Bachelor Degree in Informatics Engineering

Degree final project

Development of a benchmark suite for large vector architectures into a Continuous Integration workflow

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Author:	Rafel Albert Bros Esqueu
Director:	Filippo Mantovani
Co-director:	Fabio Banchelli
Company:	Barcelona Supercomputing Center
Tutor:	Eduard Ayguadé
Department:	AC
Specialization:	Computer engineering
Defense date:	June 28, 2022

Abstract

In the High-Performance Computing world, the processor is essential. In recent years, Europe has devoted a lot of effort into promoting European technology. The European Processor Initiative stems from this effort. As part of the initiative, multiple processors are being developed. Some implementing the RISC-V architecture, an open-source ISA. During the development of a processor, tools are fundamental to ease testing and automatize tasks. This final degree project focuses on improving a Continuos Integration pipeline used to detect bugs in an Field Programmable Gate Array (FPGA) and Linux environments emulating final user behaviour.

Resum

En el món del *High-Performance Computing*, el processador és essencial. Recentment, Europa està fent grans esforços en promoure tecnologia europea. La *European Processor Initiative* sorgeix d'aquest esforç. Com a part de la iniciativa, múltiples processadors estan sent dissenyats. Alguns implementant l'arquitectura RISC-V, una ISA *open-source*. Al llarg del desenvolupament del processador, disposar d'eines és fonamental per facilitar el testeig i automatitzar tasques. Aquest treball final de grau es focalitza en millorar una *pipeline* de *Continuous integration* emprada per detectar errors en un entorn Linux i en una *Field Programmable Gate Array* emulant un comportament d'usuari final.

Resumen

En el mundo del *High-Performance Computing*, el procesador es esencial. Recientemente, Europa está haciendo grandes esfuerzos en promover tecnologia europea. La *European Processor Initiative* emerge de este esfuerzo. Como parte de la iniciativa, múltiples procesadores estan siendo diseñados. Algunos implementando la arquitectura RISC-V, una ISA *open-source*. A lo largo del desarollo del procesador, disponer de herramientas es fundamental para facilitar el *testing* y automatizar tascas. Este trabajo final de grado se focaliza en mejorar una *pipeline* de *Continuous integration* utilizada para detectar errores en un entorno Linux y en una *Field Programmable Gate Array* emulando un comportamiento de usuario final.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In the context of a large-scale project with multiple developers working on different elements simultaneously, version control software is crucial to keep track of any change in the source code. Furthermore, as every modification of the project's code comes at a risk, a Continuous Integration system is critical to automatize the code evolution and its constant testing.

The European Processor Initiative (EPI) is a European project involving more than 30 partners from industry and academia. The main goal is to push European technological independence developing hardware and software that can foster High-Performance Computing (HPC) and automotive markets. The development of EPAC, a RISC-V based accelerator targeting HPC, is one of the goals of the project and the Barcelona Supercomputing Center contributes to it.

The design and implementation of a chip is one of the complex tasks where one (or multiple) code control services are required. The work presented in this document has been developed at the Barcelona Supercomputer Center in a group whose job is to create a software and hardware environment to test the EPAC processor and experiment with it. The main contributions of this final degree project are:

C1: The creation of a benchmark suite that tests individual features of the design hosted in the EPI project repository, and

C2: The integration in the EPI version control infrastructure of a set of tools and configurations for detecting modifications able to introduce errors in the design.

Both contributions resulting from this final degree project have been deployed in the Continuous Integration pipeline of the EPAC repository and have been adopted as solutions to detect erroneous modifications to the EPAC design preventing that such modifications enter the repository.

The following work is a Final Degree Project of Computer's Engineering specialty within the *Grau en Enginyeria Informàtica* (GEI - Informatics Engineering Degree) imparted by Facultat d'Informàtica de Barcelona (FIB) at Universitat Politècnica de Catalunya (UPC).

1.1 Terms and concepts

The most important concepts are listed and explained in this section in order to provide the reader with a fair starting point to help comprehend every further aspect of this work. Every element listed includes a reference for supplementary information.

- **High-Performance Computing (HPC)**: It consists in the execution of hugely complex applications by large computer systems made up by hundreds of processors working in parallel. Therefore, the system is characterized by massively high computational power compared to personal computers. [3]
- **Git**: It is a version control software. [4]
- **Continuous Integration (CI)**: It is a continuous methodology that consists of a set of scripts to build and test the commits of a repository. Those scripts are triggered to every push, and they help decrease the chances of introducing errors in the code. [5]
- **Job**: The most basic units of CI are jobs. A job is an execution process defined by its own script. [6]
- **GitLab runner**: Agents in charge of running the GitLab CI jobs triggered by a commit. Different types and number of runners can be configured.
- **Pipeline**: The CI main structure are pipelines, which are a set of jobs. Pipelines are divided into stages, each including one or more jobs, and intended to let developers design the workflow any way they want. [7]
- **Artifact**: File produced during the execution of a Pipeline's job. Artifacts can be downloaded or accessed for a certain amount of time after executing the pipeline. [8]
- **Stage**: Group of jobs with shared characteristics that can be executed in parallel.
- **Vector Processor Unit (VPU)**: It is a processor type that implements Single Instruction Multiple Data (SIMD) instructions, which are a type of instructions that process multiple data in a single instruction, exploiting data-level parallelism. [9]

- **Instruction Set Architecture (ISA)**: It is an interface between hardware and software. It defines the operations and instructions that the hardware executes without specifying how they are implemented. Thus, the hardware design implementation is in the hands of the programmer, whose job is to meet the specifications. [10]
- **RISC-V**: It is an open-source Instruction Set Architecture (ISA) developed at the University of California in Berkeley. It is an extensible architecture based on modules. Each module consists of a set of basic instructions that target a specific set of features. This is a fundamental aspect since hardware developers can implement more specialized processors, depending on the modules implemented. [11]
- **Field Programmable Gate Array (FPGA)**: It is a semiconductor device based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. This device can be reprogrammed to desired application or functionality requirements after manufacturing. [12]
- **Bitstream**: A file specifying the configuration of all the internal FPGA resources, so the device behaves as the programmer has developed. [13]
- **Benchmark**: It is a test used to measure the hardware or software performance, which is useful to compare different systems between them. [14]
- **Git submodule**: It is a git repository kept as a subdirectory of another git repository. [15]

1.2 Aim

Any time new lines of code are pushed to a project, a bug could be introduced unknowingly. That bug could either be immediately found, or hidden until it is detected several updates after. If that is the case, that bug could be extremely hard to find, even if using version control.

In a project with several layers designed by several partners at different points in time, a hidden bug becomes a real challenge, resulting in lots of hours spent purifying the code, instead of further developing new features. As an additional layer of security, the usage of CI becomes crucial.

By adding new layers of testing, the reliability of the whole project becomes improved, as developers can assume more safely that the code has no undetected bugs.

As a large-scale project with multiple programmers involved, trusting other people's code —and the code written by itself— is a life-saving functionality. Consequently, this project aims at guaranteeing a level as high as possible of security, by avoiding the spending of both human and technological resources detecting bugs, that can be instantly detected otherwise with a tough CI implementation.

1.3 Stakeholders

- **Researcher**
  The researcher is responsible for developing all the tasks associated with this project and documenting its progress.

- **Director**
  The director is both the project's and the researcher's supervisor, whose job is to plan and validate all the tasks and assist with all required help. For this project, the director is also in charge of *Mobile and embedded-based HPC* research team at Barcelona Supercomputing Center (BSC), which is directly involved with new technologies and EPI.

- **Barcelona Supercomputing Center (BSC)**
  BSC is involved in the development of RISC-V hardware support. It will use and benefit from the results of this project, as it will contribute to its role within the EPI project. [16]

- **European Processor Iniciative (EPI)**
  The European project will benefit from the continuous automatic hardware validation adapted to RISC-V architectures, as all EPI partners will be able to focus on further development of the project. [17]

- **Scientific community**
  The development of a new processor aimed at HPC will be a great improvement for the global technological scene, fundamentally for the European supercomputation community.

- **Team coworkers**
  All the researches within the BSC team responsible for developing the project will benefit by having the opportunity to focus on the validation of certain crucial aspects of the design, rather than constantly having to validate any new design update.

1.4 Technical competences

Here you can see the technical compentences related to this project.

- **CEC2.1**: To analyse, evaluate, select and configure hardware platforms for the development and execution of computer applications and services. *[In depth]*

  All the benchmarks have been compiled and executed in an FPGA environment.

- **CEC2.2**: To program taking into account the hardware architecture, using assembly language as well as high-level programming languages. *[Enough]*

  Bash scripts have been developed for the CI.

- **CEC2.3**: To develop and analyse software for systems based on microprocessors and its interfaces with users and other devices. *[In depth]*

  Benchmarks have been analyzed and tested in different processors.

- **CEC2.4**: To design and implement system and communications software. *[A little bit]*

  A communication method has been implemented between the CI and the FPGA.

- **CEC3.1**: To analyse, evaluate and select the most adequate hardware and software platform to support embedded and real-time applications. *[Enough]*

  The implemented benchmarks have been chosen based on the necessities of the EPI project and the architectures where those have been tested.

- **CEC4.2**: To demonstrate comprehension, to apply and manage the guarantee and security of computer systems. *[Enough]*

  The CI has been designed to protect the EPI repository.

1.5 Justification

Starting from a minimal setup able to produce a bitstream and deploy it for further usage, it only included a single benchmark with a simple design. This allowed for some basic testing, but many bugs could still go unnoticed.

In fact, right after my incorporation to BSC's team, a bug was detected, and after some research the most reasonable explanation seemed to be that it appeared a couple commits prior to its discovery.

The only benchmark implemented failed to detect that bug, because it is a benchmark that does not stress the specific element of the processor that had that precise bug.

That put into light the need for an improvement of the security of the project's CI.

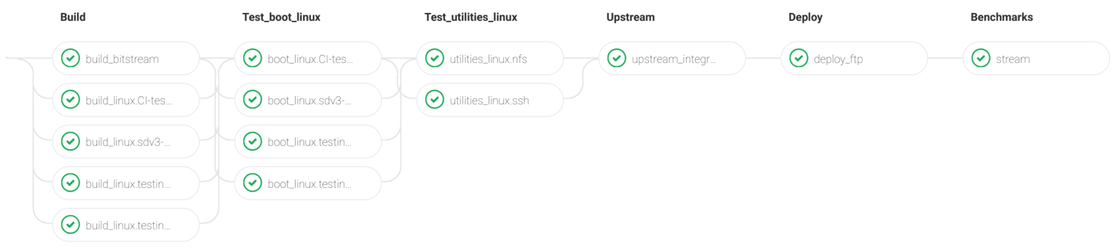


Figure 1: Initial CI pipeline prior to this work. Benchmarks are executed in the last stage. Notice how only the last step of the pipeline is relevant to this work, initial stages are outside the scope of this project. *Screenshot from EPI's GitLab.*
As a result, any tough implementation should have a set of benchmarks where each benchmark targets a different element of the processor, such as the vectorial load or the cache coherence. Therefore, when a benchmark fails developers can safely assume the bug is hidden within that element of the processor.

Given the CI will trigger a new pipeline for every new code being pushed to the repository, and because that pipeline will execute all implemented benchmarks over that new processor version, a benchmark not failing previously to the new push, but failing for that new version, tells developers when the bug has been introduced.

Thus, if the CI has a wide range of benchmarks implemented correctly, when a benchmark fails, it will give a decent amount of information to programmers, so they can find any bug or error relatively fast, saving lots of resources for the addition of new features, rather than detecting bugs.

That is the reason why this project's purpose is to end with a CI able to stress every possible failure point of the processor, consequently adding a wide range of benchmarks and making sure no processor characteristic is left behind.

Additionally to the inclusion of new benchmarks to the CI, and the improvement of the way they were implemented, the introduction of automatical performance analysis is an interesting feature the CI lacks.

Benchmarks will not only help detect critical bugs that make the processor malfunction, but benchmarks will also provide execution times and other performance statistics, so bugs slowing the processor down will also be spotted, and it will also help the improvement of the performance of the processor's design.

1.6 Scope

Prior to this work, pipelines triggered by the already existing CI had a design as seen in Figure 1. As mentioned early, this implementation lacked a more comprehensive set of benchmarks for testing purposes, therefore if a single most important crucial had to be stated, that is the addition of new benchmarks.

Obviously, a well-planned project follows an extensive list of objectives and secondary goals to help the author develop the work as successful as possible.

Nevertheless, the author should also be aware of any structural risk that could lead to a scenario where a specific task can not be satisfied.

All milestones and possible obstacles are listed hereunder.

1.6.1 Objectives and sub-objectives

Main objectives offer a set of sub-objectives derivated from them. All of them are listed and explained below.

- **Improvement of Benchmarks stage**
  As a first objective, it is established to take the already implemented benchmark as a starting point and modify how the benchmark execution is handled in order to pave the way for all further objectives.
  - **Extract performance metrics** to automatize the performance analysis of all benchmarks.
  - **Generalize scripts to adapt to any new benchmarks** for easily adding new benchmarks to the set and simplifying maintenance.

- **Comprehensive automatic bug detection**
  Identifying which benchmarks are interesting for the sake of stressing as many characteristics as possible of the processor's design and implementing them accordingly to the new scripting design, done as a previous objective.
  - Addition of **memory bound** benchmarks.
  - Addition of **compute bound** benchmarks.
  - Addition of benchmarks to stress different **instruction types of the Vector Processor Unit (VPU)**.
  - Identification and addition of any other benchmark that targets an interesting element.
- **Automatic generation of performance plots**
  Generate performance plots of each benchmark, comparing the metric results with previous versions in the master branch (releases).

1.6.2 Requirements

- **Guarantee correct benchmark execution**
  It is fundamental to test every binary before using it in the CI.
- **Avoid false positives**
  When a benchmark is malfunctioning as a result of a problem with the code being tested, the CI must act accordingly, and the job should fail.
- **Reject a commit if any of its benchmark jobs fails**
  If a benchmark fails its execution for a certain commit, that commit will not be successfull and will not be pushed to master branch.

1.6.3 Risks and obstacles

- **GitLab server malfunctions**
  In case the GitLab server is down or malfunctioning, no pipelines will be able to be executed. And no further commits will be done until it works again.

  This risk is critical, as new code updates can not be merged from any partner, therefore no version control will be done, and no automatical testing will be done either.

- **Cluster is down**
  In the scenario where the cluster where pipelines start execution is down, no further pipelines will start till either solving the issue or configuring it in another environment, which should be doable as a task within the scope of this project, but time-consuming, meaning other tasks could be sacrificed.

- **FPGAs are down**
  Depending on whether only one or both FPGAs are down, this situation will be critical or simply an annoyance, as having a functioning FPGA is enough, even though development will probably slow down, because of all pipelines and interactive sessions will end up in the same FPGA.

  Otherwise, if both FPGAs are down, no further development requiring them will be possible, and further tasks will be necessary to repair or replace.

1.7 Methodology and rigour

The EPI project is to be seen as a small, yet important, piece of this ambitious puzzle. With so many teams working in several pieces at once, some strict organisation is required. In order to achieve that, an *agile* methodology will be followed.

1.7.1 Communication

In the context of EPI, there is a weekly meeting with the partners involved in the Register-Transfer Level (RTL) validation using FPGAs. This weekly meeting acts as a valuable way of communication between all partners, as well as being a comfortable way of showing my progress to all of those interested, in addition to getting useful feedback.

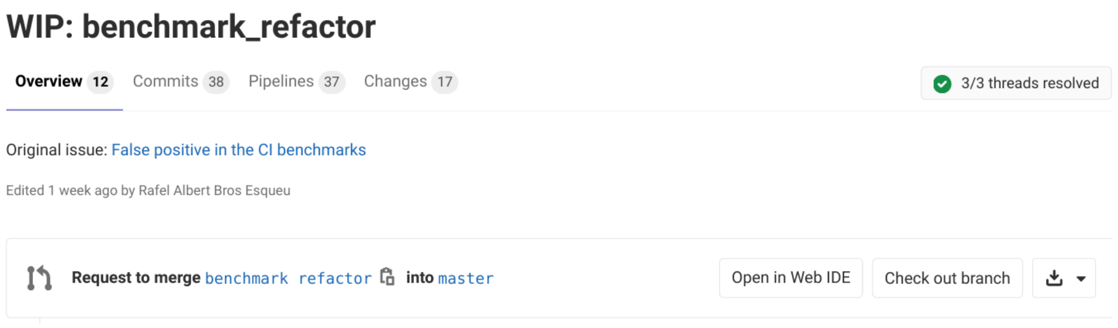
However, logically my biggest and most reliable source of help comes from both my co-workers along with my director, co-director and my tutor. Throughout my day to day at work, I constantly ask for feedback in order to take development decisions accordingly to the necessities of the overall project. Furthermore, a weekly meeting with both my director and co-director is held.

1.7.2 Monitoring tools

The project is hosted on a repository in GitLab, an open-source software based on Git that allows developing code collaboratively. GitLab provides different monitoring tools used in this project. [18]

GitLab merge requests

For the development of a particular feature, a new branch will be created. Altogether with issues, a merge request will be created, and the implementation of that feature is to be discussed there, including version control.



WIP: benchmark_refactor

Original issue: False positive in the CI benchmarks

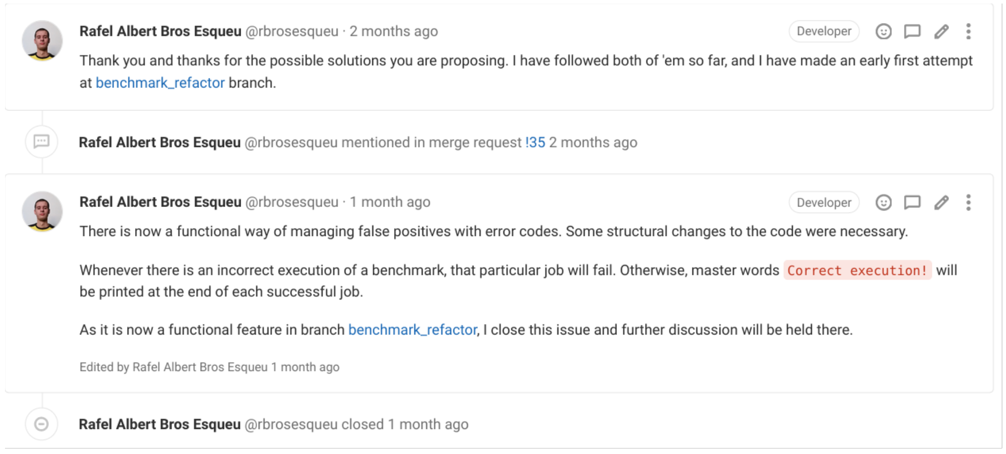
Edited 1 week ago by Rafel Albert Bros Esqueu

Request to merge benchmark refactor into master

Figure 2: Example of a GitLab merge request. It gives information about the last executed pipeline for that branch, as well as lets the creation of threads for discussion related to the development of the branch's features. *Screenshot from EPI's GitLab.*

GitLab issues

Being a GitLab project, the usage of GitLab issues appears to be the best way of organisation. Issues are created in two different repositories: one with access for all the partners involved in the RTL validation using FPGAs, and another for exclusively my team co-workers. This way, the development can adapt to the real needs of all involved in the project, while offering the opportunity to get a more personal touch when seeking advice for some aspects of my work.


