGA devices. If this was not the case, we can always split a memory interface address space into multiple ones (high/low halves, for instance) to get a number of regions that is a power of 2. This, however, can cause imbalance as the divided interface will receive double the number of accesses, but even in this case, overall bandwidth is expected to improve.

Another requirement is the size of the interleaving stride. On one hand, it must be larger than 4K, to keep AXI4 compliance [AMBAAXI]. Keeping interleaving stride above certain size, also allows to take advantage of large bursts and transaction pipelining. On the other hand, it must be power of 2 to allow bitwise operations to be performed. However, as this is a user defined parameter, these requirements are not problematic in the final design.

3.3.3 Memory Priorities

An adverse effect observed in some applications is memory access conflicts. When more than one accelerator tries to access a single memory interface, one transaction is processed while the rest have to wait for the one in progress to finish. This presents two main issues. The default behavior for the memory interconnect crossbar is to process incoming transactions in a round-robin fashion across all its slave interfaces. In this configuration, the interconnect core processes a transaction from one accelerator, and then processes transactions from another one, preventing transaction pipelining as no more than one transaction from the same accelerator will be processed in sequence. By preventing pipelining, overall throughput is reduced. This is because all transactions must pay the cost of initiating a new transaction, which is a non-negligible latency.

By enabling priorities, this latency can be hidden by allowing transactions to be pipelined. In this case a single accelerator (the one with higher priority) will send multiple transaction requests in a row and data can be transferred at maximum throughput. Even though this effect will only appear when there is contention in memory accesses, enabling priorities in an application with low memory bandwidth demand, will not cause any performance degradation. Resource wise, enabling priorities do not have any meaningful impact when compared with default round robin scheduling.

The other issue that causes default transaction scheduling, and by enabling priorities we can mitigate, is that it slows down data copies for all accelerators even if the aggregate bandwidth remains the same. This causes further performance degradation in cases where accelerators have separate copy and computation phases, which is a common pattern. By enlarging the data copy phase of all accelerators, the relative amount of time spent in computation is reduced, decreasing overall performance. By letting one accelerator finish its copies earlier, it also improves contention as the computation phase is usually less memory intensive. This allows the rest of the accelerators to progress faster.

3.3.4 Summary of improvements

Figure 3.5 shows two execution traces showing the internal state of FPGA accelerators across the same given time range. Each row of the traces represents a different matrix multiplication accelerator instance, in this case they are instances of the matrix multiplication kernel. The horizontal axis represents the execution time. Each color region corresponds to a different accelerator state. Which can be any of the following: Data copies from off-chip memory to accelerator local memory (gray), computation (blue) and data copies to external memory (red).

Note that figure 3.5 shows only six accelerators instead of the seven that we could fit by using placement improvements, explained above. This is because the logic needed to capture events from the accelerator does not allow to implement a design using 7 accelerators due to place and route conflicts. However, data should still be representative.

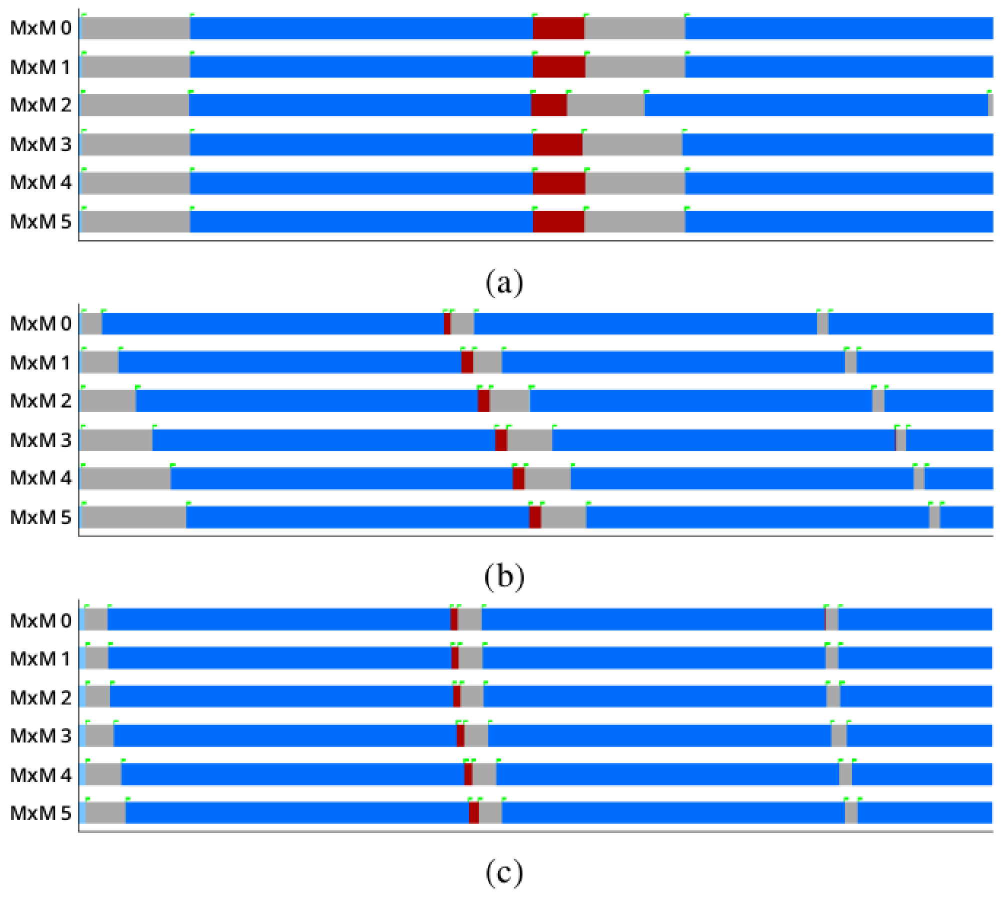


Figure 3.5: Matrix multiply execution traces with placement enabled, without (a), with memory access priorities (b), and with priorities and interleave (c)

An execution trace without memory access priorities is shown in figure 3.5a. In this case all accelerators spend roughly the same amount of time copying data due to round robin scheduling performed by the memory interconnect. Once memory access priorities are enabled, the effect is clearly shown in figure 3.5b as copy regions (gray regions) at the top are smaller than the ones at the bottom. This causes upper accelerators to finish their copies and start computation earlier than if no priorities were used. This also causes the side effect of spreading subsequent memory accesses, further improving memory access congestion. Accesses tend to line up in a way that conflicts are greatly reduced. This effect is especially notorious when tasks are created from FPGA accelerators (FPGA nested tasks), where tasks are created much closer together compared with tasks being created from the host due to accelerators having much higher throughput creating tasks than the host [BOSCH2020]. It is worth noting that even though accelerators in bottom rows can be negatively impacted in some cases, overall, time spent in data movements throughout execution is reduced, resulting in increased performance. More precisely, we spend 59% less time performing data copies when compared to a design that does not use priorities in instrumentation executions.

In our case, priorities to accelerator interfaces are not assigned in any way. The only goal is that one accelerator can process as fast as possible. Priority assignation could be user defined, allowing further tuning by users. PCIe interface, however, has the highest priority, this is done to allow data copies from the host to be finished as soon as possible, allowing more tasks to be able to start due to their data being ready. Starvation is not a problem in this approach. In the worst case, accelerators with higher priority will eventually consume all available tasks of their type and will enter idle state, allowing other accelerators to access memory resources. Therefore, application progress is guaranteed in any case. However, this is not the typical case, as accelerators usually have phases where no memory access is performed, allowing accelerators with lower access priority to progress.

Effects of interleaving can also be seen in figure 3.5c. When multiple copies overlap, copies (gray and red regions) take significantly less time than in cases without interleave, shown in figures 3.5a and 3.5. This is caused by interleaving spreading accesses to different banks, allowing them to be performed in parallel. By applying interleaving, 64% less time is spent moving data from or to off-chip memory when compared with the baseline design. When combining priorities and interleaving, time spent in copies is 21% less when comparing with a design only using priorities and 68% less compared to the baseline design.

