TRANSPARENT APPLICATION DEPLOYMENT IN A SECURE, ACCELERATED AND COGNITIVE CLOUD CONTINUUM

Grant Agreement no. 101017168

Deliverable D4.1
HW/SW IPs for workload acceleration in disaggregated DCs

Programme:	H2020-ICT-2020-2
Project number:	101017168
Project acronym:	SERRANO
Start/End date:	01/01/2021 – 31/12/2023

Deliverable type:	Report
Related WP:	WP4
Responsible Editor:	AUTH
Due date:	31/03/2022
Actual submission date:	31/03/2022

Dissemination level:	Public
Revision:	1.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017168

Revision History

Date	Editor	Status	Version	Changes
01.11.21	Dimosthenis Masouros	Draft	0.1	Initial ToC
20.11.21	Dimitrios Danopoulos, Ioannis Oroutzoglou, Argyris Kokkinis, Aggelos Ferikoglou, Kostas Siozios	Draft	0.2	Sections 3.2
1.12.21	Dimosthenis Masouros, Ioannis Oroutzoglou, Argyris Kokkinis, Kostas Siozios	Draft	0.3	Sections 4.2
21.11.21	Ioannis Oroutzoglou	Draft	0.4	Sections 5.1, 5.5
23.12.21	Aggelos Ferikoglou, Argyris Kokkinis	Draft	0.5	Sections 5.2, 5.3, 5.4, 5.6
12.1.22	Aggelos Ferikoglou, Argyris Kokkinis, Ioannis Oroutzoglou	Draft	0.6	Section 6
14.2.22	Martin Sipos	Draft	0.7	Sections 4.1.1 and 4.1.2
15.3.22	Javier Martin	Draft	0.8	Sections 4.1.2 and 4.2.2
1.3.22	Ferad Zyulkyarov	Draft	0.9	Sections 4.1.3 and 4.2.3
2.3.22	Dmitry Khabi	Draft	0.9.1	Section 3.1
8.3.22	Dimitrios Danopoulos, Dimosthenis Masouros, Ioannis Oroutzoglou, Argyris Kokkinis, Aggelos Ferikoglou, Kostas Siozios	Draft	0.9.2	Integration of all contributions
29.3.22	Argyris Kokkinis	Draft	1.0	Final corrections

Author List

Organization	Author
CC	Martin Sipos
USTUTT/HLRS	Dmitry Khabi
AUTH	Dimitrios Danopoulos, Aggelos Ferikoglou Ioannis Oroutzoglou, Kostas Siozios, Argyris Kokkinis, Dimosthenis Masouros, Stylianos Siskos
INB	Ferad Zyulkyarov
IDEKO	Javier Martin

Internal Reviewers

Mellanox, ICCS

Abstract: This deliverable (D4.1) presents the outcomes of Task 4.1 "HW/SW IPs for workload acceleration in disaggregated DCs" describing the first version of SERRANO's accelerated kernels library. Hardware acceleration is part of the SERRANO's architecture, aiming to improve both the performance as well as the energy efficiency of the applications on cloud and edge devices. This deliverable presents the acceleration methodology and the evaluation results of six different algorithms from the SERRANO's use case applications. The presented algorithms were deployed on FPGAs and GPUs with diverse characteristics and measurements regarding their latency and power consumption are presented.

Keywords: SERRANO architecture, SERRANO platform, transparent deployment, hardware acceleration, secure storage, cognitive orchestration, service assurance

Table of Contents

1 Executive Summary 13

2 Introduction 14

3 HW/SW Acceleration Techniques & SERRANO Infrastructure characterization 15
3.1 Acceleration techniques on HPC clusters 15
3.2 Acceleration techniques on Cloud and Edge 17
3.2.1 FPGA-enabled HW acceleration 17
3.2.2 GPU-enabled HW acceleration 23
3.2.3 SERRANO's Cloud infrastructure 25
3.2.4 SERRANO's Edge infrastructure 26

4 Description, Analysis and Characterization of use-case applications 28
4.1 Use-cases Description 28
4.1.1 Secure Storage 28
4.1.2 Fintech Analysis 29
4.1.3 Anomaly Detection in Manufacturing Settings 31
4.2 Performance analysis & Characterization of use-case applications 33
4.2.1 Secure Storage 33
4.2.2 Fintech Analysis 41
4.2.3 Anomaly Detection in Manufacturing Settings 50

5 HW/SW IPs library for acceleration of computationally intensive kernels 54
Acceleration of AES GCM 54
5.1 Acceleration of RLNC Erasure Coding 54
5.2 Acceleration of Savitzky-Golay Filter 56
5.3 Acceleration of Kalman Filter 59
5.4 Acceleration of Wavelet Transformation 61
5.5 Acceleration of DBSCAN 61

6 Evaluation 62
6.1 Evaluation of AES GCM 62
6.2 Evaluation of RLNC Erasure Coding 63
6.3 Evaluation of Savitzky-Golay Filter 68
6.4 Evaluation of Kalman Filter 71
6.5 Evaluation of Wavelet Transformation 74
6.6 Evaluation of DBSCAN 76

7 Conclusion 79

List of Figures

Figure 1 Register Transfer Level Methodology 17
Figure 2 High-Level Synthesis Methodology 17
Figure 3 For loop execution with and without PIPELINE directive 18
Figure 4 Code transformation when partially loop unrolling of factor 2 is applied 19
Figure 5 Partitioning an array with factor 2 for different types 20
Figure 6 Function execution with and without DATAFLOW directive 21
Figure 7 Execution model 21
Figure 8 Using multiple compute units 22
Figure 9 Utilizing multiple memory banks 22
Figure 10:A 2D hierarchy of blocks and threads in CUDA programming model 24
Figure 11 Analyzing the use-case before acceleration 25
Figure 12: Illustration of the Cloud infrastructure 25
Figure 13: Illustration of the Edge infrastructure 26
Figure 14: Secure Storage: file upload and download workflows 28
Figure 15: Investment management workflow summarized in five continuously repeating steps. 30
Figure 16:High level use-case workflow 32
Figure 17: High level architecture 32
Figure 18: Roofline model of AES-GCM 33
Figure 19: Execution time breakdown for x86 architecture 34
Figure 20: Execution time breakdown for ARM architecture 34
Figure 21 Encoding Mechanism 35
Figure 22 Roofline model of the encoding and decoding tasks 36
Figure 23 RLNC encoding (x86) execution time 37
Figure 24 RLNC encoding (ARM) execution time 37
Figure 25 RLNC decoding (x86) execution time 38
Figure 26 RLNC decoding (ARM) execution time 38

Figure 27 Encoder overall execution time (x86) 39
Figure 28 Decoder overall execution time (x86) 39
Figure 29 Encoder overall execution time (x86) 40
Figure 30 Decoder overall execution time (ARM) 40
Figure 31 Smoothing a time series 41
Figure 32 Moving window convolutions 42
Figure 33 Savitzky-Golay filter roofline model 42
Figure 34 Filter's execution time (x86) 43
Figure 35 Filter's execution time (ARM) 43
Figure 36 Execution time for multiple assets (x86) 44
Figure 37 Execution time for multiple assets (ARM) 44
Figure 38 Kalman filter's output using InbestMe's closing prices dataset 46
Figure 39 Roofline model for Kalman filter 46
Figure 40 Kalman filter execution time breakdown for x86 architecture 47
Figure 41 Kalman filter execution time breakdown for ARM architecture 47
Figure 42 Execution time for multiple assets for x86 architecture 48
Figure 43 Execution time for multiple assets for ARM architecture 48
Figure 44 Roofline model for wavelet transform 49
Figure 45 Wavelet filter execution time breakdown on x86 CPU 49
Figure 46 Wavelet filer execution time breakdown on ARM CPU 50
Figure 47 DBSCAN's output in IDEKO's UC 51
Figure 48 Roofline model for DTW 52
Figure 49 DTW x86 execution time 53
Figure 50 DTW ARM execution time 53
Figure 51 Unrolling the finite field computations by instantiating multiple parallel components 55
Figure 52 Using multiple compute units for parallel encoding - decoding 56
Figure 53 Designed acceleration mechanism 58