Rafel Albert Bros Esqueu @rbrosesqueu · 2 months ago

Thank you and thanks for the possible solutions you are proposing. I have followed both of 'em so far, and I have made an early first attempt at benchmark_refactor branch.

Rafel Albert Bros Esqueu @rbrosesqueu mentioned in merge request !35 2 months ago

Rafel Albert Bros Esqueu @rbrosesqueu · 1 month ago

There is now a functional way of managing false positives with error codes. Some structural changes to the code were necessary.

Whenever there is an incorrect execution of a benchmark, that particular job will fail. Otherwise, master words `Correct execution!` will be printed at the end of each successful job.

As it is now a functional feature in branch benchmark_refactor, I close this issue and further discussion will be held there.

Edited by Rafel Albert Bros Esqueu 1 month ago

Rafel Albert Bros Esqueu @rbrosesqueu closed 1 month ago

Figure 3: Example of a GitLab issue. *Screenshot from EPI's GitLab.*

Private GitLab issues also act as a day-to-day method for internal organisation. The director, co-director and researcher are the only ones who have access there. All the progress with tasks will be reported there, as every task has its own *card*, where all progress will be briefed, as well as the current status of that task will be stated thanks to a five level system: *To Do*, *Doing*, *Blocked*. *Done* and *Archived*.

- **To Do:** Tasks yet to be started.
- **Doing:** Tasks already started, under development.
- **Blocked:** Tasks whose development cannot continue for any reason.
- **Done:** Tasks successfully finished, that are a dependency for a task yet to be finished.
- **Archived:** Tasks successfully finished, that are no longer a dependency.

1.7.3 Validation methods

As much as possible, everything will be tested locally prior to pushing it to a repository. Once the new changes are online, that new push will trigger a new pipeline, whose results will be put into comparison with those obtained locally.

Moreover, corner cases will also be deeply analysed in order to detect as many bugs as possible.

2 Temporal planning

The estimated duration of this project is 560 hours, starting at the end of February and finishing by the start of July. In particular, the starting day is the 14th of February, 2022, while the ending day is the 1st of July, in the same year. The oral defence's date has yet to be defined, so the last possible date for the defence is the one used as a reference.

2.1 Description of Tasks

Throughout this section, the reader will find a detailed explanation of all the tasks that have been planned and scheduled for this project. Scheduling implies an estimation of hours for every task. The dependencies and time estimations for very task can be observed in Table 1. Planning has been done by considering all possible dependencies between tasks, specifying them accordingly.

Tasks are divided among four groups: *Project planning*, *Implementation of benchmarks*, *Performance analysis automatisation* and *Documentation.*

- **T1 Project planning**
  Planning and structuring the work takes place throughout these tasks. They act as a fundamental step for all the tasks to come. Most of the tasks are encompassed within GEP.
  - **T1.1 ICT tools to support project and team management**
    Initial screening over several digital tools with widespread usage that will be used to support the project's development.
  - **T1.2 Contextualization and project scope**
    Definition of the project's main objectives after understanding its place within the environment where it will be developed. This task includes the elaboration and delivery of documentation.
  - **T1.3 Tasking and temporal planning**
    Characterization of all the necessary tasks to satisfy the project's main objectives, each task with an associated estimated time to finish it. This task includes the elaboration and delivery of documentation.
  - **T1.4 Economical perspective and sustainability**
    Study of both the project's expected sustainability and necessary bud-

get for the development of all tasks. This task includes the elaboration and delivery of documentation.

- **T1.5 Personal and professional skills for project and team management**
  Identification of essential skills and techniques for correct coordination and communication for working with other people.
- **T1.6 Integration in final GEP document**
  Development of the final GEP documentation of the project by bringing together all previous deliverables and solving any issues with them.
- **T1.7 Meetings**
  The meetings will extend throughout the duration of the project, from the first to the last weeks. They serve the purpose of getting valuable constant input from the two supervisors of the project while also understanding the specific needs of all the glsepi partners. One with both my director and co-director, which requires an average of an hour per week. The second one takes place every Thursday and demands about two hours a week.

◦ **T2 Implementation of benchmarks**

- **T2.1 Understanding of the CI environment**
  Understanding the basics of Continuous Integration, as well as `YAML` codes.
- **T2.2 Understanding of an FPGA environment**
  Adjusting to the usage of an FPGA and its environment. Learning to compile benchmarks and executing them in the FPGA. Also, getting familiar with setting up the infrastructure.
- **T2.3 Refactoring benchmark structure**
  Adapting the original implementation prior to this work to a newer, simpler and more scalable one.
- **T2.4 Selection of benchmarks**
  Out of all available benchmarks, it is necessary to decide the most useful ones to detect any possible bug throughout the processor's features.
- **T2.5 Compilation of benchmarks**
  Once the benchmarking set has been established in T2.4, getting a functional binary of every benchmark is the next step. This task focuses on the compilation of the chosen benchmarks and testing them in an FPGA. If a binary is not functional and that particular benchmark requires to be recompiled, additional time may be necessary for this task.

  - **T2.6 Implementation in the CI**
    Implementation of all the binaries obtained in T2.5 in the new CI structure designed in T2.3 for benchmarking purposes.
  - **T2.7 CI validation**
    Ascertain the implementation done in T2.6 is valid. In case it is not, it may come back to the previous task to solve any problem detected. As a result of this task, a pipeline should be running with an operational job for every benchmark.

- **T3 Performance analysis automatisation**
  - **T3.1 Selection of useful metrics**
    Analysis of all the metrics provided by the execution of each benchmark and selection of the most useful ones to evaluate the processor's performance.
  - **T3.2 Implementation of metrics extraction mechanism**
    Implementation of the necessary code modifications for all benchmark jobs to provide the decided metrics in T3.1 as an artifact for later use.
  - **T3.3 Creation of a new CI stage**
    Addition of a new stage to the CI pipelines. Plots are going to be automatically generated and exported in that stage.
  - **T3.4 Implementation of plot generation mechanism**
    Ensuring that the new CI stage created in T3.3 automatically produces a plot for every benchmark job, using the data produced during benchmark execution and extracted through the implementation in T3.2. Plots generated as artifacts should be stored externally.
  - **T3.5 Implementation of data version history**
    Necessary modifications through both the CI and the external server where plots (produced after T3.4) will be stored for guaranteeing a historical data set from all executed pipelines.

- **T4 Documentation**
  - **T4.1 Writing the final document**
    Writing all the work done throughout the project as detailed as possible, including the final GEP delivery.
  - **T4.2 Oral defence rehearsal**
    Planning the defence rehearsal and all the essential information that should be explained. Designing the slides that will be used as support.

2.2 Summary of tasks

Id.	Description	Hours	Dependencies
T1	Project Planning	135	
T1.1	ICT tools to support project and team management	5	-
T1.2	Contextualization and project scope	25	-
T1.3	Tasking and temporal planning	10	T1.2
T1.4	Economical perspective and sustainability	10	T1.3
T1.5	Personal and professional skills for project and team management	5	-
T1.6	Integration in final GEP document	20	T1.4
T1.7	Meetings	60	-
T2	Implementation of benchmarks	205	
T2.1	Understanding of the CI environment	30	-
T2.2	Understanding of an FPGA environment	15	-
T2.3	Refactoring benchmark structure	30	T2.1, T2.2
T2.4	Selection of benchmarks	10	T2.3
T2.5	Compilation of benchmarks	20	T2.4
T2.6	Implementation in the CI	75	T2.5
T2.7	CI validation	25	T2.6
T3	Performance analysis automatisation	105	
T3.1	Selection of useful metrics	15	T2.7
T3.2	Implementation of metrics extraction mechanism	20	T3.1
T3.3	Creation of a new CI stage	10	-
T3.4	Implementation of plot generation mechanism	40	T3.2, T3.3
T3.5	Implementation of data version history	20	T3.4
T4	Documentation	115	
T4.1	Writing the final document	80	-
T4.2	Oral defense rehearsal	35	-
Total		560	-

Table 1: Summary of all the project tasks, including the estimated working hours of each task and dependences between them. *Source: own compilation.*

See Table 4 for the updated summary of tasks.

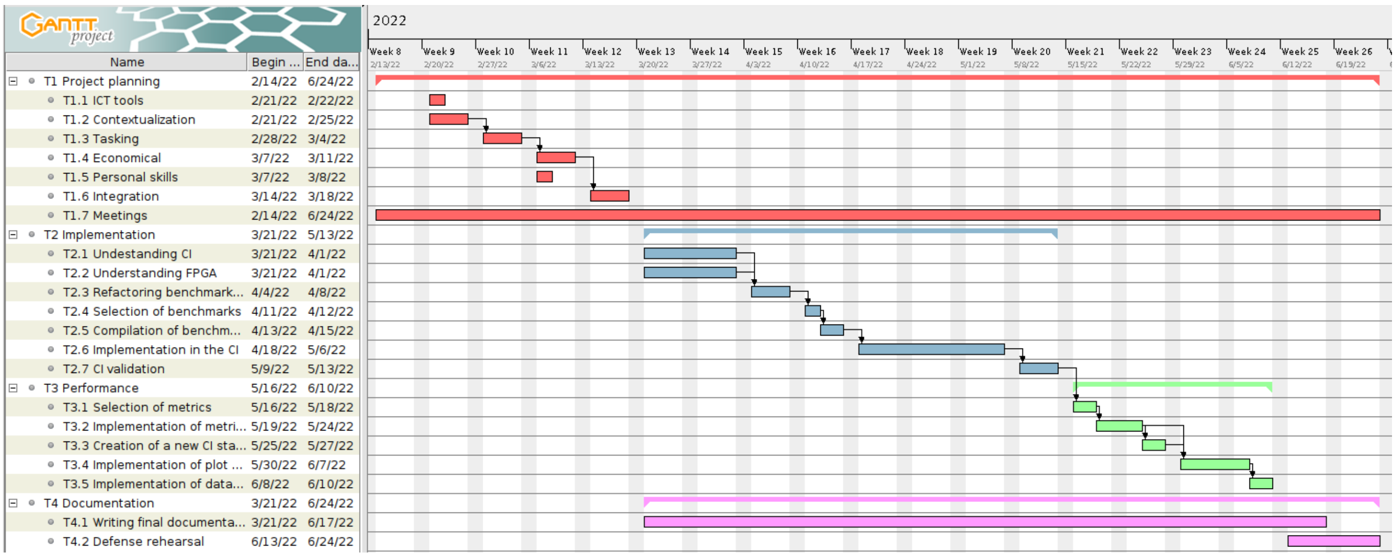
2.3 Necessary resources

All the necessary resources are defined in Table 2. Resources are classified depending on their nature: *human, hardware, software* and *additional* ones.

Item	Description	Tasks
Human resources		
Researcher	Main project developer	All
Director	Project supervisor	All
Co-Director	Project supervisor	All
GEP Tutor	Project planning supervisor	T1.{1..6}
Hardware resources		
Computer	Laptop DELL Latitude 7490	All
FPGA	Two Xilinx Virtex Ultrascale+ VCU128	T2, T3
HCA	Server to which the FPGA is connected to	T2, T3
Software resources		
Git	Control version software	All
GitLab	Remote repositories to access code	All
Atenea	UPC's web page with GEP material	T1.{1..6}
Thunderbird	Mail service used for coordinating meetings	T1.7
Vim	IDE for both code and text editing	All
Latex	Software system for document preparation	T1.{2..4}, T1.6, T4
GNUplot	Plot creation software	T3.4, T3.5, T4
Gantt Project	Gannt diagrams creation software	T1.3
Trello	Follow up and coordination tool	All
Additional resources		
Electricity	Basic supply	All
Water	Basic supply	All
Internet	Basic supply	All
Office supplies	Notebooks, pens, desk and chair	All

Table 2: Summary of all the necessary resources of this work, including the tasks that require them. *Source: own compilation.*

2.4 Gantt chart



Name	Begin ...	End da...
T1 Project planning	2/14/22	6/24/22
T1.1 ICT tools	2/21/22	2/22/22
T1.2 Contextualization	2/21/22	2/25/22
T1.3 Tasking	2/28/22	3/4/22
T1.4 Economical	3/7/22	3/11/22
T1.5 Personal skills	3/7/22	3/8/22
T1.6 Integration	3/14/22	3/18/22
T1.7 Meetings	2/14/22	6/24/22
T2 Implementation	3/21/22	5/13/22
T2.1 Undestanding CI	3/21/22	4/1/22
T2.2 Understanding FPGA	3/21/22	4/1/22
T2.3 Refactoring benchmark...	4/4/22	4/8/22
T2.4 Selection of benchmarks	4/11/22	4/12/22
T2.5 Compilation of benchm...	4/13/22	4/15/22
T2.6 Implementation in the CI	4/18/22	5/6/22
T2.7 CI validation	5/9/22	5/13/22
T3 Performance	5/16/22	6/10/22
T3.1 Selection of metrics	5/16/22	5/18/22
T3.2 Implementation of metri...	5/19/22	5/24/22
T3.3 Creation of a new CI sta...	5/25/22	5/27/22
T3.4 Implementation of plot ...	5/30/22	6/7/22
T3.5 Implementation of data...	6/8/22	6/10/22
T4 Documentation	3/21/22	6/24/22
T4.1 Writing final documenta...	3/21/22	6/17/22
T4.2 Defense rehearsal	6/13/22	6/24/22

Figure 4: Gantt chart. *Source: own creation with GanttProject.*

2.5 Risk management

Throughout the project, several possible situations could lead to a scenario where a specific task can not be satisfied as expected. As a way to adapt to those possible circumstances, the risks must be analysed. Furthermore, possible outcomes at attempting to solve the problem are also planned.

Out of all possible risks, infrastructure-related ones —in case they were to happen— are the most likely to imply a significant hold in the project. These risks are related to both hardware and software structures: the HCA server or the FPGAs on the hardware side and GitLab repositories on the software side. If the HCA server stopped functioning, an additional task for repairing the server would be necessary, and the delay would depend on the difficulty of that repairing. Alternatively, in case fixing it not being feasible, another way to trigger CI pipelines would be vital, implying another task for setting the new server up. That would be very time-consuming, and very likely some later stage tasks would not meet the specified deadline.

The same situation applies with both FPGAs, even though one FPGA being down is not a critical issue. The project can continue with only one FPGA, but it would probably slow down the testing phases of the project depending on the FPGA usage by other team members. Therefore, the critical scenario would be if both FPGAs are down, where additional tasks for repairing or replacing them would be needed.

Finally, the Git repositories being down implies that no further development of the code can be tested and implemented for all EPI partners. However, setting up a new repository in another domain is not heavily time-consuming and could be solved through an auxiliary task that would take a week maximum.

Some minor risks have been specified during the initial task explanation. In its majority, those risks imply an increase of a couple of hours per task (very rarely more than a day of additional work). Nevertheless, these minor risks are very likely to happen compared to the most significant risks, whose probabilities are relatively low. As a result, the three tasks more likely to encounter issues have been modified by adding five additional hours to their expected time. Those are T2.5, T2.6 and T3.2.

Description	Tasks	Probability	Solution	Hours
Both FPGA's down	T2,T3	Low	Repairing at least one or buying a new one	50
HCA down	T2,T3	Low	Repairing it or finding an alternative server	25
Git repository down	T{2..4}	Low	Setting up a new repository	10
Compiled binary not functional	T2.5	High	Re-compiling it	2
CI implementation malfunctions	T2.6	Medium	Testing and re-implementation	5
PA implementation malfunctions	T3.2	Medium	Testing and re-implementation	5

PA as *Performance Analysis*

Table 3: Summary of all possible risks, including estimated hours for solving each. Consider that every stated solution would become a task of its own, in case that specific risk became a reality. *Source: own compilation.*

2.6 Work plan modifications

There have been some changes with the initial planning. During the development of task block T3, a new priority in the project emerged that was considered of maximum importance. In particular, T3.4 and T3.5 have been affected. The development of task T3.4 was interrupted after the implementation of an early plot generation mechanism after 15 hours of work. Task 3.5 has been cancelled.

During task T2.7 (*CI validation*), it was detected that the implemented system did not correctly process failed jobs. Even though some hours had already been planned for the possibility of solving any bug detected during task T2.7, most of them had already been used on solving minor bugs. This problem was reported to both my director and co-director, and some EPI partners were involved in the discussion. Meanwhile, development resumed with task block T3 until a consensus was met. The dependency in task T3.1 with task T2.7 was considered not necessary, so it was removed.

After understanding the importance of a fully functional system that detects and reports failed jobs correctly, it was decided to prioritise this new feature.

Fundamentally, either this feature was not developed as part of this project, or some features previously planned had to be removed. Additionally, the most crucial goal of task block T3 lies in task T3.2. If metrics are extracted comfortably, they can be processed manually in a relatively straightforward manner, so automatisation of the later stage is not fundamental.

In consequence, the decision to add a new task and interrupt the expected development at task T3.4 has been made in harmony with all the ones involved with the project, The new task identified as T5.1 consists of implementing a system to detect and improve the different cases a job can exit due to a failure. In other words, to process exit errors such as timeouts or CPU hangs. Task T5.1 has been named *Implementation of a failure detection mechanism*, and the remaining available hours in both tasks T3.4 and T3.5 have been allocated for task T5.1, being a total of 45 hours. No budget modifications are necessary.

The project's development will end up as originally planned. Tasks T5.1, T4.1 and T4.2 are under development. They are expected to be finished within the hours assigned to them.

2.6.1 Updated summary of tasks

Id.	Description	Hours	Dependencies
T1	Project Planning	135	
T1.1	ICT tools to support project and team management	5	-
T1.2	Contextualization and project scope	25	-
T1.3	Tasking and temporal planning	10	T1.2
T1.4	Economical perspective and sustainability	10	T1.3
T1.5	Personal and professional skills for project and team management	5	-
T1.6	Integration in final GEP document	20	T1.4
T1.7	Meetings	60	-
T2	Implementation of benchmarks	205	
T2.1	Understanding of the CI environment	30	-
T2.2	Understanding of an FPGA environment	15	-
T2.3	Refactoring benchmark structure	30	T2.1, T2.2
T2.4	Selection of benchmarks	10	T2.3
T2.5	Compilation of benchmarks	20	T2.4
T2.6	Implementation in the CI	75	T2.5
T2.7	CI validation	25	T2.6
T3	Performance analysis automatisation	105	
T3.1	Selection of useful metrics	15	-
T3.2	Implementation of metrics extraction mechanism	20	T3.1
T3.3	Creation of a new CI stage	10	-
T3.4	Implementation of plot generation mechanism	15	T3.2, T3.3
T3.5	Implementation of data version history	-	-
T4	Documentation	115	
T4.1	Writing the final document	80	-
T4.2	Oral defense rehearsal	35	-
T5	Failure detection mechanism	45	
T5.1	Implementation of a failure detection mechanism	45	-
Total		560	-

Table 4: Updated summary of all the project tasks, including the estimated working hours of each task and dependences between them. *Source: own compilation.*

See Table 1 for the initial summary of tasks.