Figure 3.6 shows the speedup achieved by using the proposed features across different applications and data types:

- MM-half, single and double: Matrix multiplication in half, single and double precision, respectively. Matrix multiplication is a well-known embarrassingly parallel application. The application computes $C = C + A \times B$, being A, B and C matrices of size $N \times N$.
- Cholesky: This benchmark performs the Cholesky decomposition of a real Hermian positive definite matrix A into a lower triangular matrix L. Multiplying L by its transpose, results in the original matrix $A = L \times L$ T. In the same fashion as the matrix multiplication kernel, the input matrix A is distributed in square blocks of size $BS \times BS$ single-precision elements.
- N-body: The N-body simulation computes how a group of particles with different masses interact with each other due to gravitational forces over a period of time. Algorithm input is a set of particles, each one consisting of an initial position, mass, and initial velocity. Position and velocity are 3-dimensional single precision floating point vectors, while mass is a scalar value. The output of the algorithm is the set of particles with their positions updated due to gravitational interactions after a given amount of time steps.
- Spectra: The Spectra application computes a histogram of electronic weights between particles versus distance for a given set of particles. To do so, it needs to compute the distance between each pair of particles and then add their electronic weight to the histogram. The histogram is afterwards used to compute the X-ray spectrum of the physical material being analyzed allowing the determination of the material composition [GONZ2022].

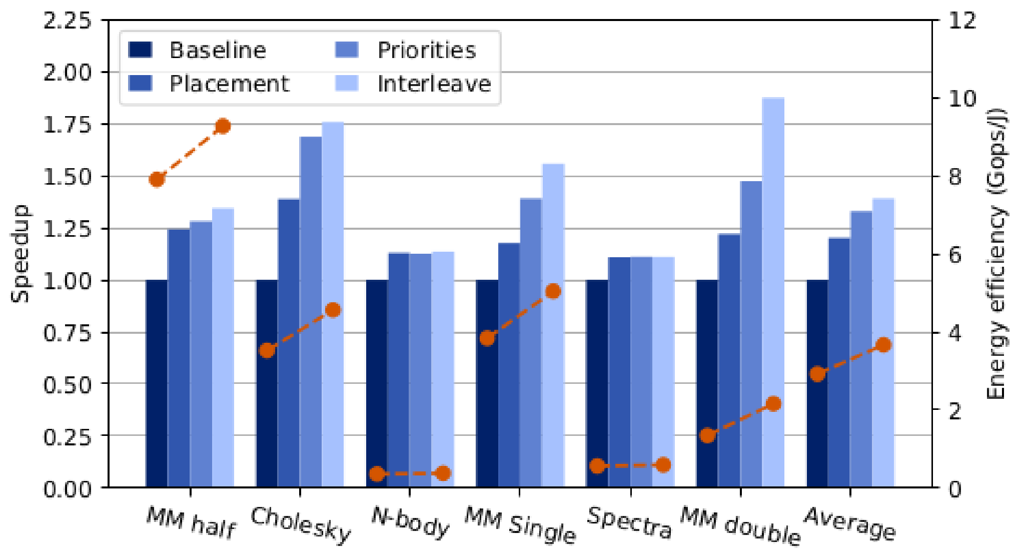


Figure 3.6: Speedup (bars) using different features across different applications. Points and dashed lines show energy efficiency

Figure 3.6 shows speedup compared to the baseline version. This baseline version (labeled as baseline) is the best design we could implement for a given application without using any of the new proposed features. Baseline design configurations and performance are shown in table 3.1 (first of each Baseline/Improved pair).

Application	Num acc.	BS	Par. comp.	Freq. (MHz)	Perf. (Gops/s)	Power (W)	Resource usage (%)				
							BRAM	DSP	FF	LUT	URAM
MM half	8 / 9	256 / 256	128	300 / 350	548 / 761.5 [1]	69.3 / 82.1	39 / 50	53 / 68	26 / 37	32 / 41	59 / 76
Cholesky	4 / 6	256 / 256	128	300 / 333	255 / 448.3 [1]	72.4 / 98.3	54 / 69	47 / 57	32 / 43	37 / 48	34 / 50
N-body	8 / 8	2048 / 2048	16	300 / 333	37 / 42.0 [2]	102.9 / 110.8	66 / 67	75 / 75	40 / 40	55 / 58	0 / 1
MM single	5 / 7	384 / 256	128	300 / 333	353 / 548.6 [1]	92.0 / 108.8	20 / 61	47 / 66	38 / 40	38 / 46	67 / 59
Spectra	10 / 10	2232 / 2232	18	300 / 333	51 / 57.2 [2]	90.5 / 96.8	47 / 48	72 / 72	32 / 35	51 / 57	57 / 57
MM double	6 / 7	128 / 128	64	250 / 300	106 / 226.9 [1]	78.7 / 105.1	40 / 61	36 / 72	31 / 39	43 / 58	18 / 30

Table 3.1: Performance, power, accelerator configuration and resource usage for all analyzed applications. Data is presented as Baseline / Improved when there is a difference between versions. [1]Gflops, [2]Gpair/s

Placement shows the improvement introduced when using placement features. Priorities bars show the improvement obtained by enabling the memory access priorities feature. Finally, interleave shows speedup when enabling interleaving on top of priorities. Dots show energy efficiency for baseline and fully improved versions.

Placement feature impact varies depending on the number of accelerators and their sizes. Implementation tools seem to be able to perform better in designs with a larger number of smaller accelerators. However, all designs benefit from an increased clock frequency. When adding more accelerators contention in memory accesses increases, limiting the overall performance achieved using this feature alone. This can be seen in the matrix multiply case where the ideal speedup from the placement feature should be 1.55x, according to the number of operations performed each cycle but it only achieves a speedup of 1.18x. This is caused by memory accesses limiting performance.

Memory interleaving and priorities improvements seem to be proportional to data type sizes. The wider data types result in a larger amount of data to be moved which results in more pressure in the memory subsystem. Therefore, memory optimizations have a greater impact. This is especially relevant when using double precision matrix multiply. In this particular case, BS is halved, resulting in 8 times less computation but only half the data that each block computation needs compared to single precision. In the case of double precision, using priorities helps, but not as much as interleaving. Even though priorities reduce overlapped copies, there are still a lot of conflicts accessing memory due to the size of the data that needs to be moved for a relatively short computation. This effect is mitigated by interleaving, which increases available bandwidth to accelerators. As commented before, N-body and Spectra applications do not take much advantage of memory improvements because these applications have a low data-to-computation rate. However, as can be seen in table 3.1 the improved memory features do not increase the resource usage of the implementations so they can be used as a default for all applications.

3.4 HBM memory access

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

One of the challenges of the goal is supporting the different memory models used by the different FPGA boards available. Figure 3.4 shows a diagram of the DDR memory interconnection scheme that AIT uses in the OmpSs@FPGA model to enable access to all the available DDR memory. On HBM-based FPGAs, in contrast, there is a single memory module providing access to the entire HBM memory through a series of channels acting like standard DDR interfaces. Each channel is serviced by a single memory controller and can be accessed through two different AXI interfaces. The HBM memory module consists of two stacks of 8 memory channels for a total of 32 pseudo-independent AXI interfaces. In this case, access to the full address space is enabled through the Global Addressing configuration which enables HBM internal micro-switches, meaning that no external crossbar is needed. Nevertheless, a simple 1-to-1 AXI Interconnect IP is used on each AXI interface for data width and protocol conversion, as can be seen in figure 3.7.

