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

WP2 New accelerator designs exploiting mixed precision

D2.10 IP for fast task scheduling, part 1

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

TEXTAROSSA

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

Grant Agreement No.: 956831

Deliverable: D2.10 IP for fast task scheduling, part 1

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

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

Deliverable No	D2.10
WP No:	WP2
WP Leader:	CINI-POLIMI
Due date:	M18 (September 30, 2022)
Delivery date:	???/??/2022

Dissemination Level:

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

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

DOCUMENT SUMMARY INFORMATION

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

WP 2 New accelerator designs exploiting mixed precision

Deliverable number:	D2.10					
Deliverable title:	IP for fast task scheduling, part 1					
Due date:	M18					
Actual submission date:	M18					
Editor:	Carlos Álvarez					
Authors:	A. Filgueras, M. Vidal, C. Alvarez, D. Jimenez, X. Martorell					
Work package:	WP2					
Dissemination Level:	Public					
No. pages:	18					
Authorized (date):	10/10/2022					
Responsible person:	Carlos Álvarez					
Status:	Plan	Draft	Working	[✓ Final]	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	2022-09-30	A. Filgueras	Draft structure
0.2	2022-10-07	A. Filgueras, D. Jiménez, C. Álvarez, M. Vidal	Writing
0.3	2022-10-10	X. Martorell	Review
0.4	2022-10-12	B. Cantalupo, P. Palazzari	Internal review

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	X. Martorell	2022-10-10
Checked by Task Leader	X. Martorell	2022-10-10
Checked by WP Leader		
Checked by Project Coordinator	Massimo Celino	2022-10-15

COPYRIGHT

ACKNOWLEDGEMENTS

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

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

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

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

DISCLAIMER

Table of contents

List of Figures

List of Tables

List of listings

List of Acronyms

Acronym	Definition
ASIC	Application Specific Integrated Circuit
BP	Business Plan
BRAM	Block Random Access Memory (in-FPGA RAM module)
DSP	Digital Signal Processor
FF	Flip Flop
FPGA	Field Programmable Gate Array
FTS	Fast Task Scheduler
HPC	High-Performance-Computing
IP	Intellectual Property
LUT	Look-Up Table
PCIe	Peripheral Component Interconnect express
SMP	Symmetric Multi-Processor
URAM	Ultra Random Access Memory (in-FPGA large RAM module)

Executive Summary

This document reports on the activities done by Textarossa partner BSC with reference to the design of the Fast Task Scheduler IP in WP2 and preliminary design and synthesis results, mainly in FPGA technology.

1 Introduction

The main objective of developing an IP for fast task scheduling is to provide an effective and efficient way to send tasks to accelerators implemented in the FPGA. The scheduler IP allows to offload the process of scheduling tasks into individual accelerators and keeping track of accelerator status and finished tasks. This reduces the communications and synchronizations between host and FPGA accelerators, increasing overall performance.

This document describes both the high-level design and the inner-most functionality description of such IP. The document is organized as follows:

- Section 2 presents the basic design, with is complemented by the information already available in deliverable 2.1 Consolidated specs of accelerators IPs.
- Section 3 explains some key algorithmic implementation details.
- Section 4 highlights the implementation results in terms of FPGA resource usage and timing capacities of the mechanism.
- Section 5 shows some earlier performance results and
- Section 6 concludes the report.

As a demonstrator deliverable, appendix A shows the source code of the IP developed.

2 Basic design

Basic design, as well as interface with accelerators and host system are described in finer-grain detail in D2.1 Consolidated specs of accelerators IPs. Only an introductory high-level overview is provided in this document.