3 Economic planning

In order to establish a rigorous budget prediction, both staff and resource costs have been considered. The expenses are calculated by analysing the basic needs for each of the tasks described in this project. Accordingly, the hours worth of work required from every role within the project's staff is specified for every planned task. Additionally, the use of technical resources is also determined similarly.

There are four roles that exist within the project's staff: *Programmer*, *Technical writer*, *Project manager* and *Tester*. A brief summary of each role's responsibilities is explained below. The costs associated to each of the roles is showed in Table 5.

- **Programmer**
  The Programmer is the central role of this project. It is responsible for implementing all the features required throughout all project tasks. The Programmer will follow the Project Manager's planning.

  Therefore, its importance is fundamental in tasks within T2 and T3. In case the Tester detects any anomaly in those already implemented features, it will be the Programmer's responsibility to solve that issue and make the code functional.

- **Project manager**
  The Project Manager is responsible for specifying the organisation of the project tasks; its importance is crucial for the tasks in T1 group.

- **Tester**
  The Tester is responsible for validating all the work done by the Programmer. This role is essential in several T2 and T3 tasks, as some tasks require a bit of testing to guarantee a successful functionality.

  Furthermore, task T2.7 is exclusively the Tester's commitment, as all the programming work done in task 2.6 requires an extensive and in-depth independent testing process.

  Nevertheless, the Tester does not require knowing the details of neither the implementation nor the code, but a complete understanding of what is expected from every functionality is crucial. In case a bug is found, the Tester provides all the information available to the Programmer, whose job is to solve that error.

- **Technical writer**
  The technical writer is the uppermost responsible for writing the final project's

documentation and T1 deliverables, based on information nourished from work done by the programmer, the Tester and the project manager.

Staff	Yearly wage (€)	Cost (€) / Hour	Individual
Programmer	28973	15.09	R
Project Manager	51283	26.71	R, D, T
Tester	31085	16.19	R
Technical writer	35674	18.58	R

R:Researcher; D:Director and Co-director; T:GEP Tutor.

Table 5: Costs associated to every role within project's staff. [19] *Source: own compilation.*

3.1 Human resources

Each role has different importance depending on the task's purposes. Hence, every task has a distinct work distribution for every role. The distribution of hours worth of task's work for every role are shown in Table 6. The final costs associated to each role are shown in Table 7.

		Staff hours				
Id.	Description	Total	P	M	T	W
T1	Project Planning	135	60	100	60	95
T1.1	ICT tools to support project management	5	-	5	-	-
T1.2	Contextualization and project scope	25	-	15	-	10
T1.3	Tasking and temporal planning	10	-	5	-	5
T1.4	Economical perspective and sustainability	10	-	5	-	5
T1.5	Skills for project and team management	5	-	5	-	-
T1.6	Integration in final GEP document	20	-	5	-	15
T1.7	Meetings	60	60	60	60	60
T2	Implementation of benchmarks	205	170	-	35	-
T2.1	Understanding of the CI environment	30	30	-	-	-
T2.2	Understanding of an FPGA environment	15	15	-	-	-
T2.3	Refactoring benchmark structure	30	25	-	5	-
T2.4	Selection of benchmarks	10	10	-	-	-
T2.5	Compilation of benchmarks	20	15	-	5	-
T2.6	Implementation in the CI	75	75	-	-	-
T2.7	CI validation	25	-	-	25	-
T3	Performance analysis automatisation	105	90	-	15	-
T3.1	Selection of useful metrics	15	15	-	-	-
T3.2	Implementation of metrics extraction mechanism	20	15	-	5	-
T3.3	Creation of a new CI stage	10	10	-	-	-
T3.4	Implementation of plot generation mechanism	40	35	-	5	-
T3.5	Implementation of data version history	20	15	-	5	-
T4	Documentation	115	35	-	-	80
T4.1	Writing the final document	80	-	-	-	80
T4.2	Oral defense rehearsal	35	35	-	-	-
Total		560	345	100	110	175

P:Programmer; M:Project manager; T:Tester; W:Technical writer.

Table 6: Summary of all the project tasks, including the estimated working hours of each task and its distribution among different roles within project's staff. *Source: own compilation.*

Staff	Cost (€) / Hour	Hours	Cost (€)	Total with SS (€)
Programmer	15.09	345	5206.05	6767.86
Project Manager	26.71	100	2671.00	3472.30
Tester	16.19	110	1780.90	2315.17
Technical writer	18.58	175	3251.50	4226.95
Total			12909.45	16782.28

Table 7: Final costs for every role within project's staff. *Source: own compilation.*

3.2 Material resources

The costs of hardware that will be associated to the budget is defined by amortizations. There are three main hardware elements in this project: a laptop, the HCA server and two FPGAs. In order two calculate all three amortizations, the real cost are to be known.

The DELL Latitude 7490 laptop is sold for 749€ at retail pages. [20] The laptop will be used for the whole duration of the project: 560 hours.

$$\text{Laptop amortization} = \frac{749\ € \times 560\ \text{hours}}{1760\ \text{hours} \times 4\ \text{years}} = 59,58\ €$$

Considering a four years period, as well as a total of 1760 hours per year.

All the costs associated with the usage of the HCA server are private and confidential. Therefore, an aproximated 5000€ figure for a six months access period is used. This cost will not be considered hardware nor software, but as a service such as internet or electricity.

For both FPGAs, the cost for buying one is $10794.00 [21], which in euro currency is about 1.10% of that, as of March 2022, totalling 23746.80€ for both FPGAs as the retail cost.

The FPGAs are directly bought to the US, so importation costs and additional fees are considered with a 5.00% increase in its cost, being 24934.14€ the final figure for buying both FPGAs.

Considering that for every new pipeline triggered an FPGA will be working for about two hours, and that for the total length of task gropus T2 and T3 a new pipeline will be triggered for every five hours of development, the amortization calculations are as follows. As there are two FPGAs, the two hours figure for every pipeline is divided by two.

$$\text{FPGAs amortization} = \frac{24934.14\ € \times \dfrac{170\ \text{T2 hours} + 90\ \text{T3 hours}}{5\ \text{hours}}}{1760\ \text{hours} \times 4\ \text{years}} = 184.17\ €$$

Considering a four years period, as well as a total of 1760 hours per year.

Electricity has been calculated by using the average cost of electricity with a non-changin hour to hour plan, meaning a constant kWh cost at 0.27€. Considering the worst case scenario, a laptop at full performance could consume up to 75W per hour, [22] given the approximate 545 hours worth of work, it amounts for 110.36€ for the whole project.

For the internet figure, a standard cost for top high speed connection plan is used, totalling 240€ for the project's total duration. [23]

Item	Description	Cost (€)
Hardware resources		
Computer	Laptop DELL Latitude 7490	59.58
FPGA	Two Xilinx Virtex Ultrascale+ VCU128	184.17
HCA	Server to which the FPGA is connected to	500.00
Software resources		
Git	Control version software	Free to use
GitLab	Remote repositories to access code	Free to use
Atenea	UPC's web page with GEP material	Free to use
Thunderbird	Mail service used for coordinating meetings	Free to use
Vim	IDE for both code and text editing	Free to use
Latex	Software system for document preparation	Free to use
GNUplot	Plot creation software	Free to use
Gantt Project	Gannt diagrams creation software	Free to use
Trello	Follow up and coordination tool	Free to use
Additional resources		
Electricity	Basic supply	110.36
Internet	Basic supply	240.00
Office supplies	Notebooks, pens, desk and chair	350.00
Total		1444.11

Table 8: Summary of all costs of the necessary resources of this work. *Source: own compilation.*

3.3 Contingency

A contingency fund is defined for prevention against unexpected situations not considered through the whole planning. The fund is the equivalent of 20% of the planned budget, as seen in Table 10 totalling 3645.28€.

$$\text{Contingency} = \text{Resources}(€) \times \text{Fund}(\%) = (16782.28 + 1444.11) \times 0.2 = 3645.28 \text{ €}$$

3.4 Incidentals

The budget has to be also adapted to situations that, in contrast to contingencies, can be predicted. There are four main incidental situations predicted, with an arbitray percentage chosen to specify how likely that situation is to happen. The incidentals have already been described as risks in previous sections of the project, costs shown in Table 9.

Description	Hours	Estimated cost (€)	Risk (%)	Final cost (€)
Both FPGA's down	50	980.85	5	49.04
HCA down	20	392.34	10	39.23
Repository down	5	98.08	20	18.16
Not meeting project's deadline	50	980.85	10	98.08
Total				205.95

Table 9: Incidentals defined and its costs. *Source: own compilation.*

3.5 Final budget

The final budget can be seen in Table 10. It takes into consideration both human and material resources (16782.28€ and 1444.11€ respectively), contingencies and incidentals. The budget totals 22077.62€.

Description	Cost (€)
Total resources	18226.39
Contingencies	3645.28
Incidentals	205.95
Total	22077.62

Table 10: Final budget. *Source: own compilation.*

3.6 Management control

A budget for a big project will not always be exact, as there are many unforeseen situations that can arise throughout development. Accordingly, a set of procedures are used to control the budget and stablish a comparison between the planned budget and the final costs.

Human deviation
Tasks will not always meet the exact amount of working hours expected. Therefore,

the deviation is calculated as follows for every task.

$$\text{Human deviation (€)} = \sum_{i \epsilon task} (estWorkHours + realWorkHours) \times costWorkHour$$

Being i any role that takes part in a particular task.

Hardware deviation
In a similar manner, the hardware usage may vary from the expected hoursi. The deviation is calculated for every hardware used during a specific task, and all deviations are summed to get the final hardware deviation of that particular task.

$$\text{Resource deviation (€)} = \sum_{i \epsilon task} (estUsedHours + realUsedHours) \times costUsedHour$$

Being i any resource used in a particular task.

Task deviation
Both Human and Hardware deviations are used to get the total deviation of a task.

$$\text{Task deviation (€)} = \text{Human deviation} + \text{Resource deviation}$$

Electrical deviation
Electricity is a resource whose cost has been calculated based on the amount of hours worth of work. Therefore, its deviation can also be calculated.

$$\text{Electrical deviation (€)} = (estUsedHours + realUsedHours) \times costUsedHour$$

Total deviation
Finally, both the overall Electrical deviation and all Task deviations are used to get the final project's deviation.

$$\text{Total deviation (€)} = \sum_{i \epsilon project} (\text{Task deviation}) + \text{Electrical deviation}$$

Being i all the tasks within the project.

4 Sustainability

4.1 Self-assessment

Luckily, in recent years there has been an enormous increase in people's concerns about the consequences of our activities and, most significantly, the effects we as human beings have on the environment. As a result of this new social perspective, a self-evaluation is required to analyse my strengths and weaknesses regarding the sustainability of my work and my knowledge about it.

I do my best to make eco-friendly decisions in my daily life, yet it is not something I can control in my workplace. Many aspects and indicators used for measuring and analysing our impact on the environment are still unknown to me. Most importantly, the extensive list of possible effects my activities have on various environmental aspects.

Even though this project will reduce the number of computing hours, therefore decreasing the electricity required, its primary focus was to improve the developer's quality of life, so the social perspective was the main reason behind this work. At the same time, the environmental cause was a secondary one.

From an economic perspective, this has been the first time I have done an analysis as deep as the previous sections. The quantity of aspects to consider is more considerable than I had expected.

I hope to develop this project as sustainable as possible based on these new insights.

4.2 Economic dimension

Regarding PPP: Reflection on the cost you have estimated for the completion of the project?

The overall cost estimated for successfully satisfying the project is 22077.62€, explained in detal in previous sections, particularly shown in Table 10.

Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)?, and ... How will your solution improve economic issues (costs ...) with respect other existing solutions?

The automatization of the bug detection process for every new code push to the EPI repositories allows for the drastic reduction of working hours, as developers will not be required to detect which feature has a bug hidden. Instead, the automatic process will detect that bug the exact moment a push is done to the repository, allowing developers to simply look at that particular piece of code instead of having to look once many new pushes have been made.

4.3 Environmental dimension

Regarding PPP: Have you estimated the environmental impact of the project

Besides the electrical consumption, there is no other environmental impact caused by this project. Therefore, the environmental footprint will be defined by the source of the electricity used.

As of 2021, 75.27% of electricity provided in Catalonia comes from non-renewable sources, so that would be the direct environmental impact of the project. [24]

Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?

Several strategies will be followed during development to minimize the project's environmental impact, basically focusing on reducing the electricity expenditure through, for example, reducing computing hours.

One of the strategies will be to avoid triggering unnecessary pipelines, as a pipeline is automatically triggered with every push to the repository. Several hours ' worth of FPGA's work will be avoided by not automatically triggering pipelines for under development pushes that were not intended to test anything.

Additionally, when a pipeline is necessary to be executed for testing purposes, some steps or stages will be avoided as they will not be crucial at that point, being activated again once development is finished.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?, and ... How will your solution improve the environment with respect other existing solutions?

Prior to the implementation of this work, all testing is done manually. The same situation occurs for performance analysis. All this work has to be done manually;

everything requires a considerable amount of computing hours and human hours worth of work.

Furthermore, whenever a bug is encountered, a large amount of testing is necessary to detect which push was responsible for introducing that bug. Constant automatic testing drastically reduces the testing hours, saving human and electrical resources, as it quickly detects the first push where that particular bug occurred.

4.4 Social Dimension

Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project?

This project offers the possibility to learn and develop team skills as communication with all the actors involved is fundamental for the proper development of this work. Furthermore, scheduling is also crucial for meeting the deadline with all tasks successfully achieved.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?, and ... How will your solution improve the quality of life (social dimension) with respect other existing solutions?

Right now, it is a simple system that does not detect all the bugs that could be otherwise detected with the addition of more benchmarks. This situation implies that developers will be able to focus on developing and improving the EPI designs rather than spending time testing the code already developed, which is a tedious job.

Most importantly, developers have to wait for those trying to detect the bugs previously introduced into the design to improve the processor's design further. This supposes a significant amount of pressure on the later ones, while it also stops the development of the EPI project.

Regarding Useful Life: Is there a real need for the project?

As previously stated, the automatization of bug detection improves the lives of the developers involved with the EPI project, while it also supposes the avoidance of bug detection bottlenecks, where progress can not resume until a specific bug is detected and solved.

5 Technical background

5.1 High Performance Computing

High-Performance Computing (HPC) consists in the execution of applications using a large amount of processors and accelerators working in parallel. The main objective is performing scientific calculus to, for example, produce simulations of magnetic fields, climate change, etc.

A scientific application would need too much time to produce its results without HPC, since its essential characteristic is the possibility to use the computing elements in parallel.

Parallelism can be obtained at different levels, which depend on where they are extracted:

- Data: Processing multiple data simultaneously through vectors, which is performed through Single Instruction Multiple Data (SIMD) instructions.
- Instruction: CPUs are pipelined to allow the execution of numerous instructions concurrently, among other ways of extracting instruction parallelism.
- CPU: A processor is composed of various cores, allowing the execution of many parallel programs allocated in different cores.
- Node: CPUs are typically sold in a socket form factor, which allows them to be mounted on a motherboard, conforming a compute node. Compute nodes are independent units of computation and can be connected through a network such as Ethernet or Infiniband.

These are some examples of parallelism at different levels that can be found in an HPC cluster.

5.1.1 Instruction Set Architecture

An ISA acts as an interface between hardware and software. It defines how the hardware is controlled by the software.

The only way for users to interact with hardware is using an ISA. For this reason, tools are important because they ease the interaction with the ISA, abstracting the user from the actual ISA specification.

In the specific case of the HPC industry, the most used ISA has been x86, for example Intel and AMD implement processors with it. Therefore, the majority of tools, such as compilers or libraries, are designed and optimized for a x86 architecture.

The x86 ISA is not the only one currently in the market, new ISAs have arised in the last years, such as Arm. Generally, in order to use those ISAs, developers have to pay royalties. An exception is RISC-V.

As RISC-V is an open-source ISA, its usage does not depend on royalties. It is a modular ISA, because its instruction set is divided in modules, each module targets a specific set of features. [11] It allows hardware developers to only implement the modules that they require.

5.1.2 Vector architectures

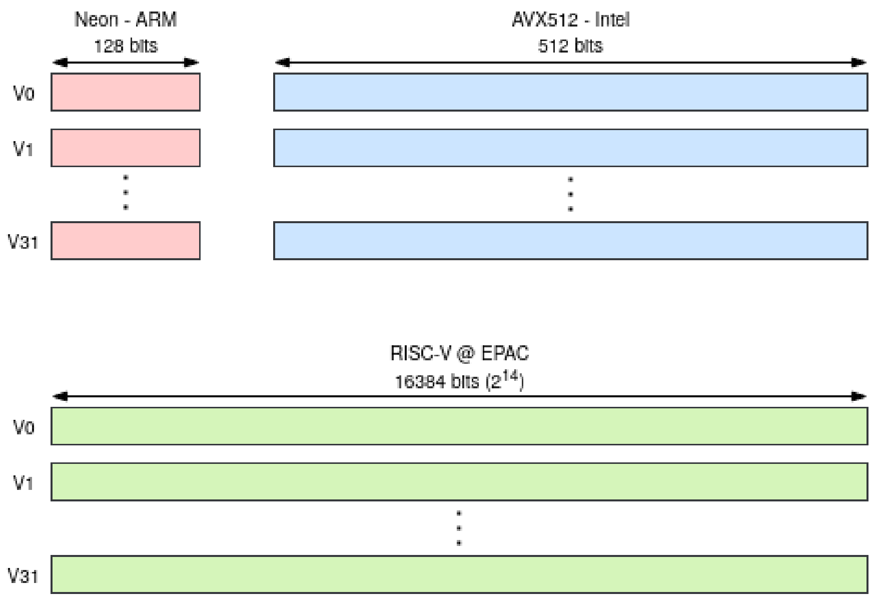


Figure 5: Vector register file comparison of different architectures. *Own compilation.*

As previously explained, parallelism can be obtained at data level with SIMD instructions. In order to be able to execute those SIMD instructions, an specific processing unit is required. In other words, the CPU needs an accelerator where SIMD instructions will be executed: the VPU.

There are different VPU architectures depending on the ISA specification used. In Figure 5 three vector register files are put into perspective. Depending on the implementation, each vector register file has a different size.

5.1.3 Clusters

A cluster is a computing system whose main objective is to increase its performance taking advantage of the number of compute nodes.

A generic cluster diagram can be seen in Figure 6. It shows that each node is composed by a CPU, an accelerator (graphic card or FPGA), and main memory. Nodes are connected through an interconnection network, which is also connected to data storage.