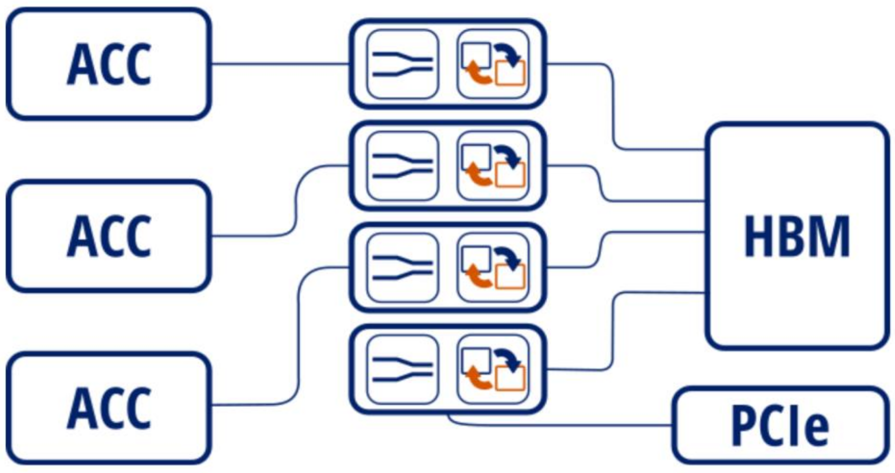


Figure 3.7: Memory interconnection diagram for HBM

All previous datapath optimizations applicable to DDR memories such as wide port access, copies optimization, etc. can be used on HBM memories out of the box.

3.5 OmpSs@cloudFPGA

OmpSs@cloudFPGA is an extension to the OmpS@FPGA framework. Its goal is to enable multi-FPGA execution via message passing, like the Message Passing Interface (MPI), but with the benefits of the task-based programming models. With OmpSs@cloudFPGA, the HLS accelerators can send and receive message messages with the OmpSs MPI for FPGAs (OMPIF) API. Each FPGA participating in the application is assigned a rank, which can be used to determine the source and destination of the messages.

```
void OMPIF_Send(const void* buf, int count, OMPIF_Datatype
               datatype, int dest, int tag, OMPIF_Comm comm);
void OMPIF_Recv(void* buf, int count, OMPIF_Datatype datatype
                , int dest, int tag, OMPIF_Comm comm);
#pragma oss task device(fpga) inout([16]a) in([16]b)
void vecSum(float a[16], float b[16]) {
   int rank = OMPIF_Comm_rank(OMPIF_COMM_WORLD);
   int nrank = OMPIF_Comm_size(OMPIF_COMM_WORLD);
   if (rank != 0) {
      OMPIF_Recv(a, 16, OMPIF_FLOAT, rank-1, 0,
                 OMPIF_COMM_WORLD);
   }
   for (int i = 0; i < 16; ++i)
      a[i] += b[i];
   }
   if (rank != nranks-1) {
       OMPIF_Send(a, 16, OMPIF_FLOAT, rank+1, 0,
                              OMPIF_COMM_WORLD);
        }
    }
```

Listing3.6: OMPIF code example

We can see an example of the OMPIF code in listing 3.6. It is very similar to the MPI equivalent functions, like MPI_Send/MPI_Recv. The main difference with MPI is that OMPIF calls don't return a status code. In the current implementation there are only blocking calls. I.e., OMPIF_Send only returns when the buffer is safe for reading or writing, and OMPIF_Recv returns when the buffer contains the matching message. A match is determined by the source rank and tag of the message. Moreover, the size of a receive operation must match the size of the matching send. Besides simple send and receive, we also implemented the collectives broadcast and allgather.
When the FPGA accelerator calls OMPIF, the runtime creates a task and submits it to the hardware task scheduler. The OMPIF runtime is composed of two modules: the message sender and message receiver, which implement OMPIF_Send and OMPIF_Recv respectively.
Besides OMPIF, we implemented a new type of task in the CPU side. In classic OmpSs@FPGA, all tasks target the only FPGA in the system. However, in the OmpSs@cloudFPGA model we have many devices, so we need a mechanism to send tasks to the cluster nodes. For that, we introduced the distributed task type. This is an FPGA task that can only be created by the CPU. When invoked once by the software program, the software runtime replicates this task for every FPGA in the cluster. Then, every device starts executing the same task, and can use the rank and cluster size to execute different code.

```
#pragma oss task device(fpga) distributed \
inout([n]a) in([n]b)
void vecSumDistributed(float *a, float *b, int n) {
   for (int i = 0; i < n; ++i) {
      vecSum(a + i*16, b + i*16);
      #pragma oss taskwait
   }
}
```

Linsting 3.7: Distributed task code example

We can see an example of a distributed task in listing 3.7. This task is executed by all FPGAs in the cluster, which creates vecSum tasks of listing 3.6 on a variable-length array a and b with blocks of 16 elements. From the CPU side, we can prepare the input data, and send it to the FPGA devices with a specific Nanos6 API. For example, a copy to a specific rank is done with `nanos6_dist_memcpy(int dst, const void* buf, int srcOffset, int dstOffset),` where dst is the FPGA rank, buf is a mapped memory pointer, and srcOffset and dstOffset are the corresponding offsets on CPU and FPGA memories. The memory pointer is mapped with the `nanos6_dist_map_address(const void* buf, int size)`, where

the size is in bytes. This function allocates memory on all FPGAs in the cluster, so the user doesn't have to call the allocate function manually.
The architecture and integration of the OMPIF runtime in OmpSs@cloudFPGA is detailed in deliverable 4.7. We also show the implementation of the programming model in the IBM cloudFPGA and BSC MEEP FPGA clusters.

3.5.1 Benchmarks

To test our programming model, we have implemented three different applications, with very different dependence and communication patterns. For this evaluation we integrated the Picos dependence manager [OMPSSFPGA] with the Fast Task Scheduler IP, since we needed support to create tasks with dependencies in HLS accelerators. Picos is a hardware extension to the task scheduler that adds dependence analysis for FPGA created tasks.

3.5.1.1 Nbody

The Nbody application is a simulator of the interaction of celestial bodies attracted by gravitational forces. The input is a set of particles with initial positions, velocities, and masses. Both positions and velocities are represented by 3-dimensional vectors. Data is represented with single precision. During the simulation, two steps are repeated iteratively. The first one is the calculation and accumulation of the forces of each particle against each other. This part is the most computationally expensive, because the number of forces grows with the formula n^2 where *n* is the number of particles. Forces are calculated using Newton's gravitational law:

$$F_{ij} = \frac{G \times m_i \times m_j \times (p_j - p_i)}{\parallel p_j - p_i \parallel^3}$$

Where F_{ij} is a vector with the forces between particles *i* and *j*, m_i is the mass of particle *i*, p_i is a vector with the position of particle *i* and *G* is the gravitational constant. The second step involves updating the particle's position and velocity according to the force calculated.

To parallelize this benchmark on a cluster, we assign a subset of the force calculation tasks to each rank proportionally. Then, before doing the update part, we do an all-gather collective to send the forces calculated by each rank to every other rank in the cluster. Finally, every rank updates positions and velocities of all particles. We do this because forces are stored consecutively in memory, while positions are stored in a different structure with padding between them. Even though we are doing more calculations in the update part, this doesn't affect performance significantly, as we can see in the results section (3.3.5).

3.5.1.2 Heat

The heat benchmark simulates the propagation of heat over a two-dimensional surface.
Propagation is calculated on each iteration *k* with the following formula:

$$A_{i,j}^{k+1} = \frac{A_{i+1,j}^{k} + A_{i-1,j}^{k+1} + A_{i,j+1}^{k} + A_{i,j-1}^{k+1}}{4}$$

Where A is a matrix of dimensions $n \times m$, with extra rows and columns on the boundaries to avoid reading positions out of bounds. Thus, the real dimensions of the matrix are, $(n + 2) \times (m + 2)$ and the indexes of the formula have the range $i \in [1, n]$ and $j \in [1, m]$.
We use a Gauss-Seidel approach, where A is used both as input and output. When the equation is executed sequentially, the top and left neighbors have the updated values from the current

iteration, while the bottom and right positions have the values from the previous iteration. The initial heat sources are placed on the boundaries, and the rest of the matrix is initialized to 0. We use double precision for this benchmark.