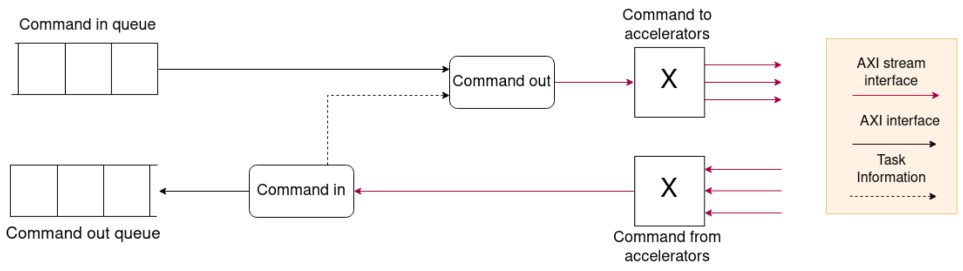

Figure 2.1 IP for fast task scheduling diagram (FTS).

Figure 2.1 shows a diagram of the IP for fast task scheduling (from now on FTS or Fast Task Scheduler). The main objective of the FTS is to take care of scheduling tasks into individual accelerators. To achieve this objective, the FTS IP is composed of two command queues, one for input coming from the CPU/exterior of the FPGA ("Command in queue") and another going to the CPU/exterior of the FPGA ("Command out queue"), two control modules ("Command in" and "Command out") and two interconnection multiplexers/demultiplexers.

The workflow in the FTS follows. First of all, tasks are sent from the host CPU to the Fast Task Scheduler by using commands that are temporarily stored in the "Command in queue". These commands are processed in order by the “Command in” module and, depending on the accelerator's availability, are sent to the appropriate one. Commands are sent through the “Command to accelerators” demultiplexer through an AXI stream interface, and only when accelerators are available (ready) in order to avoid interface contention and starvation.

Once the task has been processed by the corresponding accelerator, the accelerator informs the FTS through an output AXI stream interface that is multiplexed to reach the “Command out” module with a “Finished Task” command. “Finished Task” command is expected to be processed in very few cycles (tens of cycles at most). Therefore, although some contention can be expected when several accelerators finish at the same time submitting this command, no significant performance drop is expected in this case.

Finally, the “Command out” module is in charge of processing the “Finished Task” packet by forwarding it with the adequate format to the “Command out queue” and to notify the “Command in” module about the new ready state of the accelerator in order for the FTS to forward a new task to it.

3 Implementation

3.1 Command in module

Command in module reads commands from the command in queue, and sends them to accelerators if possible.

Module behaviour is described in pseudocode in listing 3.1. The original source code can be found in Appendix A.

```
InOut:
    cmd_in_q: set of #accelerators circular sub-queues
while true:
    foreach acc in accelerators:
        current_slot = getQFront(acc)
        cmd = getCMD(cmd_in_q, current_slot)

        if valid(cmd) && isIdle(acc) && !isRunning(cmd):
            if valid(getCMD(cmd_in_q, current_slot+1))
                optimize_copies(cmd_in_q, current_slot)
            sendCommand(cmd, acc)
            setRunning(cmd)
            setBusy(acc)
        else if valid(cmd) && isIdle(acc) && isRunning(cmd):
            popCMD(cmd_in_q, cmd)
            setFinished(cmd)
```

Listing 3.1: Module Command In basic functionality pseudocode.

As shown in listing 3.1, Cmd_in_q is a circular queue that consists of one sub-queue per accelerator. When one of the sub-queues is not empty (has a valid command) and the accelerator is idle and the valid command is not being run, the command is read and checked for copy optimizations. Copy optimizations consists on avoiding unnecessary copies that are already in the accelerator. Then, the command is sent to the device and the command is marked as running.

Otherwise, if the command in front of the queue is marked as running, but the accelerator running it is idle, it means that the command at the top of the queue has finished and we can pop it out of the queue and update command status as finished.

Another process of the same module, is run concurrently. It listens for messages from command out module and updates internal accelerator status. This is shown in listing 3.2.

```
Input:
    acc: accelerator
Ouput:
    cmd_in_queue: circular queue

setIdle(acc)
```

Listing 3.2: Update availability pseudocode in module Command In

In listing 3.2, the update process of "Command in" works as an interrupt module. When it receives an interrupt from the command out module, it updates the accelerator status.