Figure 54 Execution of multiple signals using many compute units 58
Figure 55 Batched Kalman filter's score for different batch sizes 59
Figure 56 Batched Kalman filter's MEP for different batch sizes 60
Figure 57 AES-GCM acceleration on T4 GPU 62
Figure 58 AES-GCM acceleration on Xavier AGX GPU 63
Figure 59 RLNC encoding U50 execution time 64
Figure 60 RLNC decoding U50 execution time 65
Figure 61 RLNC encoding ZCU102 execution time 65
Figure 62 RLNC decoding ZCU102 execution time 66
Figure 63 RLNC encoder (U50) acceleration 66
Figure 64 RLNC decoder (U50) acceleration 67
Figure 65 RLNC encoding (ZCU102) acceleration 67
Figure 66 RLNC decoding (ZCU102) acceleration 68
Figure 67 Filter's execution on U50 69
Figure 68 Filter's acceleration on U50 70
Figure 69 Filter's execution on ZCU102 70
Figure 70 Filter's acceleration on ZCU102 71
Figure 71 Batched Kalman filter's execution time breakdown on ALVEO U50 71
Figure 72 Batched Kalman filter's execution time breakdown on ZCU102 72
Figure 73 Batched Kalman filter's execution time speedups on ALVEO U50 73
Figure 74 Batched Kalman filter's execution time speedups on MPSoC ZCU102 73
Figure 75 Wavelet acceleration on T4 GPU 75
Figure 76 Wavelet acceleration on Xavier AGX GPU 75
Figure 77 DTW execution time breakdown on ALVEO U50 76
Figure 78 DTW execution time breakdown on ZCU102 77
Figure 79 DTW's execution time speedups on ALVEO U50 77
Figure 80 DTW's execution time speedups on MPSoC ZCU102 78

List of Tables

Table 1: Hardware PCIe Devices lineup - Detailed specs 26

Table 2: Hardware edge devices lineup - Detailed specs 27

Table 3 Filter's coefficients 56

Table 4 Batched Kalman filter’s configuration per device 60

Table 5 RLNC erasure coding Encoder 63

Table 6 RLNC Erasure coding Decoder 64

Table 7 Savitzky-Golay filter’s consumed resources for all the available designs 68

Table 8 Batched Kalman filter’s consumed resources for all the available designs 74

Table 9 DTW’s consumed resources for all the available devices 78

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
ASIC	Application-Specific Integrated Circuit
BSI	Belief Desire Intention
CFD	Computational Fluid Dynamics
CI/CD	Continuous integration/Continuous Deployment
CNC	Computer Numerical Control
CORS	Cross-Origin Resource Sharing
DOCA	Data center On a Chip Architecture
DPU	Data Processing Unit
DSE	Design Space Exploration
EDE	Event Detection Engine
EU	European Union
FaaS	Function as a Service
FPGA	Field Programmable Gate Array
GCP	Google Cloud Platform
GPU	Graphics Processing Unit
HLRS	High Performance Computing Center Stuttgart
HLS	High-Level Synthesis
HPC	High Performance Computing
HW	Hardware
IO	Input/Output
MAAS	Metal as a Service
ML	Machine Learning
MOM	Message-Oriented Middleware

MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OCI	Open Container Initiative
OSS	Object Storage Server
OST	Object Storage Target
PCIe	Peripheral Component Interconnect Express
PXE	Preboot Execution Environment
QoS	Quality-of-Service
RDMA	Remote Direct Memory Access
REST	Representational State Transfer
RLNC	Random Linear Network Coding
RoCE	RDMA over Converged Ethernet
ROT	Resource Optimization Toolkit
RPC	Remote Procedure Call
RTL	Register-Transfer Level
SAR	Service Assurance and Remediation
SDK	Software Development Kit
SFTP	Secure File Transfer Protocol
SLA	Service Level Agreement
SoC	System on a Chip
SW	Software
TLS	Transport Layer Security
TPM	Trusted Platform Module
TTM	Time to Market
UC	Use Case
URL	Uniform Resource Locator
VM	Virtual Machine

VMM	virtual Machine Monitor
VVUQ	Verification, Validation and Uncertainty Quantification
WP	Work Package
WSGI	Web Server Gateway Interface

1 Executive Summary

H2020 SERRANO aims to take important steps in providing a transparent way of deploying applications in the Edge-Cloud-HPC computing continuum. Specifically, SERRANO provides an abstraction layer that automates the process of application deployment across the various computing platforms, thus realizing an intent-based paradigm of operating federated infrastructures.

This document presents the SERRANO deliverable D4.1 entitled “HW/SW IPs for workload acceleration in disaggregated DCs” focusing on the first version of SERRANO’s accelerated kernels library. Hardware acceleration is a key component of SERRANO’s architecture, aiming to improve both the performance as well as the energy efficiency of deployed applications and underlying infrastructure.

In this deliverable, first, we give a theoretical background on acceleration techniques for HPC, Edge and Cloud infrastructures (Section 3). Specifically, we introduce the basic programming models used to accelerate applications on HPC clusters (MPI), FPGAs (OpenCL) and NVIDIA GPUs (CUDA), we describe a set of major optimization techniques per se and we present SERRANO’s hardware infrastructure for running accelerated applications. Next, we focus particularly on SERRANO’s use-case applications. We give a brief description for each use-case and we perform an extensive characterization in order to identify their computationally intensive parts (Section 4). Then, we present our first version of accelerated compute kernels, which consists of 6 different algorithms tailored to the specific needs of the use-cases workflows. We present in detail the optimization techniques used to accelerate these kernels (Section 5) and we evaluate our developed accelerated kernels (Section 6), showing that we are able to achieve up to x229 speedup compared to conventional CPU execution.

2 Introduction

Nowadays, the explosive growth and increasing power of IoT devices along with the rise of 5G networks have resulted in unprecedented volumes of data. Emerging use cases around autonomous vehicles, smart-cities, and smart factories require data processing and decision making closer to the point of data generation due to mission-critical, low-latency and near-real time requirements of such deployments. To this end, multi-layered computing architectures are emerging, where computing resources and applications are distributed from the edge of the network, closer to where data are gathered, to the cloud, realizing the edge-cloud computing continuum.

Even though edge-cloud architectures expand the computing capacity of the traditional cloud paradigm by introducing an additional huge pool of computing resources, the inefficiency of traditional CPUs to provide fast, near real-time executions has also led to the introduction of hardware accelerators, such as GPUs and FPGAs, to the aforementioned hierarchy.

Typical examples of accelerators include power-efficient devices at the edge (e.g., NVIDIA Jetson and Xilinx MPSoC), to high-performance, massively-parallel devices at the cloud (e.g., NVIDIA Ampere and Xilinx Alveo). While hardware accelerators provide increased performance gains, these benefits do not come for free, as such devices typically require more power to operate.

From the edge point of view, energy efficiency has always been a first class system-design concern, to provide low-power embedded systems design. On the cloud side, the considerable share of electricity expenses over the total cost of ownership (TCO), as well as the shift towards energy-efficient (green) computing at the data-center level, also indicate the need for efficient hardware acceleration.

3 HW/SW Acceleration Techniques & SERRANO Infrastructure characterization

3.1 Acceleration techniques on HPC clusters

The tremendous growth in computers integrates compute capacity from a single process with a single core into multi-core (multi-process), therefore bringing the opportunity to accelerate computing.

Parallel programming provides us the capability to use all compute resources by partitioning the data among processes, and all processes could process part of data concurrently, and as result accelerate the computing process. There are three main parallel programming paradigms: parallel shared-memory, parallel distributed and Partitioned Global Address Space (PGAS) programming models.

Parallel shared memory programming: Consider a single compute node that has multiple cores, and cores have shared memory access. OpenMP provides the parallel shared memory programming interface, which is basically an implementation of multithreading. In this framework, different numbers of threads would spawn and call the parallel region "#pragma omp parallel". The functionality that is provided to the multithreaded programs by OpenMP by the omp pragmas, such as "#pragma omp barrier", "#pragma omp critical" can additionally synchronize the parallel threads in order to control the safe parallel access to the part of the memory which is shared between threads. The OpenMP specification provides the detailed description about the functionality of the interface and offers many useful example[1].

Parallel distributed memory programming: In this case, we have multi processes that have distributed (disjoint) memory while data are partitioned among the processes. In this case, special protocol is required to exchange the data. MPI (Message Passing Interface) is a standard communication library to exchange information between processes. MPI has methods such as "MPI_Send", "MPI_Recv" that enable users to send and receive data between processes, and also collective operators such as "MPI_Scatter", and "MPI_Gather", which send data from one process to all processes and vice versa. The MPI Standard documentations offers detailed description of the interface. The current version of the MPI is 4.0[2].

With the increasing complexity of compute nodes, such as the increase in cores and numa nodes, the OpenMP/MPI hybrid model is playing an increasingly important role.

[1] OpenMP, The OpenMP API specification for parallel programming, https://www.openmp.org/specifications

[2] MPI Forum and MPI Documents, https://www.mpi-forum.org/docs

Although programming with this model is often a real challenge, it can offer significant advantages. Among them, three stand out:

- It allows dynamic adjustment of computational capabilities: one can easily change the number of active threads during computation, which is impossible by MPI programming model.
- Domain decomposition provides better load balancing in many cases by reducing the number of decomposition parts.
- The overhead of message exchange within compute nodes is reduced.

That hybrid parallel model was used in the development of the first HPC services. Implementation details can be found in Deliverable D4.2.