Although the formula is simple and there is only one kernel, the main challenge to parallelize this benchmark is to respect the top and left dependencies. First, we split the matrix into square blocks of a fixed size b, and we create tasks that execute the kernel for a single block. With this algorithm, each task has a dependence on the top and left blocks, except for the blocks that are on the first row and column. This type of dependence graph creates a wave-like pattern, where on each step an anti-diagonal of the matrix can be executed in parallel. The final step is to distribute the work among many nodes and decide on a communication strategy. The simplest distribution is to split the matrix by rows on each node. Each rank has assigned $n \times ranks$ consecutive rows, which are at the same time divided into $b \times b$ blocks.

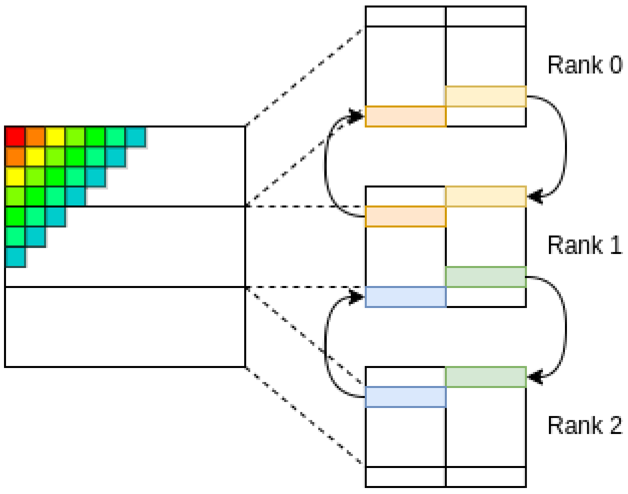


Figure 3.8: Heat wavefront on the left, matrix distribution and communication strategy on the right

Figure 3.8 shows this distribution on the right part. The matrix allocated on each node has one extra row for each block on the top, and one extra row for each block on the bottom. In the case of the figure, there is only one row of blocks, so each one has two additional rows. These are used to store data from neighbor ranks (halo points), except for the first and last, which contain the extra rows of the original matrix with the initial data. The figure also shows the data movements on each iteration of the simulation. First, each node with rank i except the last needs the first row from the neighbor $i + 1$ on the bottom extra row. This communication is performed at the block level, so as long as one block has the needed row, the computation task can start, producing a wavefront execution. This type of execution is illustrated in figure 3.7 on the left at the matrix element level. We can see anti-diagonals with different colors, which designate matrix positions that can be computed in parallel.

Except the first rank, every rank i needs the last row from the neighbor $i - 1$ on the top extra row before the computation task can start. In summary, on each iteration, every rank except the first sends its top row in parallel to the previous neighbor. Then, the first rank starts computing every block and sends the bottom row with the updated data to the next neighbor. This process is repeated for every rank except for the last one. Communication and computation tasks between iterations can overlap if dependencies do not collide.

3.5.1.3 Cholesky

The Cholesky benchmark implements a Cholesky decomposition of a symmetric positive-definite matrix in single precision, which solves the equation:

$$A = U^T U$$

Where *A* is a square matrix of size *n*, and *U* is the output which is an upper triangular matrix. The task-based implementation of this benchmark has four kernels that work on square blocks of *A* with a fixed size: *potrf*, *trsm*, *gemm* and *syrk*. Except *potrf*, they are part of the Basic Linear Algebra Subprograms (BLAS) specification. The *potrf* kernel is a Cholesky decomposition of a single block. To better understand the algorithm, we show the main code below:

```
void cholesky_solver(int n, int b, int nb,
                     float *Ab[nb][nb]) {
for (int k = 0; k < nb; ++k) {
    #pragma oss task inout(Ab[k][k])
    potrf(Ab[k][k]);
    for (int i = k+1; i < nb; ++i)
        #pragma oss task in(Ab[k][k]) inout(Ab[k][i])
        trsm(Ab[k][k], Ab[k][i]);
    for (int i = k+1; i < nb; ++i) {
        for (int j = k+1; j < i; ++j)
            #pragma oss task in([Ab[k][I]) in([Ab[k][j]) \
                inout(Ab[j][i])
            gemm(Ab[k][i], Ab[k][j], Ab[j][i]);
        #pragma oss task in(Ab[k][i]) \
            inout(Ab[k][k])
        syrk(Ab[k][i], Ab[k][k]); } } }
```

Listing 3.8: Code of the Cholesky factorization with tasks

In the code of listing 3.8, the input matrix is divided into separate square blocks in memory. The input Ab is a matrix of pointers to each block. Variable *nb* indicates the number of blocks in each dimension of *A*, and *b* is the size of a block.

A straightforward strategy to distribute the algorithm of figure 3.8 on a cluster is to assign rows of blocks to ranks. I.e. tasks are executed on the node that has the output row assigned to it. We do a cyclic distribution of block rows because the critical kernel of the benchmark is the *gemm*. The amount of *gemm* tasks created grows with a factor of $\approx (n/b)^3$, while *syrk* and *trsm* with a factor of $\approx (n/b)^2$ and potrf with n/b . All tasks created on the innermost *j* loop are parallel and have output blocks from different rows. It is critical to distribute them in different nodes to avoid bottlenecks. However, if we do this distribution, the *potrf* and *trsm* are executed by a single node. After these tasks are finished, the corresponding node broadcasts the row of blocks *k* to the rest of the cluster. The *gemm* and *syrk* tasks can be executed without barriers before moving to the next iteration of the *k* loop.

Although this implementation is easy to write, it doesn't give the best performance as seen in section 3.3.5. Instead, we propose an alternative implementation that requires more changes to the original code. Instead of distributing rows of blocks, we do a cyclic distribution on both rows and columns. This way, both *trsm* and *gemm* tasks can be executed in parallel by different ranks. However, instead of a single broadcast, we have to do sends and receive with blocks, depending on the dependencies of each task. I.e. if a task has a dependence on a block that is stored on another node, we do a send operation in that node, and a corresponding receive in the destination node.

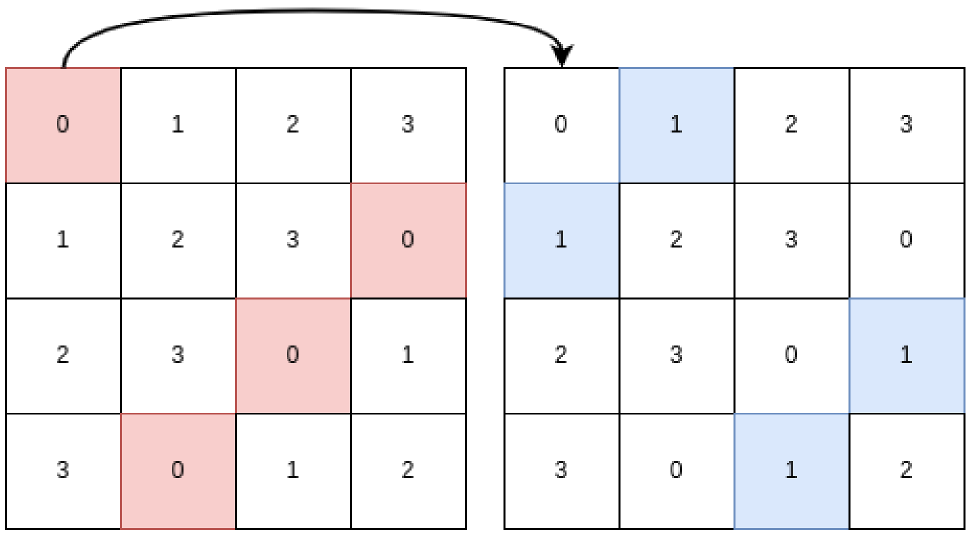


Figure 3.9: Cyclic block distribution of a Cholesky matrix and data transfer from rank 0 to rank 1

We show an example of the distribution in figure 3.9 with a matrix of 4 blocks in a cluster of 4 nodes. Blocks belonging to ranks 0 and 1 are highlighted with different colors. The figure also shows the first data movement on the first *trsm* task, because it is executed by rank 1 but needs a block that belongs to rank 0.