3.2 Command out module

Command out module listens for finished task messages from accelerators. Upon receiving a message, it sends an accelerator availability update to the command in module and writes the finished task to the finish task queue for the host to read and update task state in the runtime library.

Detailed behaviour is described in listing 3.3. The complete source code of the Command Out module can be found in Appendix A.

```
Input:
    finish_str: finished task stream
Output:
    cmd_out_q: set of #accelerators circular sub-queues

while true:
    acc, taskId = getFinishedTask(finished_str)
    cmdIn_updateAvail(acc)
    push_queue(cmd_out_q, acc, taskId)
```

Listing 3.3: Module Command Out basic functionality pseudocode.

The Command Out module waits for an incoming message from any accelerator, which notifies that a task has finished execution. This is represented as a blocking call to getFinishedTask in listing 3.3. Then, an interrupt is sent to notify that the accelerator is idle again (cmdIn_updateAvail call in listing 3.3), and the taskId of the finished task is pushed into the finished task queue for the runtime system to consume (push_queue call in listing 3.3).

After that, the software part of the runtime system will update internal task status so that progress is made through application execution.

3.3 Copy Optimizer module

The Copy Optimizer module looks for data copies that can be optimized out and reuse data from previous accelerator execution. This module is not shown in the design schematic as it is physically allocated within the Command In module. However, keeping it as a different module improves overall design readability and maintainability. Its complete original source code can also be found in Appendix A.

Copy Optimizer looks for arguments in consecutive tasks that are data copies and point to the same address. If this is the case, it updates the copy flags in order to disable that copy as the version of the data in the accelerator is already valid. Copy flags are part of the task description and are interpreted within the accelerator. Each accelerator includes OmpSs autogenerated hardware that deals with the hardware runtime notifications and data copies, in addition to the programmer FPGA task code. These copy flags are used in this autogenerated hardware to decide if a copy is necessary or not.

As described in listing 3.4, the optimizer module peeks both the task in the queue front and the next one (if it exists). Then, it iterates through the arguments and disables copies if current and next task arguments point to the same address.

In the case of inputs, copies are disabled in the next task if data can be reused according to the arguments of current task. There's a special case for multiple tasks reusing the same input. In this case, a special *chain* flag is set. Otherwise, only disabling input copy, inCopyEnabled test in line 15 of listing 3.4 would fail and data copy will not be disabled even if we can reuse data from previous execution.

For output copies, we disable copies for current task based on information regarding next task. In this case, there's no need to track optimization chains as the reuse can be set task by task.

```
 1  Input:
 2      ready_task_queue: circular queue with
 3          tasks made of the same number of arguments
 4          and copy flags,
 5      current_slot: index of the
 6          current slot in the queue
 7  task_current = ready_task_queue[current_slot];
 8  task_next = ready_task_queue[current_slot+1];
 9  for i = 0; i < #args; i = i+1 do
10      flags_current = task_current.flags[i];
11      flags_next = task_next.flags[i];
12      if task current.args[i] == task_next.args[i] then
13      if outCopyEnabled(flags_next) then
14          suppressOutCopy(flags_current);
15      if inCopyEnabled(flags_current) or
16          chainBitEnabled(flags_current) then
17          if inCopyEnabled(flags_next) then
18              enableChainBit(flags_next);
19          suppressInCopy(flags_next);
```

Listing 3.4: Module Copy Optimizer functionality pseudocode.

It is worth noting that copies can only be optimized when they are done in the same argument. Data storage for each task argument is synthesized as individual hardware resources (usually BRAM or URAM) and is connected to different parts of the accelerator. Therefore, arguments cannot be swapped in any case.

4 Resource usage

Resource usage after implementation is specified in Table 4.1. It shows the number of used resources by the FTS module and submodules as well as the total available resources in the device.