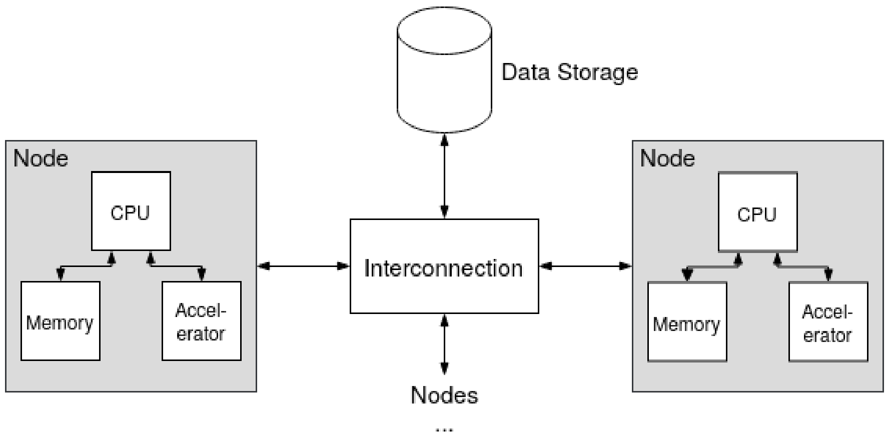


Figure 6: Generic cluster diagram. *Own compilation.*

HCA, named from *Heterogeneous Computer Architectures*, is the cluster that manages and contains the two partitions used for this project's scope: Arriesgado and Pickle.

They are managed through Slurm, a job scheduling system that lets users work in both partitions without conflict. [25]

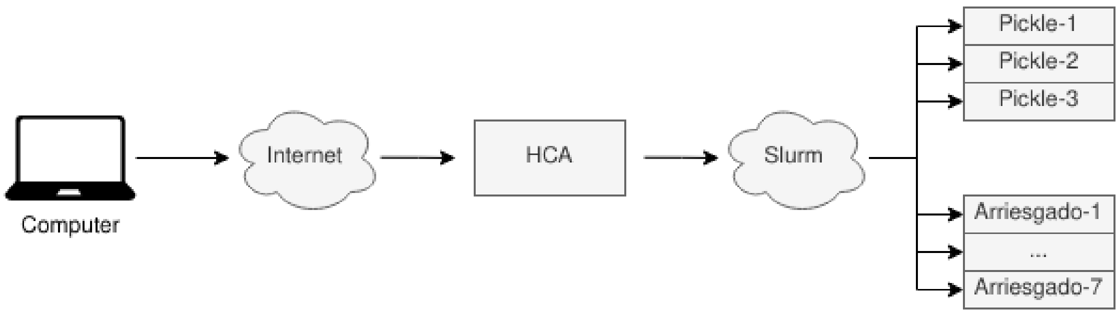


Figure 7: HCA. *Own compilation.*

The *Arriesgado* partition consists of seven Arriesgado nodes, each being a HiFive Unmatched processor with four scalar cores at 1Ghz. [26] They are used to compile and test binaries in native RISC-V architecture, avoiding cross-compilation. In other words, the Arriesgado partition is used in steps 1 and 2 explained in Figure 9 from subsubsection 5.2.2.

The *Pickle* partition has three nodes. Every node is composed of an x86 CPU and a VCU128 FPGA board connected through PCIe and Ethernet. The FPGA is reprogrammed through the x86 core, as well as loading the Linux image and starting a UART shell. The Pickle partition is used in step 3 explained in Figure 9 from subsubsection 5.2.2.

5.1.4 FPGA

An FPGA is a device with reconfigurable hardware: CLBs and programmable interconnections. Those components can be programmed using a Hardware Description Language (HDL), such as Verilog or VHDL. The RTL is designed and developed with HDLs.

The reprogrammability capacity makes FPGAs suitable for implementing custom RTL to accelerate specific parts of scientific applications. Another FPGA usage is to test and validate RTL for processors under development. Testing chips in FPGAs is crucial since it allows a quicker RTL and software testing rather than in a simulator. Moreover, the environment is closer to the final product: the chip and the software stack.

In the EPI project scope, FPGAs are used to test both the EPAC processor and the software stack being developed.

The FPGA used in this project is a Xilinx Virtex UltraScale+ HBM VCU128 Evaluation Kit. [21] An evaluation kit is a board that offers the following: hardware, IPs, design tools and pre-verified reference designs; in order to enable and ease development. [27]

An Intellectual Property (IP) is a verified RTL design provided by Xilinx that eases the usage of FPGA's elements.

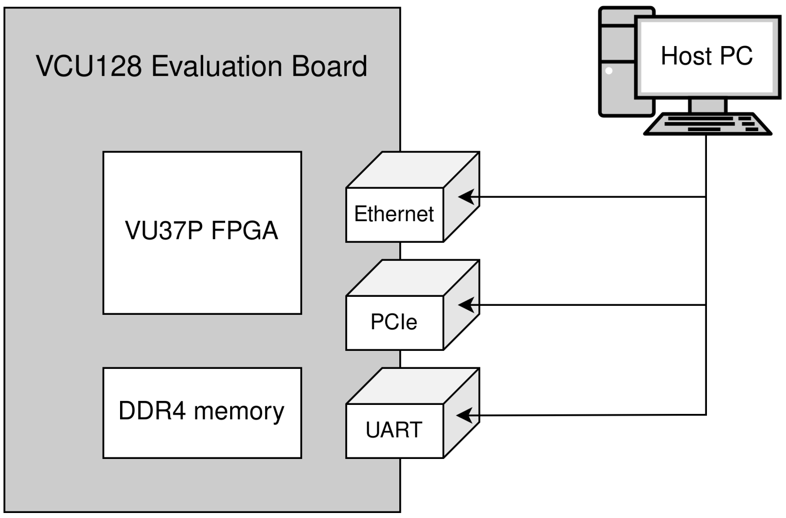


Figure 8: FPGA diagram. *Own compilation.*

The VCU128 board contains an Xilinx Virtex UltraScale+ VU37P HBM FPGA where the EPAC core is programmed.

Moreover, it has a DDR4 memory, where the Linux image is loaded. For writing in the DDR4 memory, the FPGA has a PCI Express port, and the Linux image is transferred from the host through it; see subsubsection 5.1.3 for more details.

Furthermore, it provides an Ethernet port, used to both establish SSH connection, and the UART port for the UART shell. UART is a communication protocol between devices. A UART shell uses the UART protocol in order to have an interactive session in the Linux image booted in the EPAC core programmed in the FPGA.

5.1.5 Bitstream

A bitstream is a file that contains all the configuration information of a particular FPGA, as well as all the resources to control and command the chip. In order to use an FPGA, it will be previously *programmed*, which essentially means to load a bitstream file to the FPGA.

The generation of a bitstream file requires a complex synthesis process, as the final design is based on heuristics due to the complexity of the algorithms required (NP-complete). The tool used for bitstream generation in the EPI project is Vivado, as the FPGAs used are Xilinx. Vivado is designed to work with the architecture of FPGAs designed by Xilinx. It has specific IPs for easily configuring all the FPGA aspects, such as the Ethernet or PCIe, rather than using the physical protocol. In other words, rather than directly using the phyisical layer of each component, hiding its complexity.

5.2 European Processor Initiative

The EPI is a European project that aims to design and develop the first European system-on-chip and accelerator processors for high-performance computing. The EPI accelerator (EPAC) is a RISC-V-based chip, including, among others, a RISC-V vector accelerator able to handle vector operands of up to 256 double-precision elements. The architecture behaves as a general-purpose computing platform and is supported by a complete system software tool-chain including a standard Linux, an LLVM-based compiler enabling intrinsics and auto-vectorization, and vectorized scientific libraries.

The BSC contributes to the EPAC architecture with the design of a Vector Processing Unit that support the V extension of the RISC-V Instruction Set Architecture (ISA). BSC also leads the development of the LLVM compiler for RISC-V supporting the EPAC vector accelerator including the VPU.

The main body of this project has been developed within BSC in the group developing the EPI Software Development Vehicles (SDVs). SDVs are software and hardware tools that allow experimentation with the EPAC architecture and the EPAC compiler. Even with little knowledge of the underlying architecture, scientists can compile and test their codes and collaborate with EPI experts in analyzing the outcome. The resulting study is a co-design effort providing feedback to the domain scientist who plans to efficiently run on long-vector architectures and architects and system software developers.

In this section, we will introduce the fundamental concepts that are required to understand the project's background and that are needed to achieve the project's objectives.

5.2.1 RISC-V Vector extension

As RISC-V is a modular ISA, the extension called "V" defines the architecture of the RISC-V vector unit. It is a vector length agnostic VPU.

The vector length is the register size. In a vector length agnostic VPU, the register size is not fixed by the architecture, but rather defined by the programmer.

The VPU for the EPAC processor is being developed in BSC. It is an accelerator and is the biggest part in the core and the FPGA, regarding area usage. The EPAC VPU is complaiant with the RISC-V "V" extension specification.

This accelerator is a crucial part in the EPI core, since it allows obtaining speedups that would be innaccesible only with the scalar core.

5.2.2 SDV environment

The Software Development Vehicles (SDV) environment consists of a set of tools and designs fundamentally developed by the EPI partners to ease and enable development of both hardware IPs and software applications.

An FPGA-based hardware platform, from now on refered as SDV@FPGA or FPGA-SDV, is provided for software and hardware development. The SDV@FPGA is used as an additional tool to test the EPAC processor, rather than exclusively using an RTL simulator.

An RTL simulator is designed to simulate the design to a very fine detail. This approach is useful to catch functional bugs at the logical level, but it is not suitable for executions of large programs such as an operative system.

As a reference, the process of booting a lightweight distribution of Linux in the FPGA design, takes up to 5min. In contrast, the simulation could take hours if not days.

Consequently, the usage of the SDV@FPGA does also drastically accelerates the process of debugging any malfunctioning feature of the EPAC core.

The recommended workflow for the SDV environment users, when testing applications, is founded on three basic steps, as seen in Figure 9.

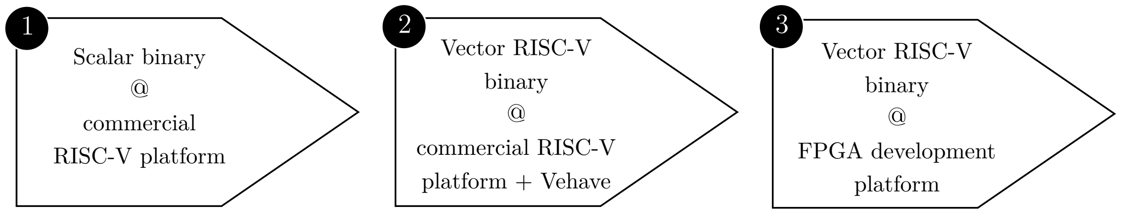

1
Scalar binary @ commercial RISC-V platform
2
Vector RISC-V binary @ commercial RISC-V platform + Vehave
3
Vector RISC-V binary @ FPGA development platform

Figure 9: Recommended workflow for binary testing. *Own compilation*

Initially, the application is ported to a commercial RISC-V platform. That application is now RISC-V compatible, but it still lacks vectorial instructions - it only contains scalar ones.

The second step is to vectorise the code. Once it has vectorial instructions, a binary is build with the LLVM compiler developed by BSC and executed through Vehave, a vector emulation software installed in all RSC-V commercial platforms. The simulation via Vehave allows for validating the user's codes.

In other words, Vehave allows the execution of binaries with vector instructions in a processor that can only execute scalar instructions, because it has no VPU.

Vehave emulates vector instructions via scalar instructions. Hence, the execution time is inaccurate, but the binary's execution behaviour is correct.

Finally, moving onto the FPGA development platform, the vectorised binary is executed in a RISC-V core supporting the "V" ISA extension.

5.2.3 EPI compiler

The programmer must be able to use the VPU. One possibility is inserting assembly inline, but it is not a comfortable way to program. To ease the generation of vectorial codes for the user, a custom compiler is being developed.

The EPI compiler is an LLVM-based compiler developed in BSC. It provides custom intrisics to use the EPI VPU, so the programmer can generate custom vector codes. Moreover, the compiler can autovectorise an scalar code.

5.3 Git repositories

A repository is a centralised storing site for code-related projects. Typically, a repository is managed with a version control software, such as Git.

The scope of this project is mainly encapsulated in three Git repositories, shown in Figure 10, within the EPI infrastructure.

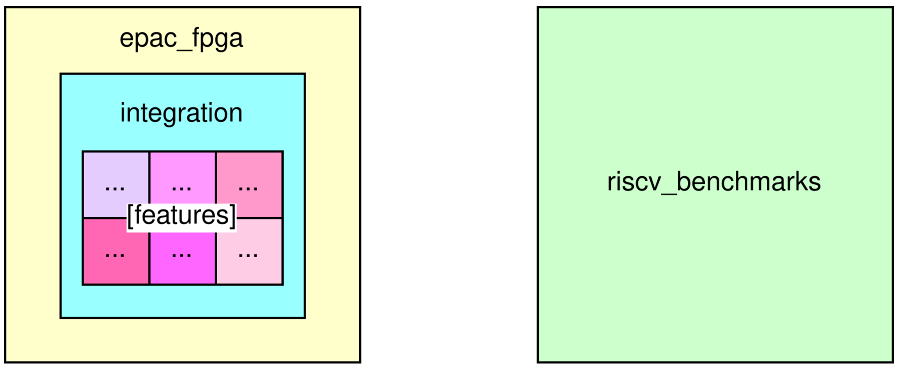


Figure 10: GitLab repositories diagram. *Own compilation.*

Each repository and its importance is explained below.

5.3.1 Integration

All the EPAC parts are contained in the *Integration* repository.

Each EPI partner is responsible for developing some of the core's features, such as L2 cache memory, the scalar core, or the VPU.

Every core parts is coded in a different repository. The Integration repository contains all the features via submodules, as can be seen in Figure 10. A submodule is a Git repository that acts as a subdirectory within another repository. All core elements are a submodule pointing to the corresponding repository.

5.3.2 EPAC_FPGA

The *EPAC_FPGA* repository is the one with the additional components needed to generate and use the EPI core in the FPGA.

Firstly, the RTL is needed in the repository to be able to use the main characteristics of the FPGA necessary to use the EPI core. Some of the main characteristics are the DDR4 memory, the PCIe and the Ethernet connection.

The information for Vivado's project to successfully generate a bitstream is also included. Additionally, the tools to ease the FPGA's usage, mainly to reprogram the FPGA, boot the Linux image and start a UART shell. Finally, the necessary files for generating Linux images are contained too.

This repository includes the *integration* repository as a submodule, as seen in Figure 10.

Moreover, this is the repository where the CI is implemented. This project will be mainly developed in this repository.

5.3.3 RISC-V Benchmarks

The *RISC-V Benchmarks* repository contains benchmarks' codes and makefiles for compilation available for any RISC-V processor.

The repository is divided between *microbenchmarks* and *HPC benchmarks*. The first ones are minimal benchmarks for testing specific features. The latter are more complex benchmarks.

6 Continuous Integration environment

A Continuous Integration setup is a fundamental tool in any large-scale project such as the European Processor Iniciative. The `epac_fpga` repository has a functional CI already implemented. As this repository is hosted in GitLab, the CI used is specific to it.

This final degree project will implement features on the last step in the `epac_fpga` CI pipeline: the `benchmarks` stage.

Throughout this section two basic aspects necessary for this work to be fully comprehended will be explained:

1. The structure the pipeline had prior to this work.
2. All the basic YAML concepts for programming a GitLab CI.

6.1 Initial pipeline structure

The initial structure of the CI is shown in Figure 11. Each stage and job is explained in the following subsections.

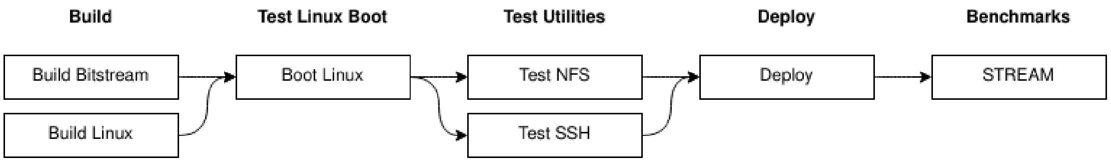


Figure 11: Initial CI pipeline. *Own compilation.*

6.1.1 Build stage

The first stage executed in the pipeline is the *Build*, as seen in Figure 11. Two main tasks occur in this stage, which are distributed in several jobs. One task is to build the SDV bitstream for the VCU128 board. The second task is to build four different Linux images compatible with the bitstream; and each image is built in a different job.

Both the bitstream and Linux images are the artifacts of their respective CI jobs and are used by the following pipeline's jobs.

Additionally, a set of reports are also produced as artifacts from the Build Bitstream job. Those reports are generated by Vivado during bitstream's synthesis, and offer additional information about the process.

The CI is automatically triggered every time a push is made. Not all commits modify the processor's RTL. Therefore, the bitstream does not need to be generated with every commit, only with the ones that modify the RTL. This bypass mechanism is implemented because the bitstream generation takes from 9 to 10 hours.

Any time a bitstream is built in a CI job, the resulting bitstream artifacts are saved in the GitLab's cache.

The cache is a simple method for saving and accessing artifacts from previous jobs, including jobs from previous pipelines. Accordingly, all subsequent pipelines will access the cache and download the last built bitstream, until a new push that modifies the RTL is made. The new triggered pipeline will produce a new bitstream, and upload it to the cache for future pipelines.

Caching the bitstream builds allows a huge save on both time and computational resources.

6.1.2 Test Linux stage

Following the Build stage, the Linux images are tested during the *Test Linux* stage. The testing consists in booting the Linux images in the FPGA 20 times each. The FPGA is previously programmed with the latest built bitstream.

In order to guarantee that the Linux image is booting correctly, the CI expects the command prompt to appear through UART shell. If there is at least one failed booting, the job fails for that specific Linux image.

6.1.3 Test Utilities stage

The *Test Utilities* stage tests two fundamental features for both the users and the following CI stages.

One the one hand, a job tests the SSH connection to the FPGA. It is a crucial feature in order to submit asynchronous jobs rather than having to use an interactive session through UART shell.

On the other hand, a job mounts a directory hosted in the x86 server via NFS. By having a mounted directory, all binaries hosted there are accessible. Otherwise, any binary that needed to be tested in the FPGA would have to be embedded in the Linux image. Each time a binary has to be embedded, the Linux image must be recompiled. Moreover, the amount of binaries per every Linux image is limited.

6.1.4 Deploy stage

The *Deploy* stage only takes place in pipelines triggered by a push into master branch. The bitstream, Linux images and bring-up tools are packed and uploaded to an FTP server accessible by all EPI partners. That package becomes the latest release.

6.1.5 Benchmarks stage

The initial setup only includes *Stream*, a benchmark for measuring sustainable main memory bandwidth, in the *Benchmark* stage.

Different Stream versions are executed, depending on the vector length and the number of computed elements: the first from 16 to 256, and the latter from 2048 (2^{11}) to 1048576 (2^{20}).