GASPI (Global Address Space Programming Interface) is an implementation of the partitioned global address space programming model (PGAS)[3] and is also a communication library for distributed memory system, such as the Hawk. The communication operations of PGAS span a global memory address space that is logically partitioned parts. Each of these parts is assigned to a certain process and its threads. Unfortunately, the PGAS model is supported only by a limited number of compilers. Even if e.g. Fortran 2008 standard offers "Coarrays", a PGAS API, it is often only partially implemented, as in the GNU or not efficiently, as for example in Intel. The problem with Intel's compiler is that the interface is implemented as a wrapper over MPI routines, which makes the "Coarrays" programs very slow.

The GASPI, PGAS-API, is based on single-sided asynchronous communication primitives which are enhanced by lightweight synchronization, therefore every thread could essentially communicate. This capability provides overlapping of communication by computation and reduce the synchronization in communication. These are the advantages of GASPI over MPI which is an appropriate communication library for heterogeneous HPC systems.

GASPI is one of the most efficient implementations of the PGAS languages. It was developed by the Fraunhofer Institute for Industrial Mathematics (ITWM). However, you must have a commercial license on the supercomputer to use it. Even if Hawk has this, it is often not the case for other systems. This is one of the main reasons why we do not support this model in the Serrano project. More details about the GASPI can be found on its official forum on the internet[4].

[3] Wiki: Partitioned global address space, https://en.wikipedia.org/wiki/Partitioned_global_address_space
[4] Forum of the PGAS-API GASPI, http://www.gaspi.de

3.2 Acceleration techniques on Cloud and Edge

3.2.1 FPGA-enabled HW acceleration

FPGA-enabled HW acceleration

FPGA Programming Models

Designing hardware for FPGAs can be performed at varying levels of abstraction with the commonly used being the register-transfer level (RTL) and the algorithmic level. These methodologies differ; RTL (i.e., VHDL, Verilog) is used for circuit-level programming while algorithmic level methodologies such as High-Level Synthesis (HLS) are used for describing designs in a more abstract way. Figure 1 and Figure 2 present the designing steps of these methodologies. Compared to RTL, HLS provides a faster and more flexible development process, as designers instruct the HLS compiler on how to synthesise accelerators by adding different directives on a C/C++ or OpenCL code.

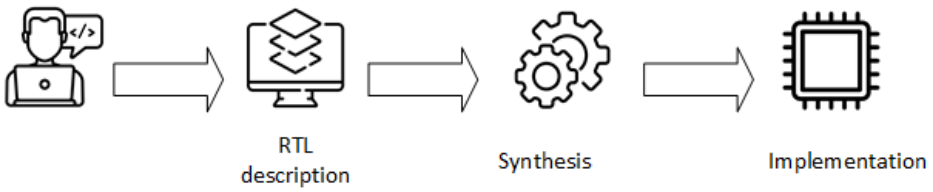

Figure 1 Register Transfer Level Methodology

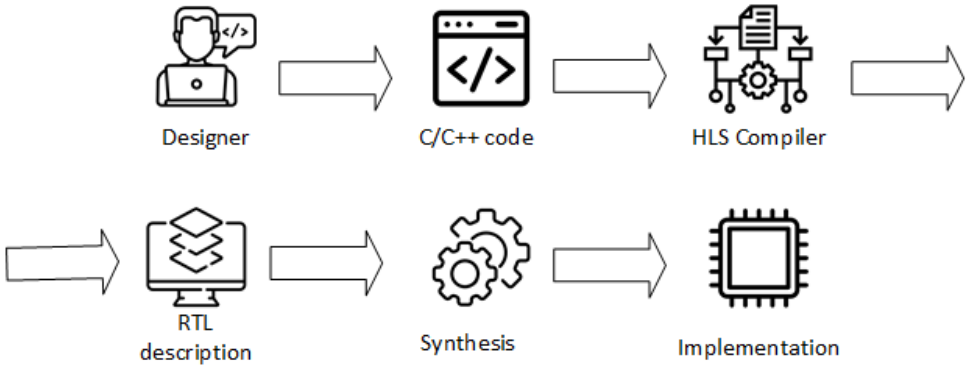

Figure 2 High-Level Synthesis Methodology

Xilinx Inc., one of the main FPGA vendors on the market, has put significant effort into providing a user-friendly toolset for employing HLS for FPGA design. In this context, Xilinx introduced Vitis, a framework that provides a unified OpenCL interface for programming edge (e.g., MPSoC ZCU104) and cloud (e.g., Alveo U200) Xilinx FPGAs. Vitis aims to simplify the designing process and as a result let developers focus on the Design Space Exploration (DSE) phase, which targets the performance optimizations with respect to the architecture and resources of the available devices. As the SERRANO platform is provisioned with both cloud and edge FPGAs, Vitis was considered a reasonable choice for the development process.

High-Level Synthesis directives

In HLS context directives are added in the C/C++ kernel code, to instruct the compiler on how to synthesise the bitstream (the representation that is used for programming the device). There is a wide range of different directives (also referred to as pragmas) that a developer can use. Pragmas are mainly divided in two categories: a) those that affect kernel performance and b) those that aim to provide helpful information to the compiler for the synthesis process (e.g., define the minimum and maximum iterations of a for loop, define false dependencies). In this section, we mainly focus on the directives that affect performance.

- **#pragma HLS PIPELINE**

Pipeline directive is used for reducing the initiation interval (II) of functions or loops by allowing the concurrent execution of operations. A pipelined block of code can process new inputs every N clock cycles, where N is the II (e.g., an II of 1, processes a new input every clock cycle). As a default behaviour, with the PIPELINE pragma Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. The emphasis will be on meeting timing, rather than on achieving II. If the Vitis HLS tool cannot create a design with the specified II, a warning is issued while the design with the lowest possible II is created. This warning can be used to analyse this design and determine what steps must be taken to create a new one that satisfies the required II (remove loop carry dependencies, restructure code etc.).

Figure 3 depicts the way a for loop is going to be implemented on hardware with and without using the pipeline directive. (A) shows the default sequential operation where there are three clock cycles between each input read (II=3), and it requires eight clock cycles before the last output write is performed. On the other hand, (B) shows the pipelined operations that require only one cycle between reads (II=1), and 4 cycles to the last write.

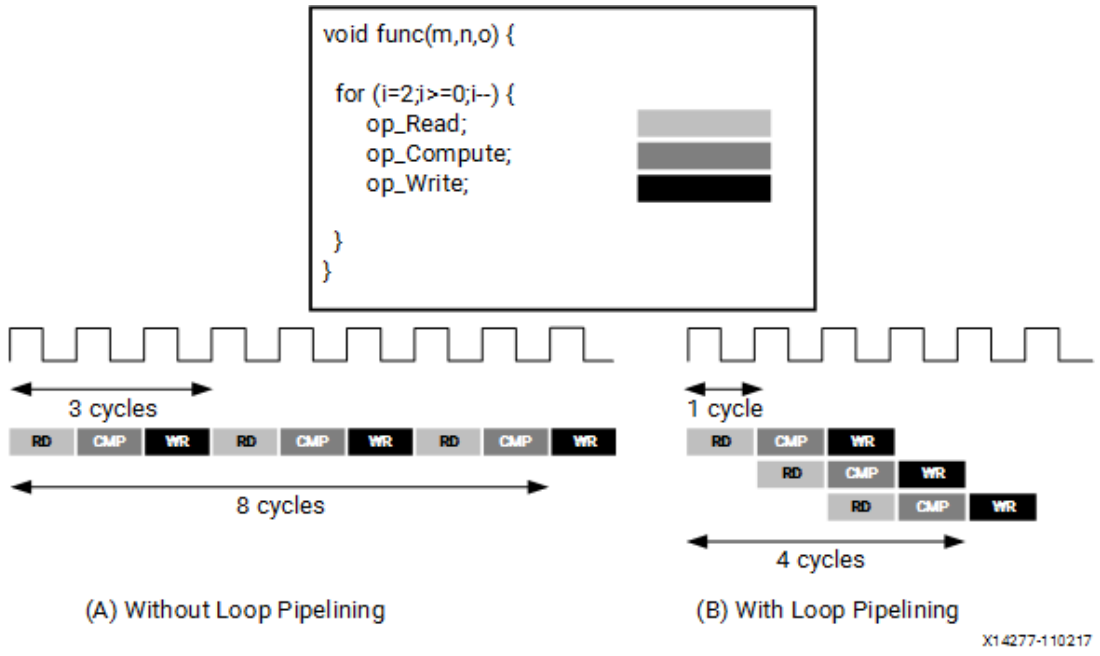

Figure 3 For loop execution with and without PIPELINE directive

- **#pragma HLS UNROLL**

By default, loops in C/C++ functions are kept rolled. In this case, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. The UNROLL pragma transforms loops by creating multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel leading to an increase in throughput and required resources.