Resource usage in Table 4.1, is shown as a hierarchy. Each module includes all its submodules. *FTS total* contains the sum of all resources used by *Comand_In* and *Command_Out*. *Command_In* includes *Copy Optimizer* module, also detailed in Table 4.1.

	LUT	FF	LUTRAM	BRAM	URAM	DSP
Available (u280)	1303680	2607360	600960	2016	960	9024
FTS total	265	356	0	0	0	0
Command Out	68	176	0	0	0	0
Command In	197	180	0	0	0	0
- Copy Optimizer	135	92	0	0	0	0

Table 4.1: Resource usage by the Fast Task Scheduler IP developed

All resources that the hardware runtime uses are well below 1% of the available resources in the Alveo U280 FPGA. It uses approximately 0.02% of LUT and 0.01% of available FF. Even considering all auxiliary logic needed for the FTS, such as memory resources for task queues and interconnection with the rest of the system, used resources are still below 0.1% of total device resources.

5 Performance

Performance of the FTS for the matrix multiplication is described in detail in D4.1. In this section, we focus in how FTS works and how enabling or disabling FTS features (copy optimizations) affects application execution. Having a Hardware Task Scheduler has already been demonstrated to provide significant performance gains over their Software counterparts [3],[6].

In order to measure performance of the FTS itself, we created a synthetic benchmark that consists of very small (few cycles) independent tasks. We run this benchmark with varying number of accelerators and different clock frequencies, showing the results in Figure 5.1. This figure shows the number of tasks per second that are handled by the system.

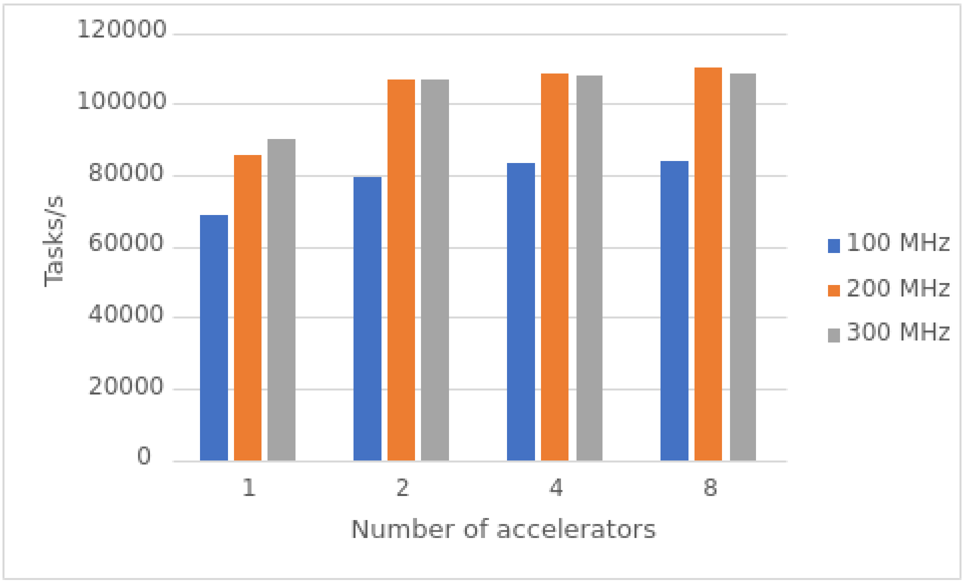

Figure 5.1: FTS performance across different frequencies and number of accelerators

It is important to remark that the FTS is able to process a task in less than 100 cycles, so its estimated maximum throughput is at least one million tasks per second [1],[4] at 100 MHz. As can be seen in figure 5.1, the obtained maximum throughput in the system is around tens of thousands of tasks per second. When running FTS at 100 MHz, the bottleneck seems to be the in-FPGA data storage. Resources in the board other than the PCIe (that always runs at 250MHz) run at the same speed as the accelerators and the FTS. That includes BRAM structures that are in the critical path of the PCIe transactions. Thus, using a frequency that is significantly slower that the PCIe working frequency makes the communication slower than its maximum potential. As can be seen in figure 5.1, as soon as the frequency is closer to the PCIe frequency, the storage frequency is not the problem anymore and the PCIe becomes the bottleneck.