Each combination between vector length and computed elements is executed once. The output is printed to GitLab's job log, so no artifact is generated in this job.

6.2 GitLab's CI YAML

The GitLab's CI is programmed in YAML files. [28] YAML (*YAML Ain't Markup Language*) is a human-readable data serialization language.

The starting point in a GitLab's CI is the `.gitlab-ci.yml` file. Generally, this file contains the whole CI code in a simple implementation.

However, for CI implementations with relatively big scopes, the `.gitlab-ci.yml` file is used as the CI entry point, and additional YAML files may be used, following a file hierarchy. Instad of writing a monolithic `.gitlab-ci.yml`, the CI developer may use those additional YAML files for distributing CI stages and jobs across them. For instance, each file is associated with a particular stage in this project.

6.2.1 Job and stage creation

The most basic structure of a CI pipeline are jobs, which are generally grouped in stages such as in the `epac_fpga` repository. [29] The simplest implementation of a job is shown in Listing 1. For this example, the job's name is `job_has_a_name`, the stage's name where it belongs is `first_stage`, and the job will execute all the code lines starting with a hyphen after the `script:` declaration. In this case, it will print the message *Hello, X!*, being *X* the username specified by *$GITLAB_-USER_LOGIN*, a predefined variable that contains the username of the user who starts the job. [30]

```
job_has_a_name:
  stage: first_stage
    script:
        - echo "Hello, $GITLAB_USER_LOGIN!"
```

Listing 1: Example code of a single job.

6.2.2 Pipeline creation

The pipeline is created depending on the order the jobs are written in YAML files. [31] In other words, the first job in a YAML file will be the first job in the pipeline and the first one on its stage. Runners are distributed according to this order.

```
first_job:
  stage: first_stage
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

second_job:
  stage: first_stage
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

third_job:
  stage: second_stage
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

fourth_job:
  stage: third_stage
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"
```

Listing 2: Example code of multiple jobs across several stages.

To visualize a created pipeline, in Listing 2 a simple CI implementation with multiple jobs and stages can be observed, and the resulting pipeline is shown in Figure 12. All jobs of a given stage will run in parallel if enough runners are available and no dependencies encountered, but two stages will not be simultaneously executed unless specified otherwise.

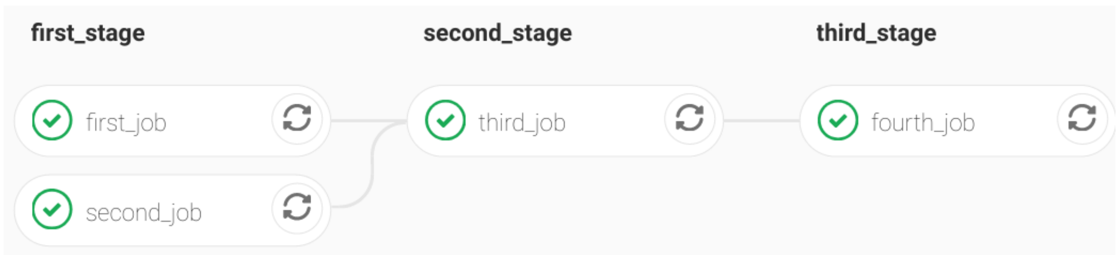
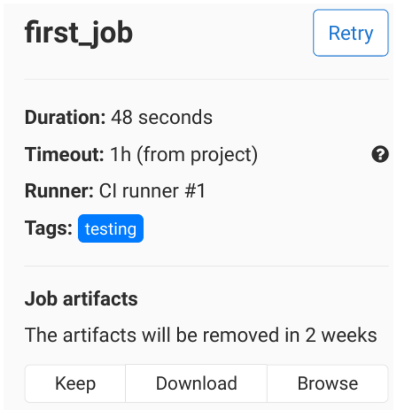


Figure 12: Pipeline created from the code in Listing 2. *Screenshot from GitLab.*

6.2.3 Job artifacts

Jobs can produce files or directories. In order for the user or other jobs to access those files, a job can store them as *artifacts*. [32]

For user access, artifacts can be accessed through GitLab's UI, as seen in Figure 13.



Duration: 48 seconds
Timeout: 1h (from project)
Runner: CI runner #1
Tags: testing

Job artifacts
The artifacts will be removed in 2 weeks

Figure 13: GitLab's UI. *Screenshot from GitLab.*

Users can click on the `Download` button and a ZIP file will download with all the artifacts from that specific job. Artifacts are deleted by default after three weeks since they were produced, but the time limit can be easily modified by developers.

The `artifacts` keyword is used to create job artifacts. In particular, inside the `artifacts` environment the `paths` keyword determines the paths of those files to be considered artifacts. The paths to files are relative to the repository directories.

A job that generates an artifact can be seen in Listing 3. The file `artifact.out` is produced during the job's script execution in the path `path/to/artifact/`.

```yaml
job_has_a_name:
  stage: first_stage
  artifacts:
    paths:
      - path/to/artifact/artifact.out
  script:
    - cd path/to/artifact
    - echo "Hello, $GITLAB_USER_LOGIN!" > artifact.out
```

Listing 3: Example code of an artifact being created.

6.2.4 Job dependencies

As previously explained, if enough runners are available, jobs will execute unless a dependency is encountered. Dependencies between jobs can either be related to other jobs or artifacts; see subsubsection 6.2.5 for the later one.

Job dependencies create directed acyclic graphs, being each graph's node a job.[1] Fundamentally, a job will not start until all the jobs stated under the `needs` keyword have finished.

If this keyword is used, parallelism between stages can occur, as shown in Listing 4. In this example, the `linux:rspec` job will start once the `linux:build` job finishes, even if the `mac:build` job has not finished. Therefore, it will not wait for the whole `build` stage to finish, but only for the jobs specified under the `needs` clause, unlike the `lint` job that will wait for all the jobs in previous stages.

```yaml
linux:build:
  stage: build
  script: echo "Building linux..."

mac:build:
  stage: build
  script: echo "Building mac..."

lint:
  stage: test
  script: echo "Linting..."

linux:rspec:
  stage: test
  needs: ["linux:build"]
  script: echo "Running rspec on linux..."
```

Listing 4: Example code of multiple jobs using `needs`. *GitLab's official webpage.* [2]

6.2.5 Artifact dependencies between jobs

In some situations, jobs may need resources produced as artifacts by other jobs in the pipeline. In that case, the `dependencies` field is used to list all the jobs whose artifacts are fetched.[33] The job will be able to access all the artifacts generated by the jobs enlisted, having direct access to the directories where those artifacts have been previously generated.

If a runner is assigned to a job with dependencies that are yet to be produced, the job will be stuck, and the runner re-assigned to another job, until dependencies

[1]See GitLab's official webpage on Directed Acyclic Graphs (DAG) in CI: https://docs.gitlab.com/ee/ci/directed_acyclic_graph/index.html

are entirely generated. Once all the required dependencies are available, the job's execution will resume again whenever a runner is available.

6.2.6 Artifact dependencies between pipelines: the cache

The *dependencies* option is only available in case an artifact has been produced in the same pipeline where another job needs it. In some situations, jobs may need access to resources produced in previous pipelines. In that case, two possibilities are available for CI developers.

The first one is to download those artifacts during script execution every time a job needs them. [34] If two or more jobs require the same artifacts, every job will download them.

The second method is to use the GitLab `cache`. [35] From one side, jobs that produce for future use in a different pipeline will store them at the `cache`. From the other side, the first job that needs an artifact stored at the `cache` requests and downloads it. Every future access within the pipeline will now have direct access to that artifact without the need to download it again.

Therefore, `cache` usage is faster than simple downloads whenever two or more jobs need a particular artifact from a previous pipeline.

6.2.7 Job templates

A template for a job can be created. Whenever a job is created using that initial job as a template, the `extends` keyword is used to refer to the template job. Using a template job is useful when a set of jobs have a shared structure, avoiding code replication.

A template is created by starting the job name with a dot. A job uses a template when the template name is specified under the `extends` field.

```yaml
.template_job:
  script: echo "Code from template"
  stage: test

job1:
  extends: .template_job
  after_script: echo "Code after template by job1"
```

```

job2:
  extends: .template_job
  after_script: echo "Code after template by job2"
```
Listing 5: Example code of a job template.

In Listing 5 an example with `extends` can be observed. The job `template_job` is used by both `job1` and `job2` as a template, and the template's code will be mixed with the jobs' code. As a result, both jobs will initially execute the template's `script` code. Afterwards, they will execute their own `after_script` code.[2]

The job's code would look as in Listing 6 if no templates were used and the jobs itself contained all the necessary code.

```
job1:
  extends: .template_job
  script: echo "Code from template"
  after_script: echo "Code after template by job1"

job2:
  extends: .template_job
  script: echo "Code from template"
  after_script: echo "Code after template by job2"
```
Listing 6: Example code of jobs from Listing 5 if no templates were used.

6.2.8 Variables

Environment variables can be set by developers using the `variables` keyword. [36] If variables are defined at the top level of the `.gitlab-ci.yml` they will be globally available for all jobs to use them.

Otherwise, they can be created in the job template and specified in the non-template job. Obviously, variables can also be created and specified directly in the job itself. In the first case, the variable will only be accessible by all the jobs under the same template, while in the latter case it will only be accessible by the job itself.

Additionally, variables can be set manually when triggering a pipeline. They will be globally available for all jobs.

[2]Besides the `script` keyword, both `before_script` and `after_script` exist. If used, their codes are respectively executed before and after the code within the `script` field.

```
variables:
  VAR_GLOBAL: "This variable is globally available"

job1:
  variables:
    VAR_JOB: "Only job1 can use this variable"
  script:
    - echo "$VAR_GLOBAL" and "$VAR_JOB"
```

Listing 7: Example code of variable creation and access.

In Listing 7 an example of two variables with different availability possibilities is shown. `VAR_GLOBAL` is a globally accessible variable, while `VAR_JOB` is a variable whose usage is limited to `job1` job.

7 Implementation

The following sections explain all the work related to the code implementation throughout this final degree project, divided among the different task groups specified in the project planning.

7.1 Benchmark's job refactor

Initially, the GitLab runners executed benchmark jobs in the FPGA as soon as the pipeline dependencies were satisfied. Furthermore, the infrastructure was at an early development phase, with no resource management system - like Slurm.

Both circumstances altogether resulted in GitLab runners kicking any user that was using the FPGA at the moment job execution started. This situation led to the incorporation of Slurm. Consequently, as a first task, this final degree project aimed to refactor the way previous developers originally programmed the benchmark job for adapting it to the introduction of Slurm.

7.1.1 Initial status

The `Benchmarks` stage contained a single benchmark executed in the stage's only job: **Stream**. Two files formed the whole stage: the `benchmarks.yml` and the `stream.sh`. The first one contained all the stage's definition in YAML and the bash code for preparing the FPGA environment, while the second one strictly included the code to execute the Stream benchmark.

In order to get the FPGA ready for the benchmark's execution, the job did several tasks during `before_script` in `benchmarks.yml`. It reprogrammed the FPGA, booted the Linux image, and mounted the shared directory that contained the benchmark binary.

```yaml
.benchmark:
  stage: benchmarks
  tags:
    - fpga
    - vivado
  variables:
    GIT_SUBMODULE_STRATEGY: none
```

```
8    needs: [build_bitstream, build_linux.CI-testing-sdv3-minimal-network-new, utilities_linux.ssh, utilities_linux.nfs]
9    dependencies:
10     - build_bitstream
11     - build_linux.CI-testing-sdv3-minimal-network-new
12   before_script:
13     - ./tools/reprogram-fpga
14     - ./tools/boot-linux-remote-and-check
15     - ./tools/ssh-cmd root@10.0.0.2 "mkdir /epi-shared && mount -tnfs4 10.0.0.1:/scratch/epi-shared /epi-shared"
16
17   stream:
18     extends: .benchmark
19     script:
20     - ./ci/benchmarks/stream.sh
```

Listing 8: Original code of `benchmarks.yml`.

The Stream job created in `benchmarks.yml` executed the file `stream.sh` during its `script` phase, as can be seen in line 20 from Listing 8.

The Stream binary accepts two parameters: the vector length (*VL*) and the problem size (*SIZE*). The `stream.sh` script executed every combination between *VL* and *SIZE* ten times each.

The binary was executed through ssh, storing both the exit code and the benchmark's output. In case any execution timed out (timeout limit at 30 seconds) or returned a non-zero exit code, the script booted the Linux image again.

If successful, it would mount the shared directory and continue normal execution. Otherwise, it would reprogram the FPGA and reboot the Linux image afterwards. Once again, it would keep normal execution if successful, but finish the script execution if the second Linux booting failed.

This whole process was due to some instabilities with the FPGA at the moment of the implementation of this stage. By the time this final degree project started, the EPI developers had already solved these instabilities.

Once a successful binary execution finished, it parsed the output and printed it in GitLab's log.

7.1.2 Developed solution

Besides the need of improvement on the way the `Benchmark` stage was programmed, the addition of Slurm to the Pickle nodes implied that the benchmark's execution had to be modified. The Slurm command `srun` has been used to queue the benchmark launch script with a limited execution time of one hour, as shown in line 17 from Listing 9.

```
1  .benchmark:
2    stage: benchmarks
3    tags:
4      - fpga
5      - vivado
6    variables:
7      GIT_SUBMODULE_STRATEGY: none
8    timeout: 2h
9    needs: [build_bitstream, build_linux.CI-testing-sdv3-minimal-
       network-new, utilities_linux.ssh, utilities_linux.nfs]
10   dependencies:
11     - build_bitstream
12     - build_linux.CI-testing-sdv3-minimal-network-new
13
14 stream:
15   extends: .benchmark
16   script:
17     - srun --job-name=CI-Benchmark-STREAM --time=01:00:00 --
       partition=fpga ./ci/benchmarks/stream/run.sh | tee benchmark-
       stream.out
18   artifacts:
19     when: always
20     paths:
21       - ./benchmarks-outputs/stream
```

Listing 9: Code developed of `benchmarks.yml`.

Besides the timeout specified by the `srun` command, a job timeout of two hours has been defined, as seen in line 8 from Listing 9. In other words, the job will have a maximum of two hours since its creation to finish all the subsequent scripts and tasks, otherwise it will abruptly finish execution and fail.

Once the `srun` command allocates resources for the Slurm job executing the `srun.sh` script, it will have a maximum of one hour to finish, always within the two hours limited by the pipeline job.

Additionally, all the content within the `before_script` environment in `benchmarks.yml` is removed.

The necessary steps for preparing the FPGA for benchmark execution have been moved to an independent file, named `run.sh`, whose code is in Listing 10. That file also contains the launch of the benchmark script to execute the benchmark, line 20 of Listing 10.

```
1  #!/bin/bash -e
2  
3  source /etc/profile.d/lmod.sh
4  module purge
5  module load vivado/2020.1
6  
7  export LC_ALL=en_US.UTF-8
8  
9  export SDV_BITSTREAM=./build/bitstream/epac_core.bit
10 export SDV_LINUX_IMAGE_PATH=./build/linux/
11 export SDV_PCIE_HOST=`hostname`
12 export SDV_REBOOT=""
13 
14 ./tools/reprogram-fpga
15 ./tools/boot-linux-remote-and-check
16 ./tools/ssh-cmd root@10.0.0.2 "mkdir /shared && mount -tnfs4 10.0.0.1:/scratch/shared/gitlab-runner/benchmarks /shared"
17 
18 mkdir -p ./benchmarks-outputs/stream
19 ./tools/scp-cmd ./ci/benchmarks/stream/stream.sh root@10.0.0.2:/root/.
20 ./tools/ssh-cmd root@10.0.0.2 "/root/stream.sh" | tee ./benchmarks-outputs/stream/stream.csv
```

Listing 10: Code developed of `run.sh`.

The usage of artifacts has been added. The benchmark’s output is stored in the `benchmarks-outputs/` directory, in a CSV file named `stream.csv`.[3]

The launch of the benchmark script in `run.sh` redirects the benchmark’s output to that file through `tee` command. This file will be the job’s artifact, as seen in line 20 from Listing 10.

```
#!/bin/bash

BENCHMARKS_DIR=/shared
```

[3]CSV: A *Comma-Separated Values* file is a text file that uses commas to stablish data columns and lines as data rows.

```
STREAM_BINARY=${BENCHMARKS_DIR}/stream/

echo "Size VL CopyBW ScaleBW AddBW TriadBW"

for VL in 16 32 64 128 256; do
  for SIZE in 2048 4096 8192 16384 32768 65536 131072 262144
   524288 1048576; do

    ${STREAM_BINARY}/stream-${SIZE}_elems-${VL}_vl > output.txt

    # Parsing the output
    copy_bw=`grep Copy: output.txt | awk '{print $2}'`
    scale_bw=`grep Scale: output.txt | awk '{print $2}'`
    add_bw=`grep Add: output.txt | awk '{print $2}'`
    triad_bw=`grep Triad: output.txt | awk '{print $2}'`

    echo "${SIZE} ${VL} ${copy_bw} ${scale_bw} ${add_bw} ${
   triad_bw}"

  done
done

rm output.txt
```

Listing 11: Code developed of `stream.sh`.

To sum up, the three resulting files have specific functionalities.

Firstly, the file `benchmarks.yml` does only manage the job's creation and queueing benchmark scripts to the Slurm queues.

Secondly, the `run.sh` handles everything related to the configuration of the FPGA environment.

Finally, the file `stream.sh` does only have the code for executing the Stream benchmark.

The objective of splitting the whole CI job has been the modularisation of work to ease both debugging and the addition of new benchmarks.

7.2 Addition of new benchmarks

At this phase, the Benchmark's pipeline stage had a single benchmark in a single job: Stream. As previously explained, a fundamental interest behind this final degree project has been to ensure the CI detects as many deficiencies as possible in the core's design being developed by the EPI team. Using only one benchmark does not fully fulfil this objective.

A set of benchmarks has been selected from all the RISC-V compatible benchmarks available at the `risc-v-benchmarks` repository at that moment.

The selection has been made according to the guidance of several BSC team members and some EPI partners with a high level of insight, as they have previously been involved in both designing and debugging EPAC core elements.

7.2.1 Selected benchmarks

All the benchmarks selected are briefly explained in this section.

Firstly, three benchmarks strictly test the load and store part of the VPU. It is a vital part since its latency and functionality are essential in many applications. They all stress the Network on Chip (NoC), home nodes and the communication between the scalar core and the VPU.

- **Buffcopy Unit**: It only loads and stores elements in contiguous locations into and from the vector unit.

  It uses the `vle.v` and `vse.v` RISC-V vector instructions.

- **Buffcopy Strided**: It only loads elements separated by a constant stride value into the vector unit. A stride is a constant number of bytes between one desired element and the next desired one.

  It uses the `vls.v` and `vss.v` RISC-V vector instructions.

- **Buffcopy Indexed**: It only loads elements indexed in a vector into the vector unit, also known as gather. A vector gather instruction consists in arranging different elements of a vector separated by a non-constant distance, an example is offered in Figure 14.