UNROLL directive allows the loop to be fully or partially unrolled. Fully unrolling a loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can run concurrently. Partially unrolling a loop lets user specify a factor N, to create N copies of the loop body and reduce the loop iterations accordingly.
Figure 4 shows the way a loop is transformed when it is partially unrolled by a factor of 2.

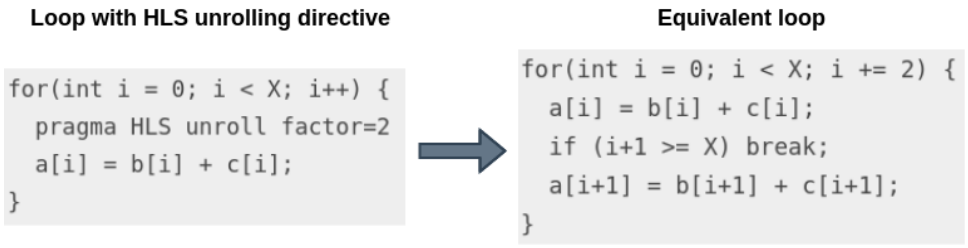

Figure 4 Code transformation when partially loop unrolling of factor 2 is applied

- **#pragma HLS ARRAY_PARTITION**

The ARRAY_PARTITION pragma is used for partitioning an array into smaller arrays or to its individual elements. Memory partitioning results in RTL with multiple small memories or multiple registers. In this way, the amount of read and write ports for storage is effectively increased leading to a potential improvement of the final design throughput. On the other hand, the design requires more memory instances or registers which might be a problem in case of FPGAs with limited memory resources.

HLS provides three types of array partitioning:

a. **Block** where the original array is split into equally sized blocks of consecutive elements of the original array
b. **Cyclic** where the original array is split into equally sized blocks interleaving the elements of the original array
c. **Complete** that splits the original array to its individual elements

Figure 5 shows how a one-dimensional array with six elements is going to be formed, after using array partitioning with factor 2 for all the available partitioning types.

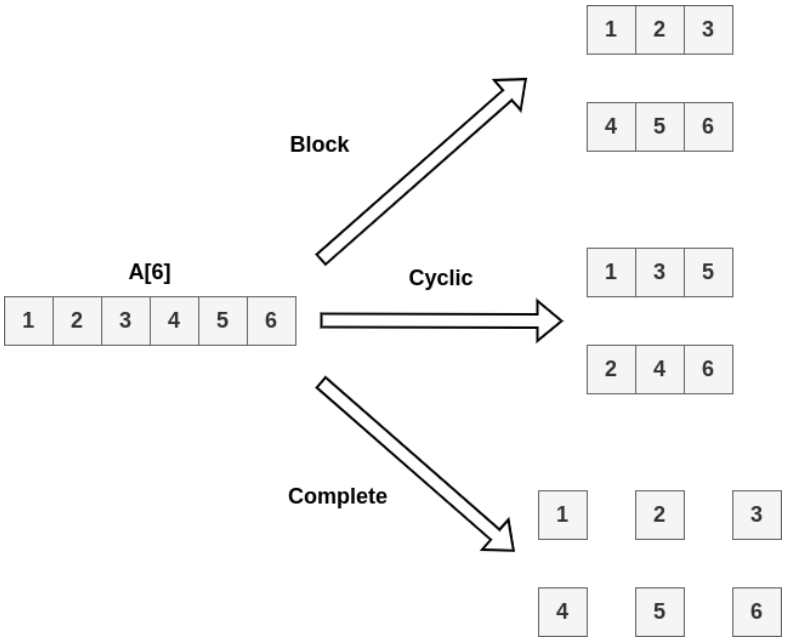

Figure 5 Partitioning an array with factor 2 for different types

- **#pragma HLS DATAFLOW**

The DATAFLOW directive enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and the overall throughput of the design.

In a C/C++ description, operations are performed sequentially. In the absence of directives that limit resources, the Vitis HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can be a limiting factor. For instance, functions or loops that access arrays must finish all read/write operations before they complete, preventing the next function or loop, that consumes the data, from starting. The DATAFLOW optimization enables the operations in a function or loop to start before the previous function or loop completes all its operations.

When the DATAFLOW pragma is specified, the HLS tool analyses the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL. If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.

Figure 6 depicts the way top function is going to be implemented on hardware with and without using the DATAFLOW directive. (A) shows the default sequential operation where top requires eight clock cycles to execute. On the other hand, (B) shows the operations executed using dataflow where top requires only five clock cycles due to function overlapping.

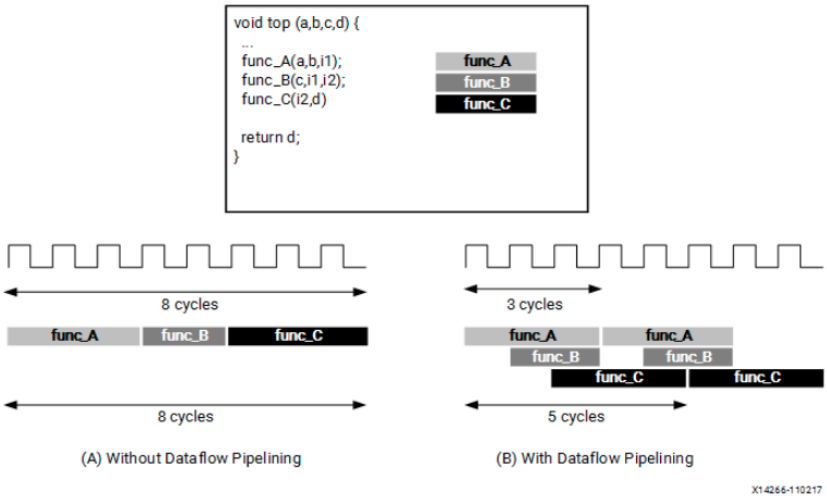

Figure 6 Function execution with and without DATAFLOW directive

FPGA memory optimizations

The execution of an accelerated application on FPGAs involves two parts.

a. Executing the host executable application on the host processor.
b. Launching the computationally intensive kernel on the hardware platform.

In a typical execution flow, the host application communicates with the hardware platform through PCIe or AXI bus for the cloud and edge devices respectively. The host program invokes the corresponding OpenCL APIs to transfer data from the processor's side to the hardware platform's global memory. Then the hardware kernel reads the data from the global memory, performs computations and writes the results back, allowing the host application to retrieve the results. This execution model is illustrated on the Figure 7 below.

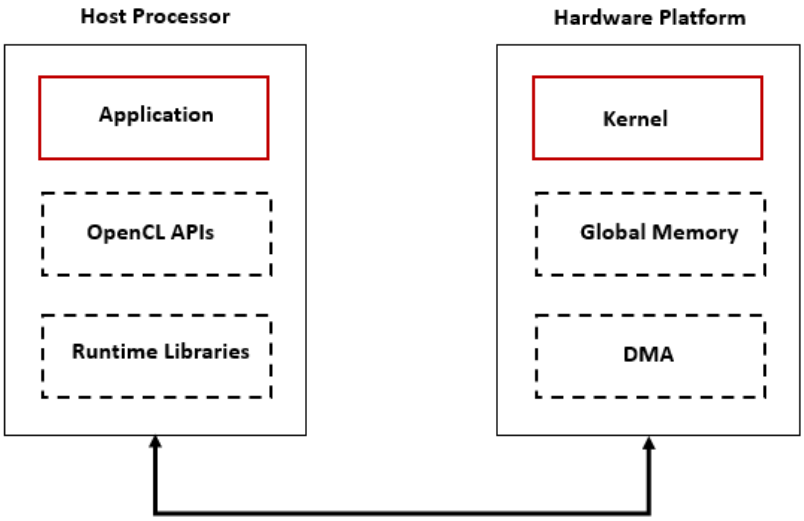

Figure 7 Execution model

To maximize the benefits of hardware acceleration, optimizations that target to minimize the latency of transferring data from and to the global memory and utilize optimally the providing hardware resources can be performed. Some memory optimization techniques are enumerated and described briefly below.

1. **Using multiple compute units**: Instantiating multiple compute units on the same hardware platform allows the concurrent execution of multiple accelerators, leading to an execution model with coarse-grained parallelism. This optimization technique requires from the host application to schedule the execution of each workload through out-of-order queues. This type of queue allows the scheduler to dispatch each workload to the compute unit that is currently available. Figure 8 below depicts an example of scheduling workloads to multiple compute units.