When increasing the number of accelerators to two or more a small improvement in system throughput can also be observed. This is due to the fact that communication with the accelerators is done by independent software (and hardware) queues. Consequently, certain number of operations can be overlapped when sending data to two or more accelerators (like lock acquisition by the software threads, structure creation and filling, etc.). This overlap allows the PCIe to increase its performance and, consequently the system throughput is increased. As can also be observed in figure 5.1, a 200MHz working system and two or more accelerators reach the maximum throughput and, without selecting a different communication mechanism it cannot be improved. The FTS is, consequently deemed fast enough to the system as it can operate orders of magnitude faster than the PCIe communication.

The data copy optimizer module is also something that we have tested from a performance point of view. Figure 5.2 shows three execution traces of an application when data reuse is deactivated or activated. This application has been annotated with FPGA tasks using OmpSs@FPGA and has been cross-compiled for and executed on a Zynq 7000 family board (two Cortex-a9 at 666MHz + FPGA running at 100Mhz) as a proof of concept. This is using two different versions, with and without copy optimizations activated, of the FTS to coordinate the two accelerators (IPs) and the software running on the two cores in the SMP. Horizontal lines in the trace show the states (different colors) on the SMP threads (two lines on the top of each execution trace) and two accelerators (two lines on the bottom of each execution trace), along the time.

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

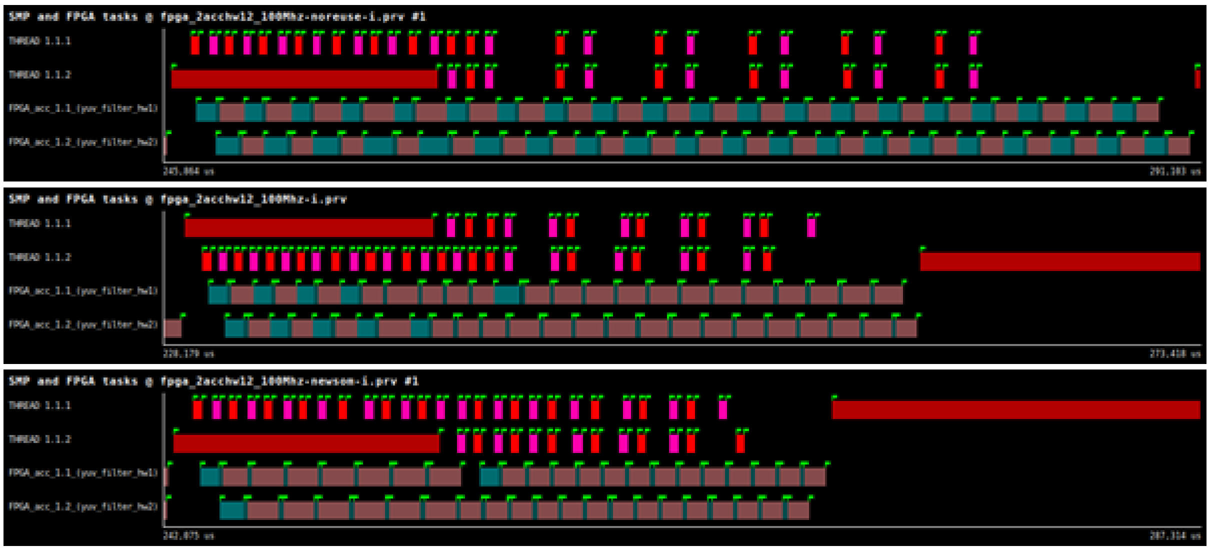
SMP and FPGA tasks @ fpga_2acchw12_100Mhz-noreuse-1.prv #1
THREAD 1.1.1
THREAD 1.1.2
FPGA_acc_1.1_(yuv_filter_hw1)
FPGA_acc_1.2_(yuv_filter_hw2)
243,864 us
293,183 us
SMP and FPGA tasks @ fpga_2acchw12_100Mhz-1.prv
228,179 us
273,418 us
SMP and FPGA tasks @ fpga_2acchw12_100Mhz-newsom-1.prv #1
242,075 us
287,314 us