  It uses the `vlx.v` and `vsx.v` RISC-V vector instructions.

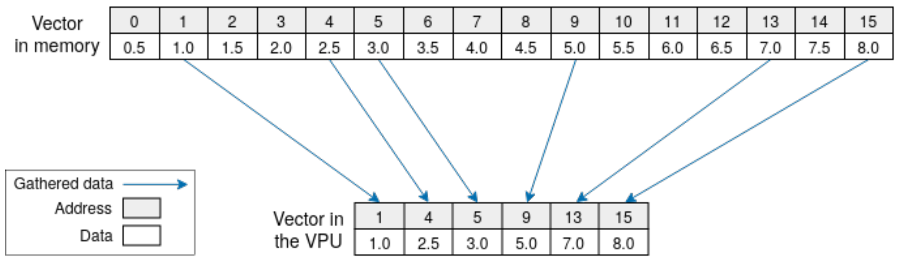


Figure 14: Gather instruction diagram. *Own compilation.*

The last benchmark whose purpose is to stress a single processor part is FMAS.

- **FMAS**: It is an algorithm that executes n FMA (*Fused Multiply-Accumulate*) instructions, floating-point multiplications and sums, without dependencies and measuring the time it takes.

  It tests the VPU floating point arithmetic.

  It is useful to know the maximum Flops/cycle a specific processor can compute.

The following benchmarks are algorithms that are part of scientific applications. They stress more than one processor part at once.

- **Jacobi 2D**: It is an iterative method to solve a system of n linear equations in n unknowns.

  It tests the NoC, the VPU's arithmetic unit and the communication between the scalar core and the VPU.

- **FFT (Fast Fourier Transform)**: Custom FFT vector implementation developed in BSC. An FFT benchmark implements an algorithm to compute the Discrete Fourier Transform (DFT).

  It tests the NoC, the VPU's arithmetic unit and the communication between the scalar core and the VPU.

  It is a widely used algorithm in several disciplines, such as engineering or science, because it converts a signal in its original domain to the frequency domain.

- **SpMV (Sparse Matrix-Vector)**: It is a kernel that performs sparse matrix-vector multiplications.

It tests the NoC, the VPU's arithmetic unit and the communication between the scalar core and the VPU.

SpMV is the most significant part of HPCG (*High Performance Conjugate Gradients*). [37] In HPC it acts as an alternative to HPL (*High Performance Linkpack*) to classify the TOP500 supercomputer list.[4]

Notice that all benchmarks indirectly test the scalar core since all the code surrounding the vectorial code is scalar. Therefore, the scalar core is tested in every benchmark but in a non-strict way.

Furthermore, the already implemented **Stream** benchmark has been kept. It performs load and store operations with a size bigger than the processor's cache size. It tests the NoC and the communication between the scalar core and the VPU.

7.2.2 Scripts generalization

With the addition of several benchmarks in mind, during the early development of this task, it became clear that the refactor previously implemented could be improved. The main two reasons are that each benchmark needed a separated `run.sh` script, and the `srun` command in the GitLab CI job had a custom path.

Two simple solutions were designed: to generalize the `run.sh` script and create a GitLab CI template.

As a consequence, instead of having a `run.sh` file for each job, the file has been slightly modified in order to be used by all the `[benchmark].sh` scripts, as the necessary process to prepare the FPGA environment for benchmark execution is shared among all benchmarks.

In order to allow this, the benchmark name is passed as an argument into `run.sh` by every job in `benchmarks.yml`, as seen in Listing 12.[5]

```
#!/bin/bash -e
    ⋮
mkdir -p ./benchmarks-outputs/$*
```

[4]TOP500 offical webpage: https://www.top500.org/.

[5]In a Bash script, the parameter $* is used to access the arguments passed to the script in its invocation.

```
./tools/scp-cmd ./ci/benchmarks/$*/$*.sh root@10.0.0.2:/root/.

./tools/ssh-cmd  root@10.0.0.2 "/root/$*.sh" > ./benchmarks-
    outputs/$*/$*.csv
```

Listing 12: Code of the generic `run.sh` script. Unchanged lines have been omitted in this listing, check Listing 10 to see them.

Furthermore, the generic `run.sh` script is now called in the `script` area shared among all benchmark jobs and passing as parameter the variable `BENCHMARK_NAME`, as seen in line 14 from Listing 13.

```
1  stage: benchmarks
       ⋮
13 script:
14   - srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --time=01:00:00
       --partition=fpga ./ci/benchmarks/run.sh ${BENCHMARK_NAME} |
      tee benchmark-${BENCHMARK_NAME}.out
15 artifacts:
16   when: always
17   paths:
18     - ./benchmarks-outputs/${BENCHMARK_NAME}
19
20 stream:
21   extends: .benchmark
22   variables:
23     BENCHMARK_NAME: stream
```

Listing 13: Code of the `benchmarks.sh` script adapted for a generic `run.sh`. Unchanged lines have been omitted in this listing, check Listing 9 to see them.

Every GitLab CI job defines the `BENCHMARK_NAME` variable with the benchmark's name being executed, as seen in line 23 from Listing 13.

7.2.3 Binary generation

After choosing the benchmarks that would be added to the pipeline stage, a binary had to be generated and their correct behaviour checked.

The steps followed to test each benchmark's binary are the ones defined in Figure 9 from subsubsection 5.2.2.

Firstly, the scalar version was compiled with the EPI compiler following the instructions provided by BSC team members and executed in Arriesgado. This scalar version would be used for future comparisons, as it provided the correct benchmark's output.

Secondly, the vectorized version was generated with the EPI compiler and tested with Behave in Arriesgado to check that the vector implementation was correct, comparing the output with the scalar one.

Thirdly, the vectorized binary was tested in the FPGA, checking that the output matches the scalar one.

After those verifications, the next step was to create a custom execution script with the necessary parameters and output parsing for each benchmark.

All the compiled and tested binaries were deployed in the shared folders of the HCA server, available by all FPGAs. This way, all binaries will be accessible during the pipeline execution.

7.2.4 Generic benchmark implementation

Every benchmark added to the pipeline requires a new job. Thus, the Stream benchmark implementation has been used as a reference for all the outcoming benchmarks.

Every benchmark job will have its own script file, named `[benchmark].sh`, being *[benchmark]* the benchmark's name. Each benchmark script has different input parameters and output values, but all share the same structure and design. They contain different loops to generate combinations between the input parameters, performing five iterations for each combination.

The output is parsed at each iteration, to store all the measures for every executed combination in a file named `[benchmark_name].csv`, including the execution parameters of each iteration. The parsed output will be stored in that file as programmed in the `run.sh` script, as seen in line 20 from Listing 10.

The characteristics of each benchmark implementation are explained in the following sections.

7.2.5 Buffcopy unit

The **Buffcopy unit** benchmark has different binaries depending on the `pipeline` value: 1, 2, 4 or 8. It defines *pipeline* consecutive load instructions launched and then another *pipeline* store instructions. For instance, a $pipeline = 2$ is translated into two load instructions, followed by two store instructions, then another two loads and two stores, and so on.

The three input parameters that all Buffcopy unit binaries need are explained in Table 11.

Input Parameters	Meaning	Values
`VL`	Vector Lenght	$2^{[4,8]}$
`elements`	Problem size	$2^{[11,16]}$
`ntimes`	Repetition of the main loop	$2^{[8,4]}$

Table 11: Buffcopy unit input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./BuffCopyUnit_pipelining<pipeline>_binary <elements> <VL> <ntimes>
```

An example of an output can be seen in Listing 14. It shows the execution result with a pipelining of 1, 2048 elements, a vector length of 16 and a ntimes of 256.

The meaningful output is in lines 7 and 8 of Listing 14.

```
1 # ./BuffCopyUnit_pipelining1 2048 16 256
2 RVL=16 GVL=16
3 SYCALL TELLS = -1
4 retries: 0
5 SYCALL TELLS = -1
6 retries: 0
7 header: Time(s) cycles Time_min(s) cycles_min BW(MB/s) BW(MB/s)(
    measure cycles @50MHz) BW_min(MB/s) BW_min(MB/s) (measure
    cycles @50MHz)
8 results: 0.03826300 1911762 0.00010400 4627 109.61774926
    109.69733680 315.07692663 354.09552626
```

Listing 14: Buffcopy unit output example.

It contains, among other measures: the execution time, the overall number of cycles, the execution time of the fastest iteration, and the number of cycles of the same iteration.

```bash
#!/bin/bash

BENCHMARKS_DIR=/shared
BUFFCOPY_BINARY=${BENCHMARKS_DIR}/buffcopy-unit

# Printing headers for output columns:
echo "Pipeline,VL,Elem,NTimes,Time(s),Cycles,TimeMin(s),CyclesMin,BW(MB/s),BWMin(MB/s)"

for pipeline in 1 2 4 8; do
  for VL in 16 32 64 128 256; do
    elements=2048
    ntimes=256
    for i in `seq 6`; do
      for j in `seq 5`; do
        # Executing the binary,
        # output is redirected to a text file:
        ${BUFFCOPY_BINARY}/BuffCopyUnit_pipelining${pipeline} ${elements} ${VL} ${ntimes} > output.txt
        # Output parsed and printed,
        # with the pipeline and iteration parameters:
        echo "${pipeline},${VL},${elements},${ntimes},`grep results: output.txt | awk '{print $2","$3","$4","$5","$6","$8}'`"
      done
      # Decreasing ntimes while increasing elements and VL:
      elements=$(( $elements * 2 ))
      ntimes=$(( $ntimes / 2 ))
    done
  done
done

# Deleting temporary output file text:
rm output.txt
```

Listing 15: Code of `buffcopy-unit.sh`.

7.2.6 Buffcopy strided

The **Buffcopy strided** benchmark has a single binary with two possible input parameters: `elements` and `ntimes`, both explained in Table 12.

Input Parameters	Meaning	Values
`elements`	Problem size	$2^{[11,16]}$
`ntimes`	Repetition of the main loop	$2^{[3,9]}$

Table 12: Buffcopy strided input parameters and their corresponding values.

The benchmark has pre-defined values of 256 for the vector length, and 64 stride bytes. As the program works with doubles, whose size is 8 bytes, it means that the stride is 8 elements ($\frac{64 \text{ B stride}}{8 \text{ B double}} = 8$ elements).

The corresponding line and argument order to execute this benchmark is:

```
./BuffCopyStrided_binary <elements> <ntimes>
```

An example of an output can be seen in Listing 16. It shows the execution result with 2048 *elements* and a *ntimes* of 8.

```
# ./BuffCopyStrided 2048 8
maxvl = 256
Array size=2048(elements), array_size=0.015625(MiB)
accum[0]=16384.000000
header:        Time(s) BW(MB/s)
result:        0.00041800      313.56937008
```

Listing 16: Buffcopy strided output example.

```
#!/bin/bash

BENCHMARKS_DIR=/shared
BUFFCOPY_BINARY=${BENCHMARKS_DIR}/buffcopy-strided

echo "Elements,Ntimes,Time(s),BW(MB/s)"

for elements in 2048 4096 8192 16384 32768 65536; do
  for ntimes in 8 16 32 64 128 256 512; do
    for i in `seq 5`; do
      ${BUFFCOPY_BINARY}/BuffCopyStrided ${elements} ${ntimes} >
   output.txt
      echo "${elements},${ntimes},`grep result: output.txt | awk
   '{print $2","$3}'`"
    done
  done
done

rm output.txt
```

Listing 17: Code of `buffcopy-strided.sh`.

7.2.7 Buffcopy indexed

The **Buffcopy indexed** benchmark has three possible input parameters: `elements`, `strideB` and `ntimes`, all three explained in Table 13.

Input Parameters	Meaning	Values
`elements`	Problem size	$2^{[11,16]}$
`strideB`	Used in the index vector generation	$2^{[3,9]}$
`ntimes`	Repetition of the main loop	256

Table 13: Buffcopy indexed input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./BuffCopyIndexed_binary <elements> <strideB> <ntimes>
```

An example of an output can be seen in Listing 18. It shows the execution result with 2048 *elements*, a *strideB* of 8 and a *ntimes* of 256.

```
1 # ./BuffCopyIndexed 2048 8 256
2 stride_bytes=8 stride_pow=3 stride_elements=1
3 effective_elements=2048(elements), allocated_array=0.015625(MiB)
4 accum[0]=2048.000000
5 header:       Time(s) BW(MB/s)
6 result:       0.01775900      236.17905240
```

Listing 18: Buffcopy indexed output example.

The important measurements printed in the output are the time (in seconds) and the bandwidth (in megabytes per second). Both measures are printed after binary execution as seen in lines 5 and 6 from Listing 18. The output is parsed in line 13 from `buffcopy-indexed.sh` script, whose code is in Listing 19.

```
1  #!/bin/bash
2
3  BENCHMARKS_DIR=/shared
4  BUFFCOPY_BINARY=${BENCHMARKS_DIR}/buffcopy-indexed
5
6  echo "Elements,StrideBytes,NTimes,Time(s),BW(MB/s)"
7
8  ntimes=256
9  for elements in 2048 4096 8192 16384 32768 65536; do
10   for strideB in 8 16 32 64 128 256 512; do
11     for i in ‘seq 5‘; do
```

```
      ${BUFFCOPY_BINARY}/BuffCopyIndexed ${elements} ${strideB} ${ntimes} > output.txt
      echo "${elements},${strideB},${ntimes},`grep result: output.txt | awk '{print $2, $3}'`"
    done
  done
done

rm output.txt
```

Listing 19: Code of `buffcopy-indexed.sh`.

7.2.8 FMAS

The **FMAS** benchmark binary has two input parameters: `VL` and `NUM_LOOPS`, both explained in Table 14.

Input Parameters	Meaning	Values
`VL`	Vector length	0, 32, 256
`NUM_LOOPS`	Repetition of the main loop	64

Table 14: FMAS input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./FMAS_binary <VL> <NUM_LOOPS>
```

An example of an output can be seen in Listing 20. It shows the execution result with a *VL* of 1 and a *NUM_LOOPS* of 64.

```
/epi-shared/rafel # ./FpuMicroKernel 1 64
requested vl=1, granted vl=1
SYCALL TELLS = -1
header: cycles  Flop/c      cycles vpu      Flop/cvpu
result: 39700   0.19989924       0          inf
```

Listing 20: FMAS output example.

The valuable output parameters are the total amount of cycles and the performance in floating point operations per cycle, those are the parsed parameters in line 13, in the `fmas.sh` script shown in Listing 21.

```
1 #!/bin/bash
```

```

BENCHMARKS_DIR=/shared
FMAS_BINARY=${BENCHMARKS_DIR}/fmas/fmas

NUM_LOOPS=64

echo "VL,NLoops,Cycles,Flop/c"

for i in ‘seq 0 32 256‘; do
  for j in ‘seq 5‘; do
    ${FMAS_BINARY} ${i} ${NUM_LOOPS} > output.txt
    echo "${i},${NUM_LOOPS},‘grep result output.txt | awk ’{print $2, $3}’‘"
  done
done

rm output.txt
```

Listing 21: Code of `fmas.sh`.

7.2.9 Jacobi 2D

The **Jacobi 2D** benchmark has two input parameters: `N` and `ITER`, both explained in Table 15.

Input Parameters	Meaning	Values
`N`	Problem size	$2^{[6,10]}$
`ITER`	Repetition of the main loop	8

Table 15: Jacobi-2D input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./Jacobi2D_binary <N> <ITER>
```

An example of an output can be seen in Listing 22. It shows the execution result with an N of 64 and $ITER$ of 8.

```
# ./jacobi2d_binary 64 8
time(s): 0.005487
```

Listing 22: Jacobi-2D output example.

The benchmark's output offers a single measurement, the execution time in seconds. It is parsed in line 12 from the `jacobi-2d.sh` shown in Listing 23.

```
#!/bin/bash

BENCHMARKS_DIR=/shared
JACOBI_BINARY=${BENCHMARKS_DIR}/jacobi-2d/jacobi2d

ITER=8

echo "N,Iter,Time"

for N in 64 128 256 512 1024; do
  for i in `seq 5`; do
    ${JACOBI_BINARY} ${N} ${ITER} | awk '{print $2}' | xargs echo
   ${N} ${ITER};
  done
done

rm output.txt
```

Listing 23: Code of `jacobi-2d.sh`.

7.2.10 FFT

The **FFT** benchmark has two input parameters and three flags, the first one being compulsatory and the last two optional: `N`, `REP`, `fftp/fftw/both`, `cycles` and `check`, all explained in Table 16.

Input Parameters	Meaning	Values
`N`	Problem size	$2^{[6,13]}$
`REP`	Repetition of the main loop	1
`fft flag`	FFT version to be executed	fftp/fftw/both
`cycles flag`	Measure cycles instead of microseconds	cycles
`check flag`	Check the FFTP results against the FFTW[6]	check

Table 16: FFT input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./fft_binary <N> <REP> <fft flag> <cycles flag> <check flag>
```

[6]The FFTW, stands for *Fastest Fourier Transform in the West* and is the standard implementation of FFT used in science and HPC.

The `check` flag enables a comparison between the results of the vectorized FFT against a scalar FFTW. If there is an above-threshold difference, it identifies which elements of the transform do not match.

Once the benchmark execution finishes, if the `check` flag is enabled, it prints `Check:Correct` when there are no discrepancies or `Check:Incorrect` otherwise.

An example of an output can be seen in Listing 24. It shows the execution result with an *N* of 64, a *REP* of 1, the `fftp` and `check` flags set.

```
# ./runfftp-fftw-static_intrinsics 64 1 fftp check
fftp_plan_time: 200
fftp_exe_time: 87
Check: Correct
accum_error: 0.00000000
avg_error: 0.00000000
max_error: 0.00000000
```

Listing 24: FFT output example.

Line 12 from `fft.sh` script, shown in Listing 25, offers the command used for executing the binary in each iteration. Differently than the example provided in Listing 24, the pipeline job execution uses the `cycles` flag, as it provides more reliable information than the time measurement offered otherwise.

Furthemore, the `check` flag is also enabled. In case any execution offers an *Incorrect* output, that information is parsed to the job's artifact.

The output parameter parsed is the total execution cycles, as shown in line 15 from Listing 25.