3.5.2 Results

3.5.2.1 Experimental setup

We evaluated the mentioned benchmarks of section 3.3.4 in the cloudFPGA cluster. The MEEP cluster implementation is still under development and there are no performance results.

In cloudFPGA, we use 56 Xilinx Kintex UltraScale xcku060-ffva1156-2-i FPGAs, with 20nm TSMC technology. The boards include 2 DDR4 modules of 8GB RAM each (16GB total). FPGAs have no CPU host attached, instead, all devices are connected to the same network. The cluster uses UDP/IP via two levels of 10G ethernet switches. In our designs, we only use one of the DDR modules to improve timing constraint satisfaction. We realized that most of the time the critical path is on the memory controllers, which has a negative slack. So, we removed one to get positive slack on all paths.

We also evaluated the benchmarks with the MareNostrum4 (MN4) supercomputer at BSC. This way we can compare our model with a regular CPU cluster using MPI. MN4 has 3456 nodes with two Intel Xeon Platinum 8160 CPUs, 94GB of DDR4 RAM each, and 14nm Intel technology. It also features a 100Gb Intel Omni-Path Full-Fat Tree network.

However, first we focus on the FPGA implementation and performance of the benchmarks on sections 3.3.5.2, 3.3.5.3 and 3.3.5.4. Then, on section 3.3.5.5 we explain the implementation and performance of all benchmarks in MN4, and a comparison with cloudFPGA.

3.5.2.2 Nbody

In all our tests we use a block size of 2048. For cloudFPGA, we can fit 4 force accumulation and one particle update accelerators in the bitstream at 200MHz clock for each node. The force accumulation kernel consists mainly of a pipelined loop with an Initiation Interval (II) of 1, and an unroll factor of 8. This means the kernel computes 8 forces per cycle per accelerator (32 forces per FPGA). In our experiments, each Stratix board has 8 force accumulation accelerators that compute 32 forces in parallel with II 1 (256 forces per FPGA). The particle update loop is pipelined with a factor of 7. It is limited by the port number of the particle and force memories. I.e., there are not enough parallel ports to do all loads and stores in the same cycle.

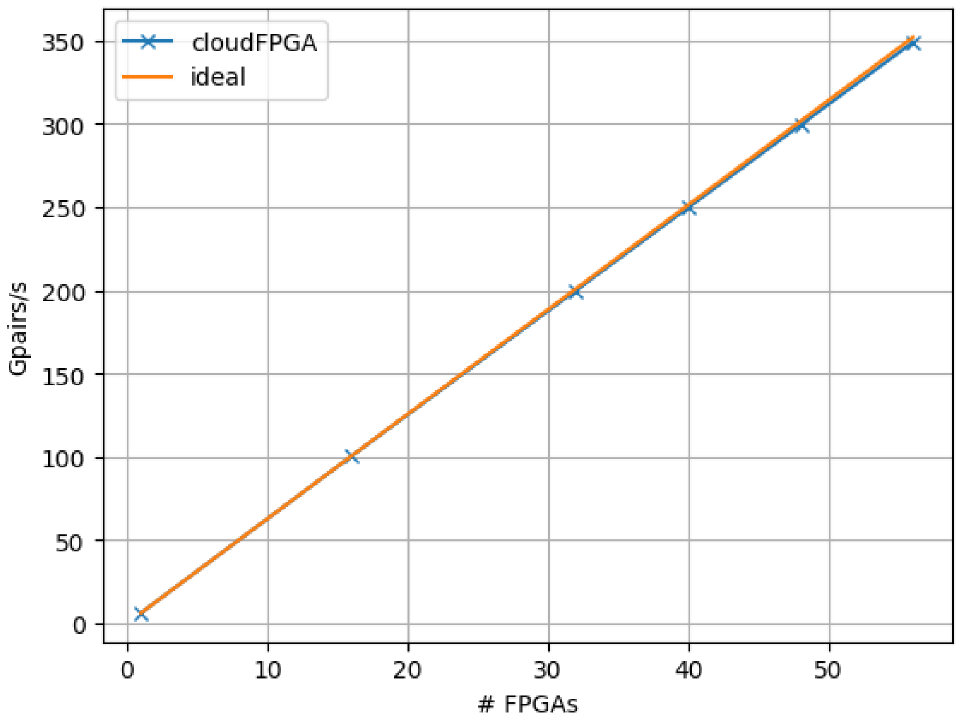


Figure 3.10: Nbody performance scalability with ~3.4M particles

With the aforementioned configuration, we run the application from 1 to 56 FPGAs. The performance results are in figure 3.10, reported as the number of forces calculated per second. The performance line is overlapping with the ideal line, because the efficiency of this application is near perfect, being 99% with 56 FPGAs.

3.5.2.3 Heat

We use a block size of 128×128 and fit 5 accelerators in the bitstream of each FPGA at 200MHz. We pipeline the main loop with II 1. To do that, the kernel visits each position of a block with an anti-diagonal order to remove any dependence in consecutive iterations. However, there is still a dependence between positions at different anti-diagonals. We can compute both with II 1 if the latency of the loop is greater than the distance between two consecutive anti-diagonals. I.e. if the latency of the loop is *l* and we are at iteration *t* on anti-diagonal *d*, any element of *d* will be read by iteration $t + d$, expecting an updated value. With II 1, a new iteration starts on each cycle, but the output of iteration *t* is calculated after *l* cycles. Therefore, we avoid the dependence if $t + l < t + d$, because by the time the kernel reads the block array, it will be already updated. The value of *l* is reported by the HLS tool itself after compilation but depends on many factors like the target frequency.

With this method, we split the main loop into three. First, to calculate all anti-diagonals that are smaller than *l* with II 1 only for a single anti-diagonal. Then, to calculate all anti-diagonals greater than *l* with II 1, overlapping anti-diagonals. Finally, one last loop to calculate the trailing anti-diagonals smaller than *l* like the first one.

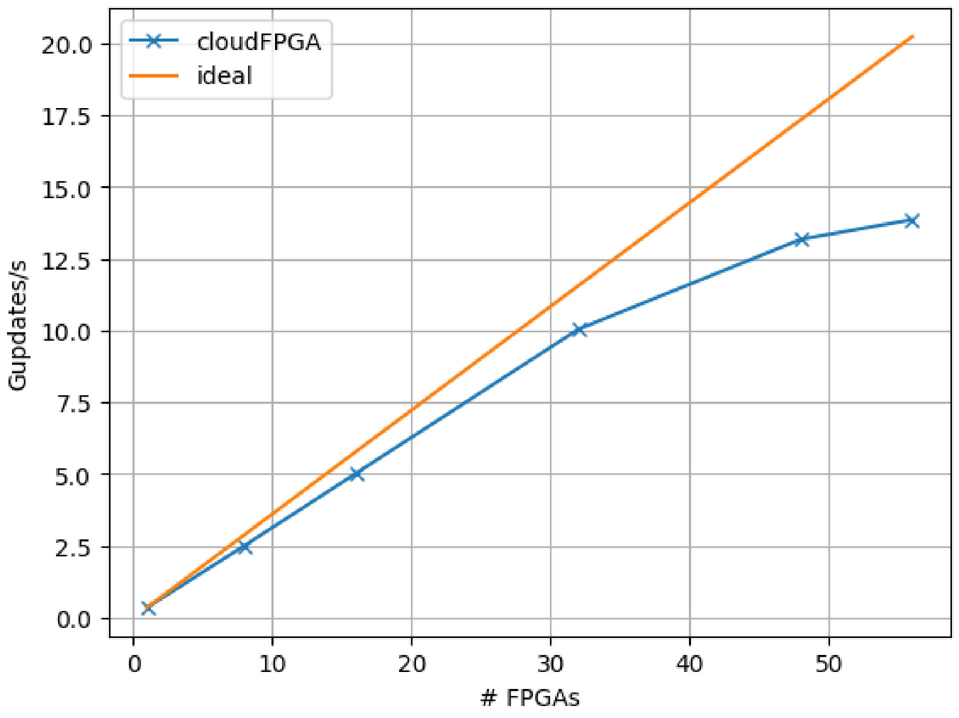


Figure 3.10: Heat performance scalability with matrix size 35K×25K

We have the results in figure 3.10, performance is reported as number of matrix updates per second. Scalability is not as perfect as with Nbody because network latency affects negatively this benchmark. First, every FPGA has to wait for the neighbor rank before it can start computing. Second, the amount of computation tasks is linear to the matrix size, while in Nbody is quadratic, thus the percentage of time spent in communication is bigger for Heat.