Figure 5.2: Execution trace of an application using two accelerators. Traces show execution behavior in the same time range. Top: FTS has copy optimizer module deactivated. Middle: FTS has copy optimizer module activated for tasks in the Command in queue. Bottom: FTS has copy optimizer module activated for tasks in the Command In queue and tasks being executed.

On the execution trace on the top of figure 5.2 we can see that there are always three states (different states start and end with flags), which is not ideal. Those tasks are always re-using the same input data but the accelerator is not conscious about this fact and is copying the input data all the time. On the other hand, the execution trace on the bottom shows the performance achieved once FTS includes the data reuse feature. In this case FTS can automatically detect data to be reused in an accelerator and help to almost remove all input copies modifying the argument copy flags of the task descriptions.

Note however that there are still tasks in the execution trace on the middle of Figure 5.2 that have three states and no data reuse is detected. This happens because originally data reuse detection among the tasks is only performed among tasks waiting in the Command In queue and no detection is done between a task being executed and tasks that arrive later to the Command In queue. This situation may happen in several applications: a task is submitted (first one) by the runtime, it immediately starts execution in the accelerator, and then, another task is submitted by the runtime. Since the first one has already started, no detection can be done between Command In queues commands. This can be solved by taking care of the task being executed in the accelerator at that moment. FTS has been improved to detect and be able to catch this situation. This can be seen on the bottom trace of Figure 5.2. The execution trace on the bottom incorporates that feature. Only the first task of all tasks being executed has to copy the data, significantly improving the first FTS version (no data reuse) and allowing first task executing-second task in Command In queue data reuse. The extra-copy seen in the execution trace of the Figure 5.2 (bottom) is because the accelerator was completely empty when a new task was submitted. The overall performance improvement with data reuse can be significant as it can be seen in Figure 5.2. Based on previous research [2] we expect this optimization to deliver significant performance gains.

In the case of application performance when using the FTS, we are going to analyse some details about the Matrix Multiplication algorithm performance related to the FTS hardware design. A more detailed analysis from a more holistic point of view can be found in Deliverable 4.1

Figure 5.3 shows performance when using FTS module described in section 3 implementation and Matrix Multiplication accelerators. As matrix multiplication algorithm is computation bound, performance is limited by accelerator execution time. Host system is able to send tasks to FTS faster than accelerators can consume them. Also, FTS is able to keep accelerators busy, as it takes few cycles to send a task to an accelerator as opposed to the accelerator taking thousands of cycles to compute the task. To highlight the importance of the Copy Optimizer module, two different configurations are showed. In the case of NoOpt, data copies optimization is disabled. In this case, no data is reused between tasks. On the other hand, in CopyOpt bars data copies optimization is enabled. As in can be seen in figure 5.3, this results in better performance results for the same algorithm using the same hardware.