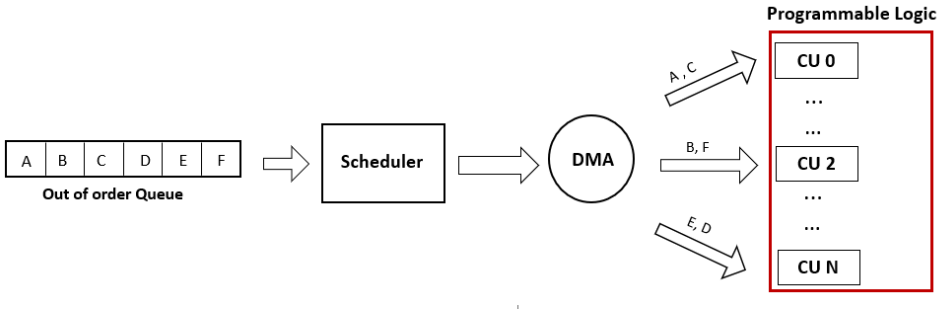

Figure 8 Using multiple compute units

2. **Using multiple memory banks:** To allow different compute units that are running on the same platform to read or write data from or to the global memory simultaneously, each of their memory interfaces should be mapped to a different memory bank. The providing memory banks might be DDR (Double Data Rate), HBM (High Bandwidth Memory) memories or both, depending on the type of the device. This technique requires from the designer to specify the memory type and the port that will be assigned to each of compute unit's interfaces. Figure 9 below shows one compute unit with two memory interfaces mapped to different DDR memory banks.

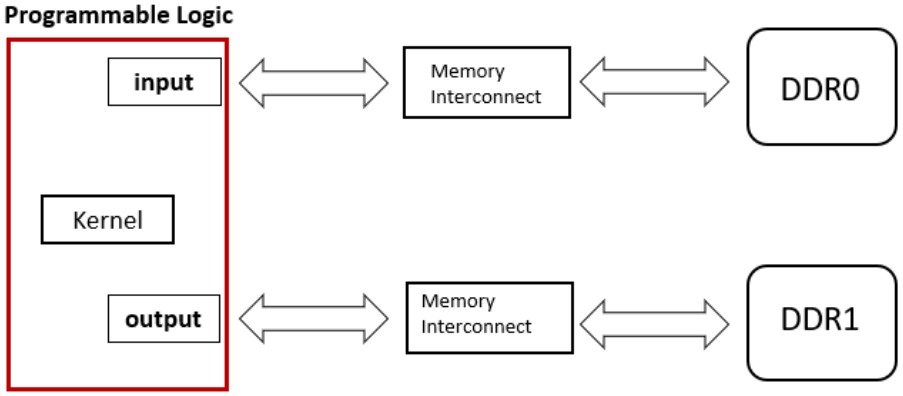

Figure 9 Utilizing multiple memory banks

3. **Utilizing the HBM resources:** Some hardware platforms that are deployed on data centres (such as the Alveo U50 card) provide HBM memory blocks instead of the typical DDR memories. Those devices occupy two HBM blocks with each one composed of 16 layers, forming a pool of high-speed memory resources that can be exploited by the designer. This type of memory offers a bandwidth of up to 460 GB/sec contrary to the 77 GB/sec of the typical DDR memories, allowing designers to boost the acceleration on memory-bound applications.
4. **Using the full AXI width:** This optimization technique targets to minimize the number of memory transactions between the global memory and the hardware compute units. The bandwidth of the AXI bus that is used for transferring data between the hardware platform and the global memory is 512 bits. However, as most data types are represented by fewer bits, the full available bandwidth remains unutilized. In order to use the provided AXI width at its full extent, data coalescing can be performed and pack multiple data elements, forming larger ones that are represented by 512 bits. This technique minimizes the latency overhead from moving data across the hardware platform and the global memory.

3.2.2 GPU-enabled HW acceleration

General-purpose computing on a GPU is the use of a GPU together with a CPU in order to accelerate computation in applications traditionally handled only by the CPU. Over the last few years, GPUs have already become a fundamental component in most of the world's fastest supercomputers, considered among the most mature and widely-used type of hardware accelerators. Additionally, while their programmability is significantly improved over the years with high level languages (such as CUDA and OpenCL) GPU programming is becoming more and more a mainstream in the scientific community.

CUDA, the most popular GPU programming model, stands for Compute Unified Device Architecture and is a parallel programming paradigm supporting only NVIDIA GPUs, which was released in 2007 by NVIDIA. In the other hand, OpenCL (Open Computing Language) was launched by Apple and the Khronos group as a way to provide a benchmark for heterogeneous computing that was not restricted to only NVIDIA GPU's. It offers a portable language for GPU programming that can run on CPUs, GPUs, Digital Signal Processors and other types of processors.

For the purposes of SERRANO project, we make use of NVIDIA GPUs and therefore the CUDA programming model is used. More specifically, there are three key language extensions CUDA programmers can use: CUDA blocks, shared memory, and synchronization barriers. CUDA blocks contain a collection of threads that can share a specific type of memory called shared memory, while they can pause until all threads reach a specified set of execution.

Additionally, the CUDA programming model enables developers to scale software, increasing the number of GPU processor cores as needed, while we can use CUDA language abstractions to program applications and divide programs into small independent problems. In a more

technical manner, GPU programs are written in .cu files. CUDA C adds the __global__ qualifier to standard C in order to alert the NVIDIA (nvcc) compiler that a function, called kernel, should be compiled to run on a device instead of the host. Developer can pass parameters to a kernel as they would with any C function after having determined the kernel grid (<<<grdiDim, blockDim>>>). Note that gridDim refers to the number of blocks, where blockDim to the number of threads per block. Figure 10 depicts a 2D hierarchy of a kernel organization.

Additionally, and before kernel execution, developers need to allocate memory in the GPU's global memory using cudaMalloc(). This call behaves very similarly to the standard C call malloc(), but it tells the CUDA runtime to allocate the memory on the device. Finally, the CUDA developer accesses memory on a device through calls to cudaMemcpy() from host code. These calls behave exactly like standard C memcpy() with an additional parameter to specify which of the source and destination pointers point to device memory.

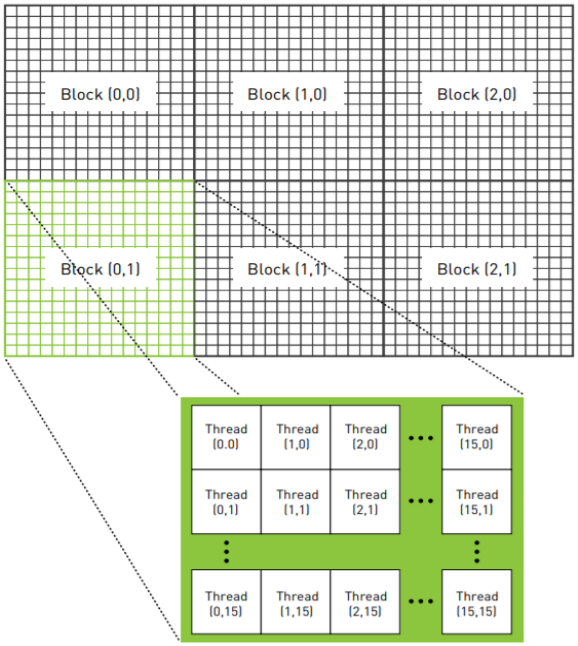

Figure 10:A 2D hierarchy of blocks and threads in CUDA programming model

For the scope of SERRANO, use-cases' related applications provided by InBestMe, IDEKO and CC that need to be accelerated are written in either Python or C++. To convert them to CUDA applications, we first need to profile and analyse them in order to determine the computationally intensive regions as depicted in Figure 11.

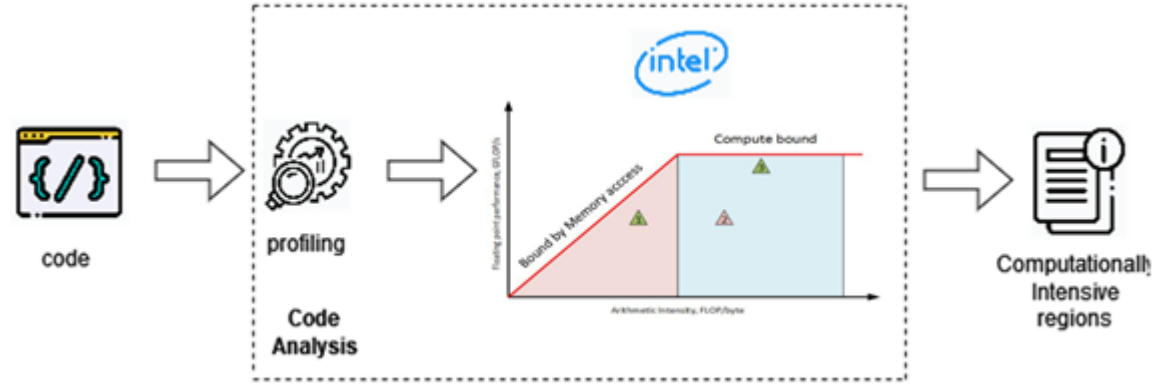

Figure 11 Analyzing the use-case before acceleration

3.2.3 SERRANO's Cloud infrastructure

AUTH will develop a novel server infrastructure coupled with both programmable FPGA accelerators and GPU devices. It will be customized based on the data-centre's application requirements. These FPGA and GPU accelerators will be used not only to increase the performance of servers but also to increase the power efficiency. These devices will be connected in the server though PCI, specifically PCI Express 3.0 x16, an interface standard for connecting high-speed components. Then they will be selected in a seamless and common mechanism for deployment based on the application, requirements, etc. The devices and their specifications are summarized in the following table.