```
1  #!/bin/bash
2
3  BENCHMARKS_DIR=/shared
4  FFT_BINARY=${BENCHMARKS_DIR}/fft/fft
5
6  REP=1
7
8  echo "N,Rep,ECycles"
9
10 for N in 64 128 256 512 1024 2048 4096 8192; do
11   for i in ‘seq 5‘; do
12     ${FFT_BINARY} ${N} ${REP} fftp cycles check > output.txt
13     if grep -q "Correct" output.txt; then
14       # In case output is correct, parses it:
      echo ${N} ${REP} `awk '/fftp_exe_cyc/ {print $3}' output.txt
   ` | sed 's/ /,/g'
    else
      # In case output is not correct:
      echo "${N},${REP} is INCORRECT"
    fi
  done
done

rm output.txt
```

Listing 25: Code of `fft.sh`.

7.2.11 SPMV

The **SPMV** benchmark has two input parameters: `VL` and the input matrix, both explained in Table 17.

Input Parameters	Meaning	Values
`VL`	Vector length	256
`MTX`	Matrix used	-

Table 17: Jacobi-2D input parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./spmv_binary <VL> -m <input matrix>
```

An example of an output can be seen in Listing 26. It shows the execution result with a *VL* of 256 and a matrix named `Matrix4.mtx`, located in `spmv/spmv_-inputs/Matrix_4`.

```
# ./spmv_sellcslib_static 256 -m spmv/spmv_inputs/Matrix_4/
   Matrix_4.mtx
  shift: 6
{
  "Benchmark": "SPMV EPI",
    "Algorithm version": "sellcslib",
    "Verification test": "Pass",
    "Number of threads": 1,
    "Problem summary": {
      "Input name": "Maragal_4.mtx",
      "Matrix Num. Rows": 1964,
      "Matrix Num. Columns": 1034,
      "Total non-zero elements": 26719,
      "Non-zero elements per row": 13,
      "Num. averaged iterations": 10
    },
    "Memory statistics": {
      "Total memory allocated [MB]": 0.000000,
      "Reference version read BW [GB/s]": 0.064049,
      "Reference version read WR [GB/s]": 0.002271,
      "sellcslib version read BW [GB/s]": 0.000000,
      "sellcslib version write BW [GB/s]": 0.000000
    },
    "Performance statistics": {
      "Time allocating and loading data [s]": 1.915078,
      "Time converting to sellcslib format [s]": 0.052991,
      "Reference version execution time [s]": 0.006920,
      "Reference version GFLOPS/s": 0.007722,
      "sellcslib version execution time [s]": 0.005182,
      "sellcslib version GFLOPS/s": 0.010313
    },
    "Version specific stats": {
      "Row order sigma window": 16384,
      "Task Size": 16
    }
}
```

Listing 26: SPMV output example.

The output offers information about the execution configuration, as well as memory and performance statistics measured during the benchmark's execution. The two most insightful output parameters are the total execution time (in seconds) and the average GFLOPS/s. The parsing can be observed in lines 14 and 15 from the `spmv.sh` script, seen in Listing 27.

```
1  #!/bin/bash
2  
3  BENCHMARKS_DIR=/shared
4  SPMV_BINARY=${BENCHMARKS_DIR}/spmv/spmv
5  
6  VL=256
7  MTX=${BENCHMARKS_DIR}/spmv/spmv_inputs
8  
9  echo "VL,Mtx,Time,GFlops/S"
10 
11 for i in `seq 6 10`; do
12   for j in `seq 5`; do
13     ${SPMV_BINARY} -v ${VL} -m ${MTX}/cage${i}/cage${i}.mtx >
   output.txt
```

```
    time=`grep "sellcslib version execution time" output.txt | awk
    '{print $6+0}'`
    gflops=`grep "sellcslib version GFLOPS/s" output.txt | awk '{
   print $4}'`
    echo "${VL},cage${i},${time},${gflops}"
  done
done

rm output.txt
```

Listing 27: Code of `spmv.sh`.

7.2.12 Stream

The **Stream** benchmark binary has no input values. Instead, there are different binaries, each one with the parameters used for execution already defined: `SIZE` and `VL`, explained in Table 18. Therefore, the Stream job will iterate over all binaries using the decided parameter values, as shown in line 11 from Listing 29.

Parameters	Meaning	Values
`SIZE`	Problem size	$2^{[11,20]}$
`VL`	Vector Length	$2^{[4,8]}$

Table 18: Stream parameters and their corresponding values.

The corresponding line and argument order to execute this benchmark is:

```
./stream-<SIZE>-elems-<VL>-vl
```

An example of an output can be seen in Listing 28. It shows the execution result with an executed binary with a pre-defined *SIZE* of 2048 elements and a *VL* of 16.

```
# ./stream-2048_elems-16_vl
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 2048 (elements), Offset = 0 (elements)
Memory per array = 0.0 MiB (= 0.0 GiB).
Total memory required = 0.0 MiB (= 0.0 GiB).
Each kernel will be executed 10 times.
```

```
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
 -------------------------------------------------------------
 Your clock granularity/precision appears to be 20 microseconds.
 Each test below will take on the order of 364 microseconds.
    (= 18 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            197.4     0.000197     0.000166     0.000303
Scale:           133.2     0.000263     0.000246     0.000393
Add:             226.5     0.000264     0.000217     0.000363
Triad:           219.4     0.000225     0.000224     0.000229
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all
 three arrays
-------------------------------------------------------------
```

Listing 28: Stream output example.

The Stream benchmark features four different functions. Each function is executed 10 times, and the best bandwith of an iteration obtained out of all executions is reported. The parsed parameters are the best bandwith for every function during the whole benchmark's execution, as shown in lines from 13 to 16 in Listing 29.

```
1  #!/bin/bash
2
3  BENCHMARKS_DIR=/shared
4  STREAM_BINARY=${BENCHMARKS_DIR}/stream/
5
6  echo "Size,VL,CopyBW,ScaleBW,AddBW,TriadBW"
7
8  for VL in 16 32 64 128 256; do
9    for SIZE in 2048 4096 8192 16384 32768 65536 131072 262144
     524288 1048576; do
10     for i in `seq 5`; do
11       ${STREAM_BINARY}/stream-${SIZE}_elems-${VL}_vl > output.txt
12       # Parsing the output:
13       copyBW=`grep Copy: output.txt | awk '{print $2}'`
14       scaleBW=`grep Scale: output.txt | awk '{print $2}'`
15       addBW=`grep Add: output.txt | awk '{print $2}'`
16       triadBW=`grep Triad: output.txt | awk '{print $2}'`
```

```
      echo "${SIZE},${VL},${copyBW},${scaleBW},${addBW},${triadBW}
   "
    done
  done
done

rm output.txt
```

Listing 29: Code of `stream.sh`.

7.3 Performance analysis

The work explained so far has been done to improve the capacity of the CI to detect core malfunctions and bugs. At this point, the implementation required developers to look at long CSV files produced by each benchmark if they wanted information about their performance. In order for developers to get a faster way of noticing whether a commit has improved, not modified, or worsened the core's performance, a system to generate automatic plots with the measurements produced during the `benchmarks` stage was implemented.

7.3.1 Developed solution

The starting point was the creation of a new stage, named `plotting`, that be would execute after the `benchmarks` stage to process all the CSV files containing each benchmark's execution results. The stage has been implemented following the structure of all the other stages. Initially, it only contained the implementation for one benchmark, Jacobi-2D, to then scale it to the other ones.

Consequently, the first step was creating a folder named `plotting` inside the CI directory, in the *epac_fpga* repository. Inside, there is a YAML file to describe the stage and program its jobs, named with the stage's name accordingly.

The `plotting` stage contains a single job, whose `script` section includes a line to execute a Gnuplot script that will produce the benchmark's plot. Gnuplot is a command-line plotting program which can be used as scripting language to automate generation of plots. The `plotting.yml` contents are displayed in Listing 30. The Gnuplot script was named `jacobi-2d.gnp`, whose code is shown in Listing 31.

```
 plotting:
   stage: plotting
```

```
3    tags:
4      - fpga
5      - vivado
6    needs: [benchmark.jacobi-2d]
7    script:
8      - gnuplot ci/plots/jacobi-2d.gnp
9
10 artifacts:
11   name: plots
12   paths:
13     - "./ci/plots/jacobi-2d.png"
```

Listing 30: Code of `plotting.yml`.

In order for the `plotting` job to access the CSV files produced by the benchmark jobs as artifacts, those jobs have to be specified as `needs`, as identified in line 6 from Listing 30.

Futhermore, a modification was made in the `benchmarks` stage, specifically the `jacobi-2d` job. The modifications were made to produce an additional artifact file with the average execution results of each input parameters combination, as each combination was executed five times. The average calculus was performed in the Jacobi job rather than in the Gnuplot script because it was more complex to do with Gnuplot than with bash.

The `plotting` job produces a PNG file that contains the plot produced by `jacobi-2d.gnp`, using the average values contained in `jacobi-2d-averages.csv`. That PNG file is the job's artifact, as seen in line 13 from Listing 30.

```
#!/bin/gnuplot

INPUT_FILE='jacobi-2d-averages.csv'

set terminal pngcairo enhanced dashed crop size 1024,768 font "Ubuntu,20"
set output './ci/plots/jacobi-2d.png'
set datafile separator ","

set xlabel "N"
set ylabel "Time (s)"
set grid
set key outside
set key above center title "Benchmark: Jacobi-2D" font ",18"
set key autotitle columnhead font ",16"

set style data histograms
```

```
17  set style fill solid border -1
18
19  cd "ci/benchmarks-outputs/jacobi-2d"
20  plot INPUT_FILE using 3:xticlabels(1) title "Current version", \

21  for [fn in system("ls -t *jacobi-2d.csv | head -n 5")] fn using
     3:xticlabels(1) title fn
```

Listing 31: Code of `jacobi-2d.gnp`.

The plot produced by the `jacobi-2d-gnp` script can be observed in Figure 15.

Furthermore, the `jacobi-2d.gnp` script includes support for an additional feature: plotting more than one data set, as seen in lines 20 and 21 from Listing 31. All the CSV files produced by every pipeline would be stored, and the Gnuplot script would plot the benchmark's data set of the last two pipelines in the same branch, altogether with the current pipeline's data set. However, the database where all CSV would be kept had yet to be implemented. In Figure 15, the two historical data sets were manually named and added for the Gnuplot script to access them.

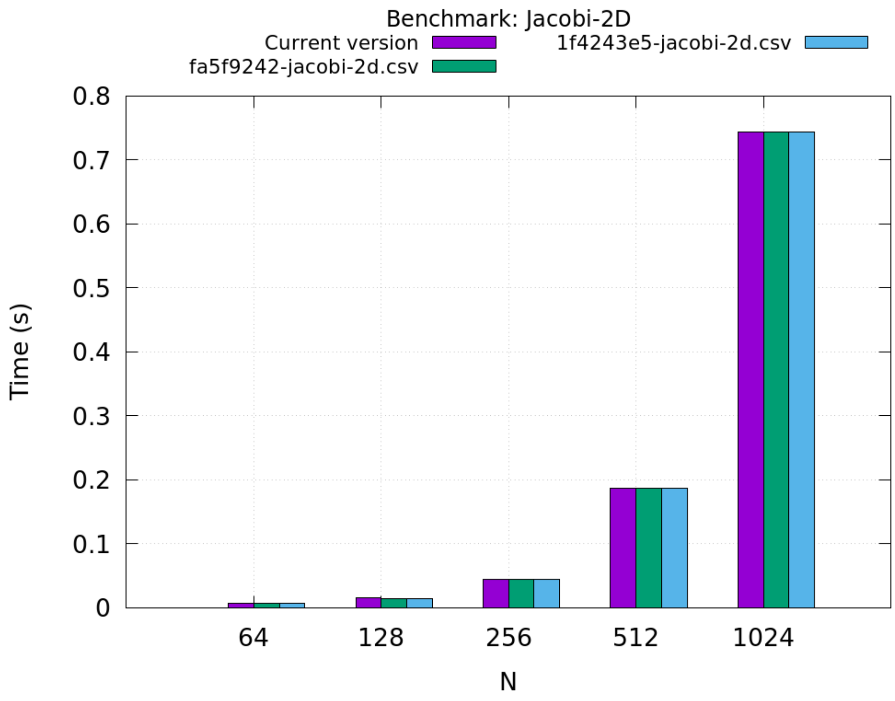


Figure 15: Plot produced during `plotting` stage. *Own compilation.*

The implementation of this task would have continued by adding a `[benchmark_-name].gnp` Gnuplot script for every benchmark implemented in `benchmarks` stage.

This would allow the `plotting` stage to produce a plot for every benchmark. In fact, as several benchmarks produce different output measurements, various plots could be produced for a few benchmarks.

However, development was stopped at this point, when the decision to develop the timeout and failure management system was taken.

7.4 Failure and timeout management

During the development of this final degree project, it was noticed that the benchmark jobs only failed when the pipeline's job timed out, but not if the execution of the benchmark itself failed. Furthermore, the CI jobs did not provide information when killed by timeout.

In a CI environment that seeks to offer its users additional testing features and ease development, it is fundamental to provide reliable and intuitive information.

For instance, if the execution of a benchmark is not correct because a core feature is malfunctioning and the CI is not providing that information to developers, the bug will go unnoticed. As a consequence, this implementation would not accomplish the fundamental objectives of this final degree project.

Therefore, proper failure and timeout scenarios management had to be implemented.

This interrupted the expected development of this final degree project; this situation led to a change in priorities, as explained in subsection 2.6. The development of the plotting stage became secondary to the benefit of a failure and timeout management system.

7.4.1 Initial status

The initial implementation had minimal timeout management and no failure management. This minimal setup was formed by two clauses in `benchmarks.yml`, shown in lines 8 and 14 respectively. The first one limits the total execution time of CI's jobs to two hours. The key option for the second one is the `-time=01:00:00`, which stops the Slurm job execution after one hour.

```
1 .benchmark:
  stage: benchmarks
  tags:
    - fpga
    - vivado
  variables:
    BENCHMARK_NAME: none
  timeout: 2h
  needs: [build_bitstream, build_linux.CI-testing-sdv3-minimal-
    network-new, utilities_linux.ssh, utilities_linux.nfs]
  dependencies:
    - build_bitstream
    - build_linux.CI-testing-sdv3-minimal-network-new
  script:
    - srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --time
    =01:00:00 --partition=fpga ./ci/benchmarks/run.sh ${
    BENCHMARK_NAME} | tee benchmark-${BENCHMARK_NAME}.out
```

Listing 32: Modified code of `benchmarks.yml`.

Three possible failure situations were identified, explained in Table 19. These errors are grouped into two categories: timeout and failure. The timeout group is formed by the **Resources unavailable** and **Long benchmark execution**. The other error, the **Wrong benchmark execution**, belongs to the failure group.

The current error management only identified one of the three possible failure situations. The only one detected was **Resources unavailable** because of the `timeout` YML clause. Therefore, this had to be fixed to detect them, and guarantee the job itself communicates to the EPI developers the cause of the wrong execution of the job itself.

Error	Description
Resources unavailable	No Pickles available during a long time
Long benchmark execution	Benchmark execution takes longer than it should
Wrong benchmark execution	Incorrect benchmark execution result

Table 19: Possible failures during the CI job execution.

7.4.2 Timeout management

The main objective was to differentiate when a job was timing out due to unavailable resources or unexpected longer benchmark execution.

Firstly, as previously explained, the initial implementation only detected one timeout situation: the unavailability of resources. And it was detected thanks to the `timeout` YAML parameter. In other words, the `time` option used in the `srun` command, from line 14 in Listing 32, did not have the expected behaviour, which was detecting the **long benchmark execution** error.

The `srun time` option is used to specify the total run time of the job allocation. Each task being executed is killed once the time limit is met. Hence, a `time` slightly higher than the longest execution expected would detect any abnormal execution that, for example, any hang in the core could cause.

As a first procedure, the unexpected `srun` behaviour was to be solved. A line as `exit $?` was added to the `script` job's environment, following the `srun` command line. It did not work as expected, because the `srun` command output was pipelined to a `tee` command. This meant that the exit code saved in `$?` was the `tee` one, instead of the desired one: the `srun` code.

Due to changes already done in the previous refactor, this `tee` was no longer necessary, so it was removed.

Then, an `if` clause statement was added to provide information to the EPI developers about the errors produced, as seen in Listing 33. The expected exit code from `srun` for the long benchmark execution error is 143.

```
script:
  - srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --time=01:00:00 --partition=fpga ./ci/benchmarks/run.sh ${BENCHMARK_NAME}
  - EXIT_CODE=$?
  - |
    echo "EXIT CODE:" $EXIT_CODE

    if [ $EXIT_CODE -eq 0 ] ; then
      echo -e "\e[32mSUCCESS: Benchmark execution finished correctly!" # Message in green
      exit 0
    elif [ $EXIT_CODE -eq 143 ] ; then
      echo -e "\e[31mERROR: Benchmark execution killed by timeout. Execution was taking longer than it should!" # Message in red
    else
      echo -e "\e[31mERROR: Something wrong!"
    fi

    exit 1
```

Listing 33: Added `if` clause to `benchmarks.yml`.

During the constant validation, an important detail was discovered: the CI job will immediately fail if any command within the `script` section returns a non-zero exit code. Whenever a `srun` command ends by the `time` option, its exit code is non-zero, and therefore the CI job's script ends abruptly, no further code lines are executed. The solution was saving the exit status in a variable as shown in Listing 34.

```
  - srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --time=01:00:00
    --partition=fpga ./ci/benchmarks/run.sh ${BENCHMARK_NAME} ||
    EXIT_CODE=$?
```

Listing 34: Saving the exit code from `srun` command.

This worked because if the `srun` fails, the variable assignment is always performed correctly, ending with a zero exit code. The assignment can capture any non-zero exit code from the `srun` command and continue execution with the if statement.

To identify a **resource unavailability** situation and notify developers, it could not be through the `timeout` YAML clause, so another approach was followed.

The `srun` command offers some tools to control timeouts. The first possibility was to use two of them: the `time` and `deadline` options.

The `deadline` option abruptly finishes both the allocation and execution once the specified date and time are met. Each task being executed is killed once the time limit is met. Similarly, if the resource allocation is yet to finish, the `time` option will stop it. Therefore, the programmer has to consider the maximum execution time assigned (`time` option) and the maximum waiting time for resource allocation, to specify the correct deadline hour.

The `srun`'s `now` parameter is used to get the exact date and time the `srun` command starts. Adding the desired time limit to `now` generated the expected deadline, as seen in Listing 35.

```
\centerline{srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --
    deadline=now+3minutes --time=01:00:00 --partition=fpga ./ci/
    benchmarks/${BENCHMARK_NAME}/run.sh || EXIT_CODE = $?}
```

Listing 35: The `srun` command with the `time` and `deadline` options.

In other words, the problem would be related to the binary execution if the `srun` command fails due to the `time` option. Otherwise, the problem would be related to the impossibility of getting resources for the `srun` job if the failure is due to the `deadline` option.

This first implementation proved partially unsuccessful, as the `srun deadline` option was not precise enough and had significant delays. In other words, ending earlier or later than specified. The problems were most likely related to the CI environment characteristics, and were far from this project's scope.

As an alternative to the `deadline` option, the `timeout` command from the GNU core utilities package was chosen.

The `srun` option `time` was still used for the previously explained case, while the `timeout` command would be used to specify a time limit to the whole `srun` execution: the resource allocation and the benchmark execution time. As a consequence, the `timeout` command value is specified taking into account the execution time limit set in the `srun time` option plus an arbitrary time limit for the `srun` job to be queued and allocated, as shown in Listing 36.

```
    - timeout 115m srun --job-name=CI-Benchmark-${BENCHMARK_NAME}
   --time=01:00:00 --partition=fpga ./ci/benchmarks/run.sh ${
   BENCHMARK_NAME} || EXIT_CODE=$?
```

Listing 36: The `timeout` command used altogether with the `srun` command.