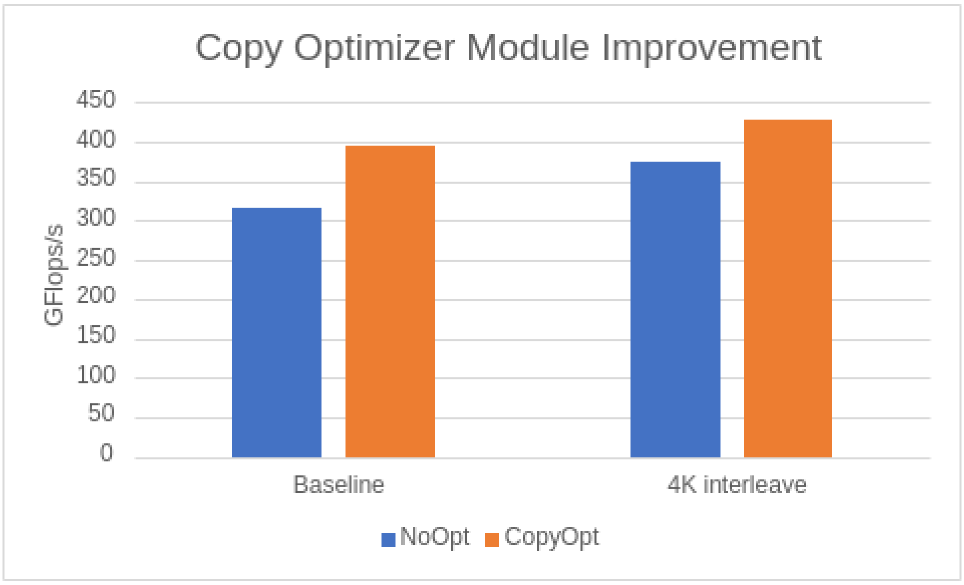

Figure 5.3: Matrix multiplication kernel performance with FTS

In the case of baseline, which does not enable memory access interleaving, memory optimizations have a bigger effect on overall execution while with memory interleaving the effect is diminished. This is because with interleaving (that spreads data over all the different memory banks available), memory accesses are faster and take up less part of total execution time, dampening the effect of this optimization. Even in worst case scenario, enabling this optimization does not have a negative impact on performance, due to its cost being negligible. Checking for data reuse in FTS takes something in the order of 10 cycles per task.

Note that these first results reported here are competitive with what are to the best of our knowledge the current best results reported for a Matrix Multiplication algorithm over an Alveo U200 FPGA [5]. As can be seen in Deliverable 4.1 (section 3.4.2 Performance improvement results), such figures can be further improved with a careful software-hardware co-design [2] between the programming model and the FTS. We are currently preparing a conference paper to publish this project outcome.

6 Conclusions and Future Work

As this deliverable shows, the IP for fast task scheduling is being actively developed and progressing as expected. A first version of the IP has been designed, implemented and tested in the Textarossa IDV-E platform. The design has been integrated with the OmpSs@FPGA task-based programming model in WP 4.2 Task-based Models and its functionality has been verified.

After that, an improved version that saves data transfers between the FPGA accelerators and the FPGA main memory has also been designed, implemented and tested. The results show that the whole IP is fast and has an efficient resource usage in the Textarossa IDV-E platform. Indeed, to the best of our knowledge the IP allows for the current fastest implementation of the Matrix Multiplication reported in the literature. Some preliminary results have been used to publish an early results conference paper [7] and another paper with the later results is under preparation.

Our future work, in addition to finishing the current paper, includes further improving the IP to obtain even more performance out of the system, test different applications and evolve it to support the new features planned in Work Package 4.

7 References

[1] Jaume Bosch, Miquel Vidal, Antonio Filgueras, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé: Task-Based Programming Models for Heterogeneous Recurrent Workloads. In Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2021. Lecture Notes in Computer Science, vol 12700. Springer, Cham. https://doi.org/10.1007/978-3-030-79025-7_8. 2021.

[2] 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)

[3] Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé: Asynchronous runtime with distributed manager for task-based programming models. Parallel Computing, Volume 97, 2020, 102664, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2020.102664.

[4] Jaume Bosch, Miquel Vidal, Antonio Filgueras, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé: PPoPP '20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. February 2020. Pages 419–420. https://doi.org/10.1145/3332466.3374545

[5] Johannes de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler: Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In FPGA '20: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2020. Pages 244–254. https://doi.org/10.1145/3373087.3375296

[6] Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero: A Hardware Runtime for Task-Based Programming Models. In IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 9, pp. 1932-1946, 1 Sept. 2019, doi: 10.1109/TPDS.2019.2907493.

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

Appendix A. Hardware modules source code

Source code of all hardware modules described in this document, 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