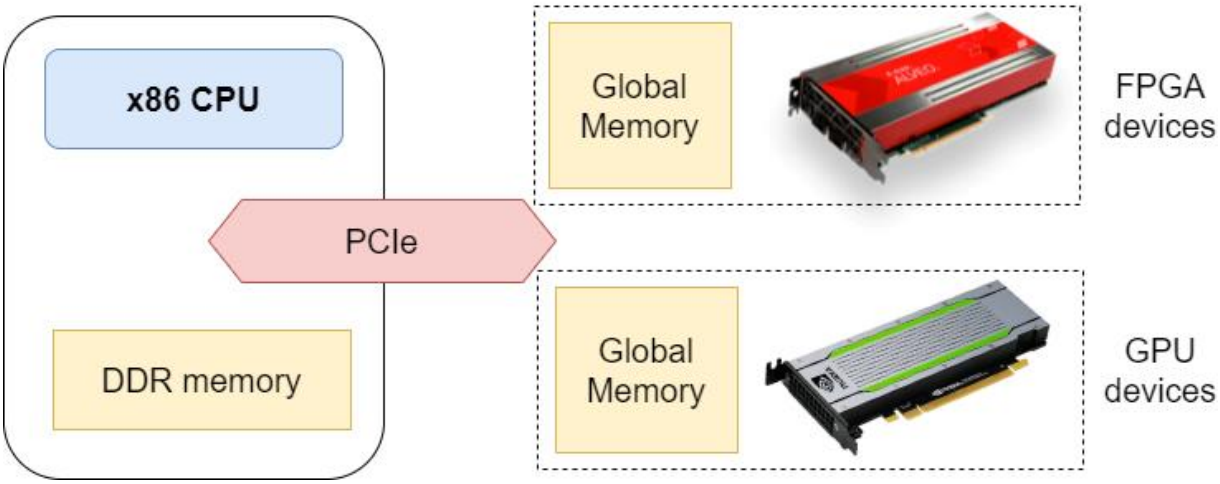

Figure 12: Illustration of the Cloud infrastructure

Table 1: Hardware PCIe Devices lineup - Detailed specs

Device name	Alveo U200 	Alveo U50 	Nvidia T4 
Device Type	FPGA	FPGA	GPU
Width	Dual Slot	Single Slot	Single Slot
Thermal Cooling	Passive	Passive	Passive
PCI Express	Gen3x16	Gen3x16, 2x Gen4	Gen3x16, x8
DDR Total Capacity	64 GB	-	16 GB (GDDR6)
DDR Total Bandwidth	77 GB/s	-	320.0 GB/s
HBM Total Capacity	-	8 GB	-
HBM Total Bandwidth	-	460 GB/s	-
Network Interface	2x QSFP28	1x QSFP28	-
DSP Slices	6840	5952	-
Maximum Power	225W	75W	70W

3.2.4 SERRANO's Edge infrastructure

On the edge side, accelerators are typically deployed as a System on a Chip (SoC) device. This includes the device chip whether it is FPGA or GPU, the processor which is usually an ARM embedded CPU, the DDR memories and the peripherals. The illustration of the edge infrastructure is depicted below showing the architecture of SERRANO acceleration devices that will be employed on the edge for accelerating critical workloads near their operation.

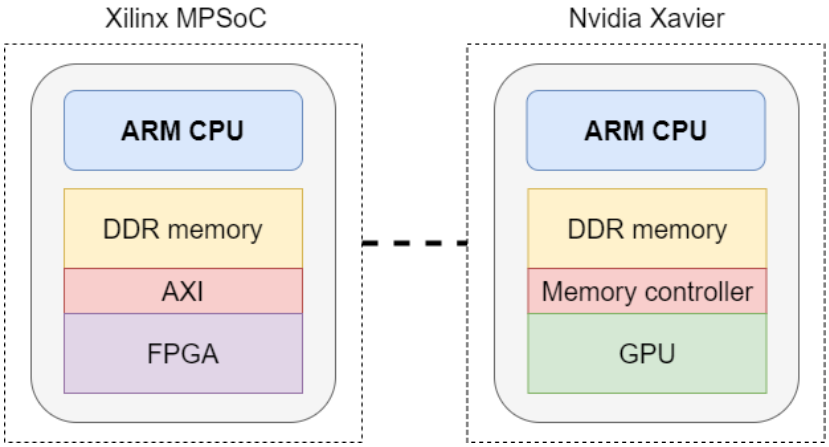

Figure 13: Illustration of the Edge infrastructure

The Table below presents the main characteristics of each platform that will be used from AUTH towards the application deployment and acceleration. There will be a range of devices each with its own assets and weaknesses depending on the workload scenario.

Table 2: Hardware edge devices lineup - Detailed specs

Device name	**ZCU102** 	**ZCU104** 	**Xavier NX** 	**Xavier AGX** 
CPU chip	quad-core ARM Cortex-A53+ dual-core Cortex-R5F	quad-core ARM Cortex-A53+ dual-core Cortex-R5F	6-core NVIDIA Carmel ARM®v8.2	8-core ARM v8.2
GPU	Mali-400 MP2	Mali-400 MP2	384-core Volta GPU+Tensor cores	512-core Volta GPU+Tensor cores
Memory Capacity	4GB 64-bit	2GB 64-bit	8 GB 128-bit	32GB 256-Bit
Memory Bandwidth	20 GB/s	20 GB/s	51.2 GB/s	137 GB/s
Camera Interfaces	USB 3.0	USB 3.0	2x MIPI CSI-2 DPHY lanes	(16x) CSI-2 lanes
Display Interface	64 GB	HDMI and display port	HDMI and display port	HDMI 2.0
PCI Express	PCIe Gen2	PCIe Gen2	PCIe Gen3	PCIe Gen3
DSP Slices	2520	1728	-	-
Maximum Power	15W	15W	15W	30W

4 Description, Analysis and Characterization of use-case applications

4.1 Use-cases Description

4.1.1 Secure Storage

Use-case Description: The first use case focuses on providing a secure, high-performance file storage solution. It builds on the multi-cloud technology adopted by SkyFlok, a commercially available file storage and sharing solution from Chocolate Cloud. The use case expands the purely cloud-based architecture to the edge of the network, in this case the customers' existing IT infrastructure. By leveraging edge storage locations, the latency and throughput of storage operations can be significantly improved. The expansion in edge storage locations makes possible to better tailor the storage service to the user needs. This is achieved through a set of storage policies. These features are part of a refocus from individual end-users to enterprise customers. To complete it, the use case will feature an S3-compatible storage API.

Use-case Applications and/or Workflows Definition:

The storage service showcased by the use case will include the most common endpoints of Amazon's S3 API and will expose separate APIs for managing storage policies as well as storage resources telemetry data. Among the various storage workflows, two stand to benefit most from acceleration: file upload and download. These will be the most commonly called endpoints and their performance will lie at the heart of the use case's evaluation. Figure 13. illustrates both workflows. There are three main data processing steps, encoding, encryption and erasure coding. The first improves cost-efficiency, while the latter two are essential in meeting security, privacy, availability and reliability requirements.

Figure 14: Secure Storage: file upload and download workflows

The first step of uploading a file is compression while the last step of downloading a file is decompression. This simple feature is meant to ensure efficient use of storage resources. The actual technique has not yet been firmly set. Main candidates include Deflate, a widely used algorithm which forms the basis of the gzip file format. Snappy is another interesting choice. It is a more recent technique developed by Google that prioritizes performance over compression ratio. Compression must precede encryption as the, at least in theory, uniformly distributed nature of encrypted data is not suitable for compression.

Security is one of the key aspects of the first use case. File data is kept confidential both in transit and at rest through a combination of encryption, erasure coding and other techniques that act as a set of barriers. Every file that is uploaded to the service is encrypted on the customer's premises before being uploaded to the storage locations. As such, even if an attacker could intercept data either in transit or break into a storage location, it would not be able to decrypt it. The use case employs industry-standard AES encryption using the GCM block scheme. It is included in NSA Suite B Cryptography and is used in a large number of communication protocols such as TLS 1.3, SSH, and IEEE 802.1AE (MACsec) Ethernet security. 256-bit keys are used along with randomly generated IV-s. A detailed description on what steps this use case takes to ensure data privacy and security can be found in Deliverable 3.1 and 3.2.