This method proved successful, and the timeout times were adjusted: `srun`'s `time` option was set to fifteen minutes, while the `timeout` command was set to one hour and fifty-five minutes. The YAML's timeout was kept at two hours, leaving five minutes to execute all non-`srun` code.

Finally, another case was included in the `if` clause to evaluate the timeout command exit code, which is 124. The final `benchmarks` stage script is shown in Listing 37.

```
  script:
    - EXIT_CODE=0
    - timeout 115m srun --job-name=CI-Benchmark-${BENCHMARK_NAME}
   --time=00:15:00 --partition=fpga ./ci/benchmarks/run.sh ${
   BENCHMARK_NAME} || EXIT_CODE=$?
    - |
      echo "EXIT CODE:" $EXIT_CODE

      if [ $EXIT_CODE -eq 0 ] ; then
        echo -e "\e[32mSUCCESS: Benchmark execution finished
   correctly!" # message in color green
```

```
      exit 0
    elif [ $EXIT_CODE -eq 143 ] ; then
      echo -e "\e[31mERROR: Benchmark execution killed by timeout. Execution was taking longer than it should!" # message in color red
    elif [ $EXIT_CODE -eq 124 ] ; then
      echo -e "\e[31mERROR: Job allocation killed. Waited in queue for too long! Retry job when resources are available." # message in color red
    else
      echo -e "\e[31mERROR: Something wrong!"
    fi

    exit 1
```

Listing 37: Code of the `if` clause implemented in `benchmarks.yml`.

7.4.3 Failure management

To detect the failure situations, the first step was to **catch the binary's exit code in `[benchmark_name].sh`**.

The initial concept was to always get the exit code by creating the variable and assigning the binary's exit code: `EXIT_CODE=$?`; by placing the line right after each binary execution. Additionally, an `if` clause would check the captured exit code value. If the exit code had a non-zero value, the job would immediately end with an `exit 1` command, which is the exit code used for failed scripts.

However, the final approach ended up being the addition of `|| exit 1` code at the end of the binary execution line, because it provided a more straightforward way to end the script execution by avoiding an `if` statement. If the execution failed, it would execute `exit 1`; normal script execution would continue otherwise. The execution line from the Stream's `stream.sh` is as follows:

```
$STREAM_BINARY/stream-$SIZE_elems-$VL_vl > output.txt || exit 1
```

The next step was to **capture the exit code in the `run.sh` script**, as the failed `[benchmark_name].sh` script would not be enough for the CI job to end in failure as expected.

The `run.sh` is between every `[benchmark_name].sh` script and the main `benchmarks.yml` file, where the CI would be able to report the benchmark's failure to the final CI users. Thus, the exit code the `run.sh` script receives from the `[benchmark].sh` script has to be transmitted to the `benchmarks.yml` file.

Capturing the exit code from each `[benchmark_name].sh` script implied some code changes, as the `./tools/ssh-cmd` tool used for launching the benchmark execution script through ssh would hide the exit code. One possibility was to change that tool directly, but it was avoided as other CI stages outside this project's scope use it.

Instead, the tool's content was directly written into the `run.sh` script. The tool is, in fact, a long single-line `sshpass` command. The exit code can be caught by adding a code such as `RETVAL=$?` in the following line, as shown in Listing 38. Now, the `run.sh` is able to pass the exit code to the next step: the `benchmarks.yml` file.

```
sshpass -p riscv ssh -o ConnectTimeout=30 -o GlobalKnownHostsFile=/dev/null -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o PreferredAuthentications=password -c chacha20-poly1305@openssh.com root@10.0.0.2 "/root/stream.sh" > ./benchmarks-outputs/stream/stream.csv

RETVAL=$?
exit $RETVAL
```

Listing 38: Capturing the exit code in `run.sh`.

Finally, the last step was **managing the exit codes in the `benchmarks.yml` file**, adding another case in the `if` statement as can be seen in Listing 39.

```
  script:
    - EXIT_CODE=0
    - timeout 115m srun --job-name=CI-Benchmark-${BENCHMARK_NAME} --time=00:15:00 --partition=fpga ./ci/benchmarks/run.sh ${BENCHMARK_NAME} || EXIT_CODE=$?
    - |
      echo "EXIT CODE:" $EXIT_CODE

      if [ $EXIT_CODE -eq 0 ] ; then
        echo -e "\e[32mSUCCESS: Benchmark execution finished correctly!" # Message in green
        exit 0
      elif [ $EXIT_CODE -eq 1 ] ; then
        echo -e "\e[31mERROR: Benchmark execution finished incorrectly!" # Message in red
      elif [ $EXIT_CODE -eq 143 ] ; then
        echo -e "\e[31mERROR: Benchmark execution killed by timeout. Execution was taking longer than it should!" # Message in red
      elif [ $EXIT_CODE -eq 124 ] ; then
        echo -e "\e[31mERROR: Job allocation killed. Waited in queue for too long! Retry job when resources are available." # Message in red
      else
        echo -e "\e[31mERROR: Something wrong!"
      fi

      exit 1
```

Listing 39: Implemented `if` clause in `benchmarks.yml`.

8 Evaluation of objectives

This section briefly reviews the final state of the main objectives defined in sub-subsection 1.6.1.

- **Improvement of Benchmarks stage**
  Performance metrics are automatically generated. A CSV file is produced as an artifact by every benchmark, with every file offering the parsed output data of each benchmark execution.
- **Comprehensive automatic bug detection**
  Seven additional benchmarks have been implemented to the project's pipeline.

  If at least one benchmark fails, the associated job will fail and the commit will be automatically blocked as a consequence.

  In Figure 16 there is a pipeline whose jobs have not failed, in that case the commit that triggered that pipeline is not blocked, as seen in Figure 17.

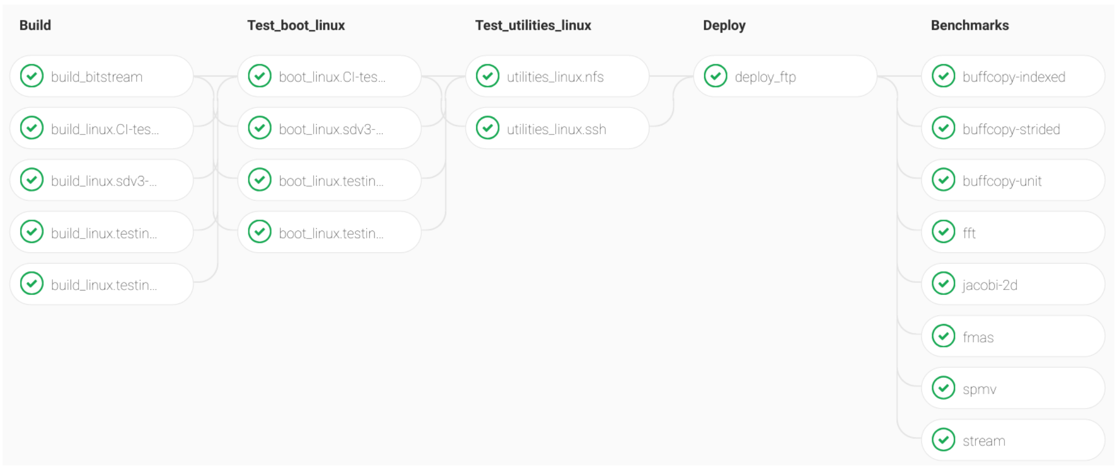


Figure 16: A successful pipeline. *Screenshot from GitLab.* [1]

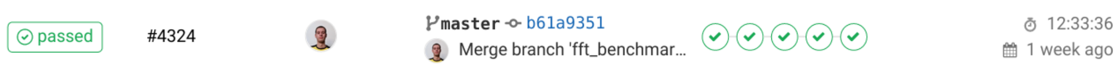


Figure 17: A commit that has ben accepted. *Screenshot from GitLab.* [1]

In contrast, a pipeline shown in Figure 18 has failed jobs. As a consequence, the commit that triggered it has been blocked by the CI, as displayed in Figure 19.

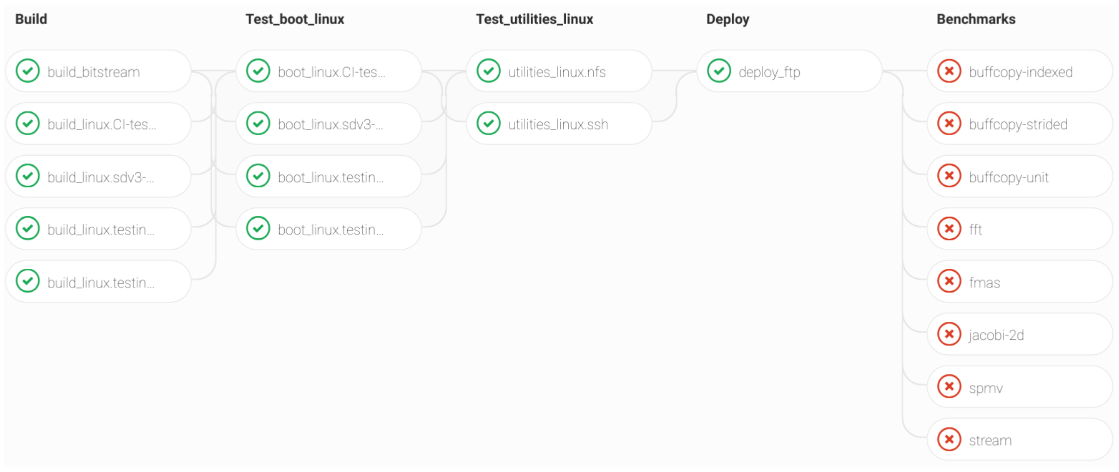


Figure 18: A failed pipeline. *Screenshot from GitLab.* [1]

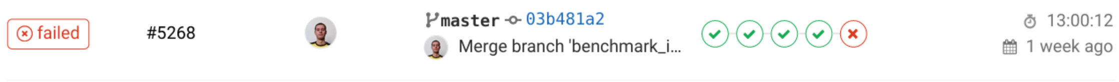


Figure 19: A commit that has ben blocked. *Screenshot from GitLab.* [1]

Jobs will fail if a benchmark execution returns a non-zero exit code. Furthermore, jobs also fail if they surpass the various time limits specified.

- **Automatic generation of performance plots**
  Plots are automatically generated for one job: Jacobi 2D. A plot will be produced as a PNG image by every Jacobi 2D job, and stored as an artifact, as shown in Figure 20.

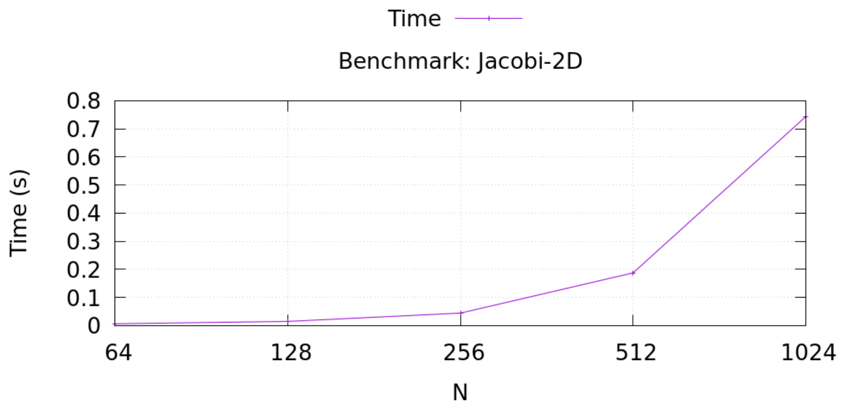


Figure 20: Plot generated by Jacobi 2D benchmark. *Own compilation.*

The objectives stated in subsubsection 1.6.1 have been fulfilled. The `Benchmarks` stage has been modified to produce and store performance metrics. A wide set of benchmarks have been included for an exhaustive core testing. In consequence, the repository has been made safer.

Moreover, automatic generation of performance plots has been included. However, in exchange of a safer setup, performance plots are not available for all benchmarks.

9 Conclusions

This final degree project aimed to contribute to the Continuous Integration of the European Processor Iniciative project repository, through the addition of a benchmark set to test individual features (C1) and the creation of an environment to identify errors in the design (C2).

The two main contributions have been divided in several specific objectives. All initial objectives have been fullfilled, except for the last tasks of the automatic generation of performance plots objective (task group T3): T3.4 and T3.5. Instead, the timeout and failure management has been achieved, a higher priority non-expected objective.

In the end, this change in priorities allowed this project to meet its primary aim: to guarantee that the resulting CI implementation is rigorous and eases the Register-Transfer Level development as much as possible. I have demonstrated the importance of my contributions with the examples presented in section 8 (Evaluation of objectives), where I have shown that commits that break the design are blocked by the Continous Integration pipeline.

Developing this project has allowed me to learn what Continuous Integration is and its advantages, how to create jobs and stages and how they work, and how to detect deficiencies with an existing implementation and correct them.

Working with a GitLab's CI implied learning about YAML coding and improving my bash scripting skills. Moreover, allowed me to learn about script modularization.

Furthermore, it let me learn what an Field Programmable Gate Array is and how to use it; improve my understanding of benchmarks and how to compile and execute them, and identify which and why are the relevant metrics reported by the program.

9.1 Future steps

After finishing this project, the most immediate steps are to resume the development of the performance analysis features that have been halted in order for the failure and timeout management to be developed.

Most notably, plots could be generated for all the implemented benchmarks. Benchmarks that output a data set with various magnitudes have a plot for each measurement unit.

Furthermore, an automatically genereted data base of CSV files obtained after every pipeline execution could also be implemented. It would allow for an easy access to historical data, as well as allowing for the automatized generation of plots with previous data sets for reference.

Additionally, extra benchmarks could be added to increase the CI's capacity to detect errors in specific core features.

References

[1] Project's GitLab (private). [Online]. https://epicore.bsc.es/gitlab/epac/epac_fpga. Accessed: February 2022. (document), 16, 17, 18, 19

[2] GitLab's official webpage - needs. [Online]. https://docs.gitlab.com/ee/ci/yaml/#needs. Accessed: May 2022. (document), 17

[3] High-Performance Computing (HPC). [Online]. https://insidehpc.com/hpc-basic-training/what-is-hpc/. Accessed: February 2022. 1.1

[4] Git. [Online]. https://git-scm.com/. Accessed: February 2022. 1.1

[5] Continuous Integration (CI). [Online]. https://docs.gitlab.com/ee/ci/. Accessed: February 2022. 1.1

[6] Continuous Integration jobs. [Online]. https://docs.gitlab.com/ee/ci/jobs/. Accessed: February 2022. 1.1

[7] Continuous Integration pipelines. [Online]. https://docs.gitlab.com/ee/ci/pipelines/. Accessed: February 2022. 1.1

[8] Job artifacts. [Online]. https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html. Accessed: February 2022. 1.1

[9] Vector Processor Unit (VPU). [Online]. https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data. Accessed: February 2022. 1.1

[10] Instruction Set Architecture (ISA). [Online]. https://www.arm.com/glossary/isa. Accessed: February 2022. 1.1

[11] RISC-V. [Online]. https://riscv.org/. Accessed: February 2022. 1.1, 5.1.1

[12] Field Programmable Gate Array. [Online]. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. Accessed: February 2022. 1.1

[13] FPGA bitstream. [Online]. https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html. Accessed: February 2022. 1.1

[14] Benchmark. [Online]. https://www.computerhope.com/jargon/b/benchmar.htm. Accessed: February 2022. 1.1

[15] Git submodules. [Online]. https://git-scm.com/book/en/v2/Git-Tools-Submodules. Accessed: May 2022. 1.1

[16] Barcelona Supercomputing Center. [Online]. https://www.bsc.es/. Accessed: February 2022. 1.3

[17] European Processor Iniciative. [Online]. https://www.european-processor-initiative.eu/. Accessed: February 2022. 1.3

[18] GitLab. [Online]. https://about.gitlab.com/. Accessed: February 2022. 1.7.2

[19] Glassdoor. [Online]. https://www.glassdoor.com. Accessed: March 2022. 5

[20] Amazon webpage for buying DELL Latitude 7490. [Online]. https://www.amazon.com/Latitude-Intel-i7-8650U-Windows-Laptop/dp/B079Q5Y4Q1. Accessed: March 2022. 3.2

[21] Xilinx webpage of Virtex UltraScale+ HBM VCU128 FPGA Evaluation Kit. [Online]. https://www.xilinx.com/products/boards-and-kits/vcu128.html. Accessed: March 2022. 3.2, 5.1.4

[22] Energuide. [Online]. https://www.energuide.be/en/. Accessed: March 2022. 3.2

[23] Standard internet plan. [Online]. https://www.masmovil.es/solo-internet. Accessed: March 2022. 3.2

[24] Idescat. [Online]. https://www.idescat.cat/pub/?id=aec&n=500. Accessed: March 2022. 4.3

[25] Slurm official webpage. [Online]. https://slurm.schedmd.com/overview.html. Accessed: April 2022. 5.1.3

[26] HiFive Unmatched processor from SiFive official webpage. [Online]. https://www.sifive.com/boards/hifive-unmatched. Accessed: February 2022. 5.1.3

[27] Xilinx evaluation kits. [Online]. https://www.xilinx.com/products/boards-and-kits.html. Accessed: March 2022. 5.1.4

[28] Official YAML webpage. [Online]. https://yaml.org/. Accessed: April 2022. 6.2

[29] GitLab's official webpage - jobs. [Online]. https://docs.gitlab.com/ee/ci/jobs/. Accessed: May 2022. 6.2.1

[30] GitLab's official webpage - predefined variables. [Online]. https://docs.gitlab.com/ee/ci/variables/predefined_variables.html. Accessed: May 2022. 6.2.1

[31] GitLab's official webpage - ci/cd pipelines. [Online]. https://docs.gitlab.com/ee/ci/pipelines/. Accessed: May 2022. 6.2.2

[32] GitLab's official webpage - artifacts. [Online]. https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html. Accessed: May 2022. 6.2.3

[33] GitLab's official webpage - dependencies. [Online]. https://docs.gitlab.com/ee/ci/yaml/#dependencies. Accessed: May 2022. 6.2.5

[34] GitLab's official webpage - artifacts acess. [Online]. https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html#access-the-latest-job-artifacts-by-url. Accessed: May 2022. 6.2.6

[35] GitLab's official webpage - cache. [Online]. https://docs.gitlab.com/ee/ci/caching/. Accessed: May 2022. 6.2.6

[36] GitLab's official webpage - variables. [Online]. https://docs.gitlab.com/ee/ci/variables/#custom-cicd-variables. Accessed: May 2022. 6.2.8

[37] HPCG official webpage. [Online]. https://www.hpcg-benchmark.org/. Accessed: February 2022. 7.2.1