3.5.2.4 Cholesky

We could add 1 accelerator of each type with 128×128 block size at 200Mhz. The HLS code of gemm, syrk and trsm calculate a block row every two cycles. We could calculate a row per cycle by increasing the II, but this increases proportionally the resource usage. On the other hand, the potrf kernel has more dependencies on internal loops, so we cannot pipeline or unroll it efficiently without transforming significantly the original code. This is why it is the slowest kernel, nevertheless, it is not the most critical because the total execution time of potrf is very small compared to the others.

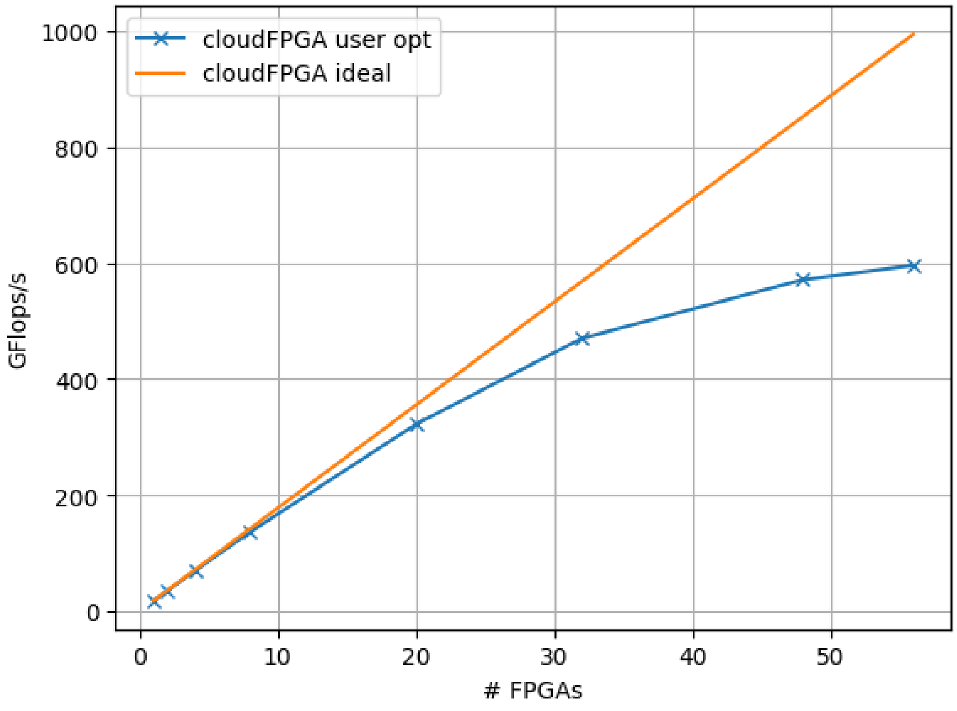


Figure 3.11: Cholesky performance scalability with matrix size 43008

Figure 3.11 shows the performance of Cholesky from 1 to 56 FPGAs as floating-point operations per Second (Flops/s). This application shows even less scalability than Heat, reaching 60% efficiency with 56 nodes. We theorize this is caused by two factors. First, like in Heat, network latency is limiting performance, because with more FPGAs and bigger matrices, there are more tasks that have communication. Second, the matrix size is not big enough to achieve maximum performance. In our proof-of-concept implementation, all nodes allocate the full matrix, so we are limited by the memory available in a single node. We use one DDR module of 8GB, with 1GB reserved for the intermediate buffer explained in deliverable 4.7, so we have 7GB available. This is why we only test with a matrix size of 43008. The average parallel blocks per node is 6, which might not be enough to stress the system.

3.5.2.5 CPU cluster implementation and comparison

For all benchmarks we use the same task parallelization as the FPGA implementations with Intel MPI instead of OMPIF. The main changes are in the kernel optimizations.

In N-body, we use Intel AVX-512 intrinsics to vectorize the force accumulation kernel loop, with a block size of 512. The Heat benchmark kernel uses the original sequential code for a single block of size 512×512 . We also use MPI to send and receive 8 rows in parallel. In Cholesky, we use the Intel Math Kernel Library (MKL) implementation of the kernels with a block size of 512×512 .

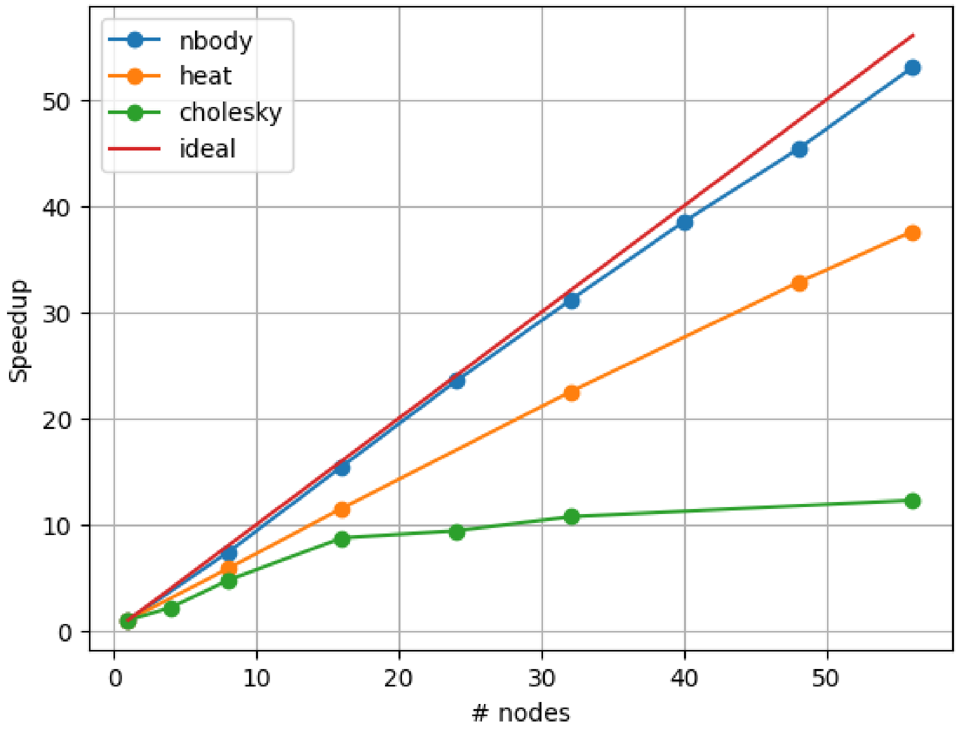


Figure 3.12: Benchmarks normalized performance of MareNostrum4 cluster

Results are summarized in figure 3.12, where we show performance as speedup to fit every benchmark in a single plot. Like in cloudFPGA, we test from 1 to 56 nodes. Problem sizes used are: N-body uses 6M particles, Heat 65K rows per node and 131K columns (weak-scaling), and Cholesky 143360 matrix size. For this figure we focus on scalability. We see similar effects as with cloudFPGA. Nbody scales almost perfectly, and Heat and Cholesky show degradation in performance. With MN4, this effect is even greater. We believe the causes are similar. First, communication time has bigger effects on Heat and Cholesky due to dependencies with computation tasks, and the amount of communication tasks grow proportionally to computation tasks. Second, in Cholesky we have the same memory limitation. With 56 nodes and matrix size of 134460, there is an average of 5 blocks per node, but each node has 48 cores. Also, the threads used for communication block a core until the operation is finished. This implies that receive operations waiting for a matching message prevents the cores from doing useful work. This limitation could be solved by modifying the runtime, and pausing the thread while there are no messages.