Finally, the file is erasure coded before it is uploaded to the storage locations. The encrypted data is divided into a number of fragments that are then recombined using a novel erasure code, Random Linear Network Coding (RLNC). The result of the RLNC encoding is a slightly larger number of encoded fragments which can be distributed directly to the storage locations. The extra fragments provide redundancy, necessary to keep data accessible when some of the storage locations are inaccessible. The redundancy also protects against permanent data loss. Finally, RLNC encoding is in essence the creation of new linear combinations of data using randomly selected coefficients. The random nature of the operation as well as the distribution of data to physically separate location has an added privacy-enhancing effect. It forces a malicious user to have to try to break into several locations, then try to solve the system of equations without knowing the coefficients before it is even possible to try to break the encryption.

4.1.2 Fintech Analysis

Use-case Description: The INB use case is related to investment management, specifically portfolio optimization. The portfolio optimization is a process which has the objective to change the distribution of an investment portfolio such that the portfolio has the optimal risk-return. There is not a single solution for the portfolio optimization and it depends on many factors most importantly the changing market conditions as well as the investors characterization such as their objectives, investment capacity and risk tolerance. For example, the market conditions change every second, and the price of a specific share changes continuously. At the same time, the investor may be interested in long term investment (e.g. 10 years) or short term investment (e.g. 2 years), has high investment capacity (e.g. annual income of 100.000 EUR and wealth of 1M) or low investment capacity (10.000 EUR and wealth

of 10.000 EUR). The portfolio optimization will generate different portfolio distribution for these two investors. The use case is detailed in more details in D2.2 and D2.3.

Use-case Applications and/or Workflows Definition: Figure 15` summarizes the investment management in five step which run continuously.

The **market analysis** analyses and classifies the investment instruments. The analysis includes calculating many different technical indicators for one or more instruments, either individually or collectively. Example indicators that are calculated include return, risk, distribution, moving averages, volatility, Treynor measure, Sharpe ratio, Sortino ratio, Jensen measure, drawdown, underwater, FFT, and many others. Typically, these indicators are calculated for different time windows, for example 1, 2, 3 ... 60 months. Because of the high number of possible combinations that exist, the market analysis can be a very compute intensive phase. After the metrics are calculated, the results are further analysed in order to identify what can be the best instrument or set of instruments to use in portfolios.

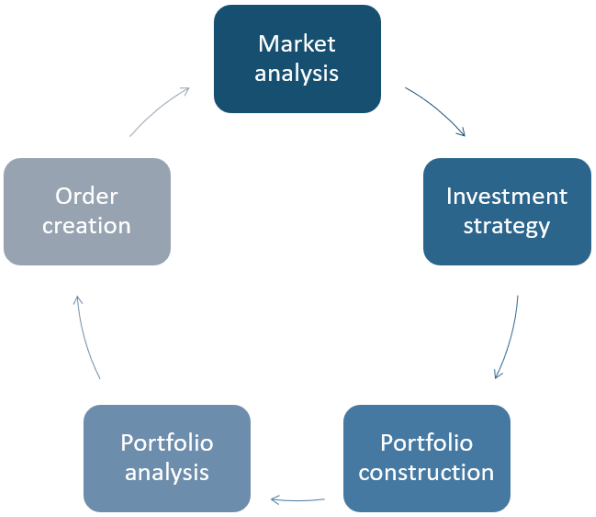

Figure 15: Investment management workflow summarized in five continuously repeating steps.

During **investment strategy** phase, the results from the market analysis are analysed with respect to the investor's own profile such as risk tolerance, investment capacity, objectives, horizon and other criteria. Example strategies are speculation for rapid and risky growth, long term but volatile growth or low risk slow growth for savings. During this phase, different investment instruments are classified as suitable or not suitable for different investment strategies.

The **portfolio constructions** combine the results from the market analysis and the investment strategy in builds investment profiles. Investment profiles, are template portfolios, which are composed of the instruments analysed during market analysis and shortlisted after applying investment strategies. Additionally, and most importantly, besides choosing what to buy or sell, this phase also determines what should be the distribution of each investment instrument in the investment profile. Higher profile means higher expected return but also higher risk to lose money due higher volatility. During this phase, a lot of what if portfolio analysis and

simulations are made in order to estimate the expected return and the probability distribution in time. The analysis can be very extensive and compute intensive due to the huge number of combinations that exists between investment instruments, their distributions and the investment horizon.

The last step in the investment management is the **order creation** or also what InbestMe refers to as **rebalancing**. During the rebalancing, orders are created to make the actual distribution of the portfolio to match the investment profile to which the portfolio is assigned to. For example, if the investment portfolio is profile 8 and is slightly off from the distribution that is expected to be. For example, the cash of the portfolio is expected to be 1% but it is 3.77%. As a result, during the rebalancing, orders will be created and executed in order to align the actual portfolio to match the expected distribution.

4.1.3 Anomaly Detection in Manufacturing Settings

Use-case Description:

Companies that manufacture expensive parts with high added value are very demanding in terms of machine availability and quality assurance. Predictive maintenance, evaluation of remaining useful life, and diagnosis of critical machine elements are state-of-the-art practices. However, some of the techniques used require the machine to stop before performing the analysis. As a result, the various hardware components are idle most of the time, waiting for analysis procedures to begin, something the manufacturing industries are eager to avoid.
Downtime for faulty devices in an industrial plant must be kept to a minimum to achieve high system availability. In the competitive world of manufacturing, getting the most out of your machine can be the difference between being competitive or not. Therefore, and mainly after the emergence of the Industry 4.0 paradigm, many techniques and methods are being widely applied to meet a simple but complex objective: to keep the machine running most of the time.
This UC aims to detect machine's ball-screws anomalies, by processing the amount of data generated in real-time by high-frequency sensors.

Use-case Applications and/or Workflows Definition:

Figure 16 describes a high-level workflow for the use case for a single machine. When the machine starts machining, data is continuously / periodically generated. Data is being monitored by mechanism running on the SmartBox attached to the machine. When new data is found, the Stream Processor Connector sends it to a Stream Data Processing Tool. This could be a Kafka-like mechanism or any processing tool provided by SERRANO.

The Stream Processor Connector sends the data to the Data Processing Layer, where the correspondent processor application is triggered depending on the nature of the data. Data coming from accelerometers is sent to the Acceleration Processor, while data coming from Temperature sensors will be send to the Temperature Processor.

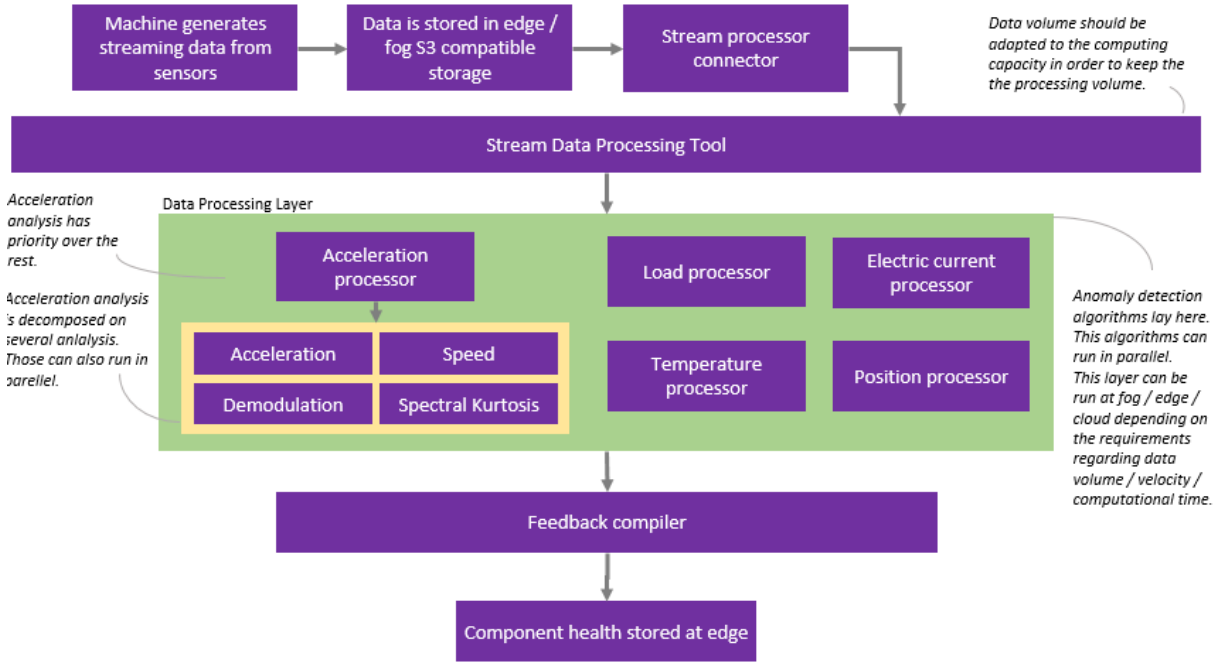

Figure 16:High level use-case workflow

Data processed at the Data Processing Layer may be processed at edge, fog or cloud resources depending on the client requirements. This use case will integrate the cloud part with the UC1, which provides a secure storage layer within SERRANO.