Cluster	Nbody			Heat			Cholesky		
	Gp/s	Pf/W	Eff.	Gup/s	Pf/W	Eff.	GFlops/s	Pf/W	Eff.
cloudFPGA	348	0.374	99%	12.9	0.020	70%	596	1.12	60%

MN4	356	0.154	94%	208	0.012	67%	19299	1.61	22%

Table 3.2: Performance, performance per power and efficiency of all benchmarks

We summarize all results in table 3.2, where we put together the performance, performance per power and efficiency of cloudFPGA and MN4. Although absolute performance is lower in cloudFPGA for all benchmarks, this one has better scalability and energy efficiency. Cholesky is the only one that has less performance per Watt, but the scalability is 2.7x higher. To compare both clusters, we also must consider the technology gap, since Kintex UltraScale FPGAs have older technology (20nm vs 14nm). Despite this, we managed to improve the performance per Watt and scalability.

Another fact to consider is the problem's size. We can observe that every CPU benchmark uses bigger sizes in all benchmarks. For example, with 56 nodes in N-body we use double the number of particles, in Heat the matrix is 519 times bigger, and in Cholesky 12 times. This means that FPGAs reach peak performance with much smaller inputs than the CPUs, especially in Heat. There are several reasons for that, like the runtime overheads, the number of computing units and their speed. For FPGAs, we demonstrate that FPGAs have significantly less overhead in section 3.3.3.6. Regarding the number of computing units, MN4 nodes have 48, while FPGA nodes have between 5 and 8. Regarding the computing unit speed, it also depends on the benchmark. For example, in Nbody, cloudFPGA accelerators' performance is ~1.5Gpairs/s, while in MN4 it is ~0.13. However, this is not the case for Heat and Cholesky.

In summary, we demonstrated that FPGA clusters can help improve energy consumption while maintaining a useful performance in HPC applications. Furthermore, with OmpSs@cloudFPGA we can efficiently and easily program these clusters in a similar way like in classic CPU clusters. We believe this can help to introduce HPC programmers to FPGAs.

3.5.2.6 Runtime overheads

We have measured the overheads of the task scheduling and message passing runtimes for cloudFPGA and MN4. The results are in table 3.3. For the task scheduling, we measure the average time to process a task for every benchmark. To do that, we execute the application removing the computation code. I.e., the code creates tasks that have almost 0 execution time. This way we are only measuring the time to execute the code creation code and the runtime task processing.

Cluster	Nbody	Heat	Cholesky	Network	
	ns/task	ns/task	ns/task	Badnwidth (MB/s)	Latency (us)
cloudFPGA	310	629	441	300	13.12
MN4	2008	1844	9798	3512	110.00

Table 3.3: Average task processing time for every benchmark and network bandwidth and latency

For the network overheads, we run a microbenchmark between two nodes. Rank 0 sends an arbitrary number of bytes to rank 1 in chunks of a fixed size. To do that, we use multiple calls

to OMPIF_Send/Recv or MPI_Send/Recv. With this benchmark, we measure the bandwidth of the network and the overheads of the message passing runtime.
The latency is measured using the ping command on MN4, while on cloudFPGA we use hardware counters in the message sender IP.

In task scheduling, the software runtime in MN4 is 6.47x, 2.93x and 22.21x slower than the hardware runtime of cloudFPGA. The main reason is that task creation and processing is optimized in hardware with local memories, so creating a task takes only a few cycles, while a memory load in a CPU core may take hundreds. The difference between every benchmark is due to the HLS code implementation, which has a significant effect. For example, in Nbody the time per task is much lower because the task creation loop is pipelined with II 1, while on the rest it is not.

MN4 has a 100Gb network while cloudFPGA is 10Gb, so it is expected to find a ~10x difference in bandwidth. However, we see that both are far from the peak bandwidth, due to the protocol layers and the message passing runtime overheads. MN4 reaches 28% of the peak, while cloudFPGA reaches 24%. This means that our runtime is not optimized as well as the Intel MPI runtime, and that we can improve significantly the bandwidth.

The latency of the cloudFPGA cluster is much lower than MN4. There are two main factors that affect latency, the distance between nodes and the runtimes. Although not optimized for bandwidth, latency added by the FPGA runtime is on the order of tens of cycles, while on a CPU the operating system adds a significant latency. Moreover, MN4 cluster is much bigger so it is expected that a message has to travel significantly more distance than in cloudFPGA.

4 Conclusions

In this deliverable we have shown the work done in the task-based runtime models (OmpSs and StarPU) in the Textarossa project.

As explained in this deliverable and deliverables 4.7 HLS flow and 4.8 Framework for efficient CNNs inference on a TEXTAROSSA node, the additions to the OmpSs@FPGA framework done in the context of the project have been significant. The whole framework has been upgraded to use the new version of the programming model OmpSs-2. Also, the framework compiler support has changed from the old Mercurium compiler to a new fork of llvm compiler. Finally, the framework has been made compatible with Vitis HLS system, the last version provided by the FPGA vendor.

The new implementation supports all the features present in the previous version of the framework and also adds new improved characteristics that result in better performance and programmability. Support for mixed precision data, new C++ features, and better usage of the IDV-E resources are among the improvements described to name a few.

Finally, a new system to program large, distributed clusters of FPGAs is described and evaluated. This new feature is to the best of our knowledge the most advanced runtime system to perform this kind of work. The capability to execute complex computational problems in FPGA (like NBody or Cholesky problems) with performance competitive with state-of-the-art HDL designs, as reported, proves its value.

5 Future Work

As a future work we plan to continue developing the systems described in this deliverable. Apart from maintaining the tools for others to use we plan to address a wider audience of users by incorporating our ideas into open standards like OpenMP.

We really believe that the distributed tasks idea presented in this deliverable will be helpful to program large clusters of FPGAs. Porting this solution from the IDV-E platform developed in the project to other commonly available platforms (from the same or different vendors) and evaluating its capabilities is a necessary next step. In addition to finalizing the publication of the journal paper already submitted for review, we need to prepare at least one more paper centered in the distributed task idea.

Finally, we plan to integrate all the work presented here with some of the project applications and further evaluate them in IDV-E. We expect to improve performance even further while maintaining programmability as presented here.

References

[AMBAAXI] AMBA AXI and ACE Protocol Specification Version E, ARM. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e

[BOSCH2020] J. Bosch, M. Vidal, A. Filgueras, C. Álvarez, D. Jiménez-González, X. Martorell, and E. Ayguadé, "Breaking master-slave model between host and fpgas," in Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 419–420. [Online]. Available: https://doi.org/10.1145/3332466.3374545

[BREAKMSM] Jaume Bosch, Miquel Vidal, Antonio Filgueras, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé: Breaking master-slave model between host and FPGAs. PPoPP 2020: 419-420

[Cattaneo2021] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, Giovanni Agosta: Architecture-aware Precision Tuning with Multiple Number Representation Systems. 2021 58th ACM/IEEE Design Automation Conference (DAC), https://doi.org/10.1109/DAC18074.2021.9586303

[CHAMELEON] Agullo, Emmanuel and Augonnet, Cédric and Dongarra, Jack and Ltaief, Hatem and Namyst, Raymond and Thibault, Samuel and Tomov, Stanimire, Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, GPU Computing Gems.

[Cherubin2020] Stefano Cherubin, Giovanni Agosta: Tools for Reduced Precision Computation: a Survey. ACM Computing Surveys, Volume 53, Issue 2, April 2020. https://doi.org/10.1145/3381039

[DAN21] Marco Danelutto,, Gabriele Mencagli, Massimo Torquati, Horacio González-Vélez, Peter Kilpatrick: Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. Int. J. Parallel Program. 49(2): 177-198 (2021)

[DS19] Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, Marco Aldinucci, Marco Danelutto: Power-aware pipelining with automatic concurrency control. Concurr. Comput. Pract. Exp. 31(5) (2019)

[EUROEXA] EuroEXA project. https://euroexa.eu/. Last retrieved, september 2022.

[ExaGeoStat] Abdulah, Sameh, et al. "Exageostat: A high performance unified software for geostatistics on manycore systems." IEEE Transactions on Parallel and Distributed Systems 29.12 (2018): 2771-2784.

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

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

[FILFPT] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell: FPGA framework improvements for HPC applications. FPT 2023, accepted, to be published in December 2023.

[FINE2021] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, “Transformations of high-level synthesis codes for high-performance computing,” IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 5, pp. 1014–1029, 2021.

[FLUSEPA] Jean Marie Couteyen Carpaye, Jean Roman, Pierre Brenner. Design and Analysis of a Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. International Journal of Computational Science and Engineering, Inderscience, 2017.

[Fossati2020] Nicola Fossati, Daniele Cattaneo, Michele Chiari, Stefano Cherubin, Giovanni Agosta: Automated Precision Tuning in Activity Classification Systems: A Case Study. Proceedings of PARMA-DITAM 2020, January 2020. https://doi.org/10.1145/3381427.3381432

[FPGAOMPSS] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta: Application Acceleration on FPGAs with OmpSs@FPGA. FPT 2018: 70-77

[GONZ2022] C. Gonzalez, S. Balocco, J. Bosch, J. M. de Haro, M. Paolini, A. Filgueras, C. lvarez, and R. Pons, “High performance computing pp-distance algorithms to generate x-ray spectra from 3d models,” International Journal of Molecular Sciences, vol. 23, no. 19, 2022. [Online]. Available: https://www.mdpi.com/1422-0067/23/19/11408

[HETEROPRIO] Task-based Fmm for heterogeneous Architectures, Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, Toru Takahashi, Concurrency and Computation: Practice and Experience, Volume 28, Issue 9, 25 June 2016, Pages 2608-2629

[Lof21] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Alves de Araujo, Massimo Torquati, Marco Danelutto, Luiz Gustavo Fernandes: The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. Future Gener. Comput. Syst. 125: 743-757 (2021)

[Magnani2021] Gabriele Magnani, Daniele Cattaneo, Michele Chiari, Giovanni Agosta: The Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control. Proceedings of PARMA-DITAM 2021, January 2021. https://drops.dagstuhl.de/opus/volltexte/2021/13639/

[Magnani2022] Gabriele Magnani, Lev Denisov, Daniele Cattaneo, Giovanni Agosta: Precision Tuning in Parallel Applications. Proceedings of PARMA-DITAM 2022, June 2022.

https://drops.dagstuhl.de/opus/volltexte/2022/16121

[Men21] Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, Marco Danelutto: WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks. IEEE Trans. Parallel Distributed Syst. 32(11): 2748-2763 (2021)

[MORBSCDS] Lucas Morais, Daniel Jimenez-González, Carlos Álvarez: Exploring Task Scheduling Sensitivity to L1 Cache configuration on a 32-core RISC-V Processor. 9th BSC Doctoral Symposium Book of Abstracts. 2022.

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

[OMPSSGUIDE] 2023. OmpSs@FPGA User Guide. https://pm.bsc.es/ftp/ompss/doc/user-guide

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

[Ott21] Alberto Ottimo, FSP: a Framework for Data Stream Processing Applications targeting FPGAs, Master thesis, Computer Science and networking master, University of Pisa, 2021, https://etd.adm.unipi.it/theses/available/etd-09032021-173034/

[PASTIX] Grégoire Pichon, Mathieu Faverge, Pierre Ramet, Jean Roman. Reordering Strategy for Blocking Optimization in Sparse Linear Solvers. SIAM Journal on Matrix Analysis and Applications, Society for Industrial and Applied Mathematics, 2017, SIAM Journal on Matrix Analysis and Applications, 38 (1), pp.226 - 248.

[QMCkl] Scemama, Anthony, and Claudia Filippi. "Software development strategy in the TREX Center of Excellence." CECAM Workshop: The importance of being HPC Earnest. 2020.

[SCALFMM] Task-based Fmm for Multicore Architectures, Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, Toru Takahashi, SIAM Journal on Scientific Computing (SISC), SIAM, 2013, SIAM J. Sci. Comput., 36(1), C66-C93.

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

[XILINX- 2] https://xilinx.github.io/XRT/master/html/xocl_ioctl.main.html

[XILINX- 3] https://xilinx.github.io/XRT/master/html/mgmt-ioctl.main.html

[XILINX- 4] https://xilinx.github.io/XRT/master/html/xrt_native_apis.html

Appendix A. OmpSs framework source code

The source code of the complete OmpSs@FPGA framework can be found in the OmpSs@FPGA public Github page:

https://github.com/bsc-pm-ompss-at-fpga

The most recent implementation of the fast task scheduler is integrated in the OmpSs-2@FPGA release that is the version currently under development (OmpSs-1@FPGA is no longer updated, only bug fixes are applied to it, the OmpSs@FPGA framework and components referred in this deliverable are all related to the new OmpSs-2@FPGA release).

Also, the source code of all hardware modules described in deliverable 2.10, as well as the wrappers that interconnect and instantiate them, are available via BSC's B2Drop platform:

https://b2drop.bsc.es/index.php/s/tbEzqEHegxNXLP6

The source code of StarPU that includes the patch to support FPGAs as OpenCL devices and the Multreeprio scheduler can be found in the public Gitlab page:

https://gitlab.inria.fr/htayeb/starpu-multiprio-scheduler

Appendix B. Published research articles

The results of the research reported in this deliverable have been published in three conference papers:

- [FILBSCDS] Antonio Filgueras, Daniel Jimenez-González, Carlos Álvarez: Improving resource usage in large FPGA accelerators. 9th BSC Doctoral Symposium Book of Abstracts. 2022 (https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/9thbsc-%20so-doctoral-symposium). This article covers the initial implementations of the task-based programming model for PCIe attached large FPGA boards and how to improve the applications resource usage.
- [FILFCCM] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell: Improving Performance of HPC Kernels on FPGAs Using High-Level Resource Management. FCCM 2023: 213 (https://ieeexplore.ieee.org/document/10171527). This article includes how a set of memory and placement improvements to the framework allowed different precission applications to take advantage of multi-SLR FPGAs (as the one used in IDV-E).
- [FILFPT] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell: FPGA framework improvements for HPC applications. FPT 2023, accepted, to be published in December 2023. This last article, already accepted for presentation at the upcoming International Conference on Field-Programmable Technology (FPT'23) evaluates how different applications benefit from the improvements developed in this work package.
- [HAYMUL] Hayfa Tayeb, Bérenger Bramas, Abdou Guermouche and Mathieu Faverge: MulTreePrio: Scheduling task-based applications for heterogeneous computing systems, Compas National Conf. 2022. This article describe the first version of Multreeprio.

Also, a journal research paper titled "Automated parallel execution of distributed task-graphs with FPGA clusters" covering the joined stream and task-based programing models' proposal is under review by the Future Generation Computing Systems journal. In addition, a conference paper titled "Dynamic Tasks Scheduler with Multiple Priority-based Trees on Heterogeneous Computing Systems" providing a more in-depth performance study of Multreeprio is under review at the IPDPS conference.