These Processing Applications analyse the incoming data. As an example, the Acceleration Processor may use Fourier Transform to look for unhealthy symptoms over frequencies; the Load Processor may use some anomaly detection algorithm to look for peaks. In summary, here is where system intelligence resides.

The computations results of the Processing Applications are sent to a Feedback compiler layer and these results are stored at the edge for further notifications.

With this architecture in mind, the UC proposes to demonstrate two application intents:

- Position processor intent
- Acceleration processor intent

The first intent will try to integrate *Kmeans* and *DBScan* kernels. Autoencoders are also among the algorithms proposed to demonstrate in this intent. The second intent will try to be demonstrated using *KNN* and *FTT* kernels.

The high level architecture diagram of both intent applications could be simplified this way:

Figure 17: High level architecture

4.2 Performance analysis & Characterization of use-case applications

4.2.1 Secure Storage

AES-GCM

AES-GCM (Advanced Encryption Standard with Galois Counter Mode) is a block cipher mode of operation that provides high speed of authenticated encryption and data integrity and is suitable to be employed in communication or electronic applications. It consists of two main functions, block cipher encryption and multiplication and it can either encrypt or decrypt with 128-bit, 192-bit or 256- bit of cipher key. For the SERRANO use case we consider a 256-bits AES-GCM.

This algorithm is designed to provide both data authenticity (integrity) and confidentiality while it is defined for block ciphers with a block. **Galois Message Authentication Code** (**GMAC**) is an authentication-only variant of the GCM which can form an incremental message authentication code. Both GCM and GMAC can accept initialization vectors of arbitrary length.

In order to analyse the AES-GCM algorithm and characterize its regions that are to be accelerated, a roofline analysis is performed with Intel Advisor tool. The roofline model illustrates the arithmetic intensity of the most computational parts of the application.

Figure 18 depicts its output roofline, where both green and yellow dots represent the various functions of the AES workload.

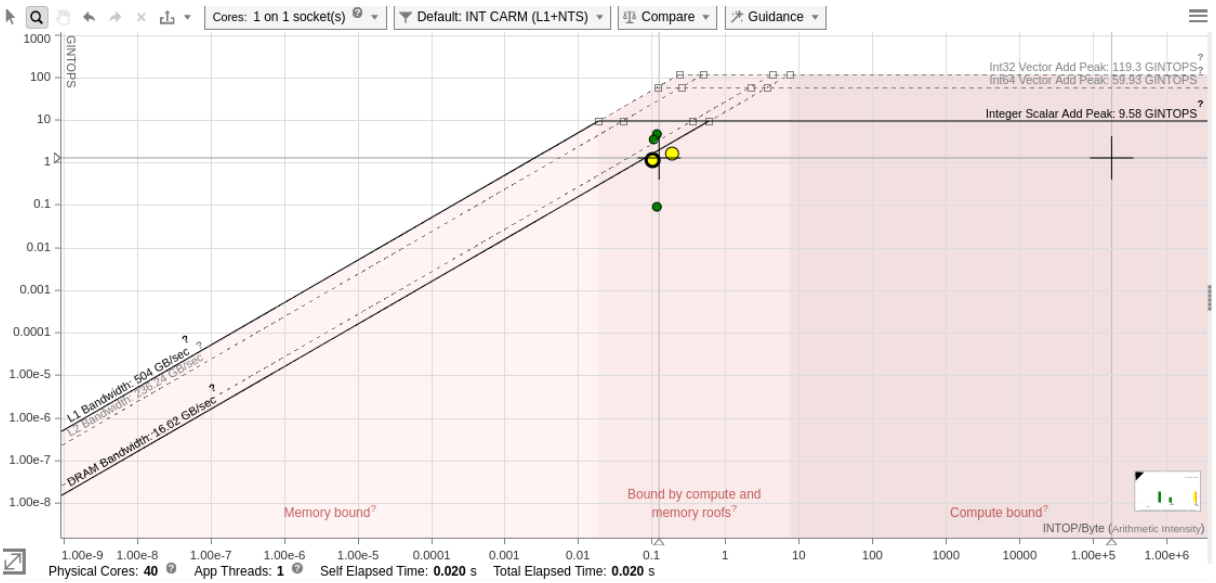

Figure 18: Roofline model of AES-GCM

The roofline model illustrates the arithmetic intensity of the most computational parts of the application. Note that the 2 yellow dots stand for the encryption and the decryption methods accordingly and as demonstrated in the figure, they are both bounded by compute and memory roofs. However, the fact that they do not belong to the memory bounded tasks encourages us to achieve successful accelerations as they seem to execute more operations than transferring input bytes.

Afterwards, we profile the algorithm on 2 different CPUs, a x86_64 80-core Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz and a 6-core A57 ARM @ 2GHz. We encrypt and decrypt a 32 MB input file and we record the latencies. Figure 19 and Figure 20 depict the recorder execution latencies of encryption and decryption workloads and the rest of the application that constitutes mainly read/write and memory transfers operations.

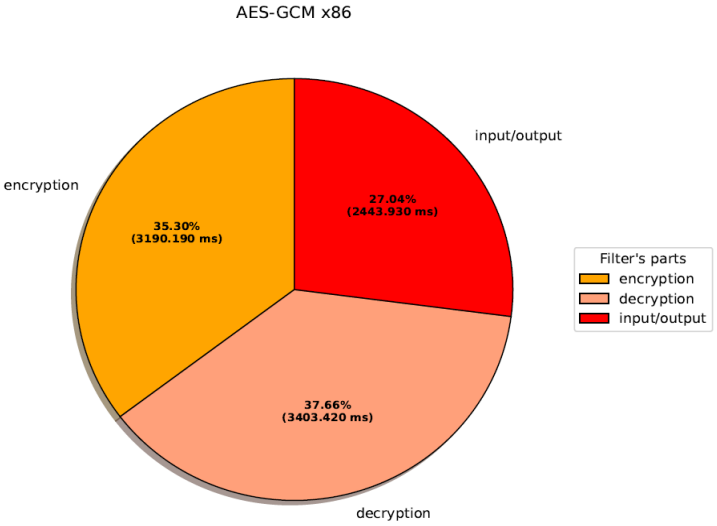

Figure 19: Execution time breakdown for x86 architecture

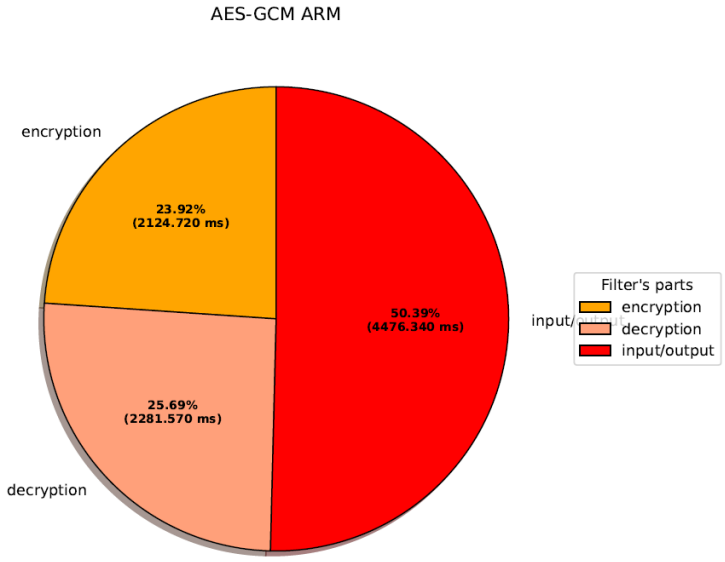

Figure 20: Execution time breakdown for ARM architecture

It is clear that despite the fact that the input consumes a lot of execution latency, there is a lot of room for acceleration of both the encryption and decryption functions

Random Linear Network Coding - Erasure Coding

Random Linear Network Coding (RLNC) is a coding scheme that maps the input data to encoded output symbols through finite field arithmetic operations. In the context of the specific scenario, RLNC erasure coding is used for encoding and decoding data as described below.

1. **Encoding**
   The encoding process operates by dividing the input data to a predefined number of chunks, each chunk of the same size. Then, the extracted data fragments are multiplied with pseudo randomly generated coefficients over the Galois Field (2^8), forming the encoded symbols. This process is illustrated in Figure 21 Figure 21 Encoding Mechanism below.

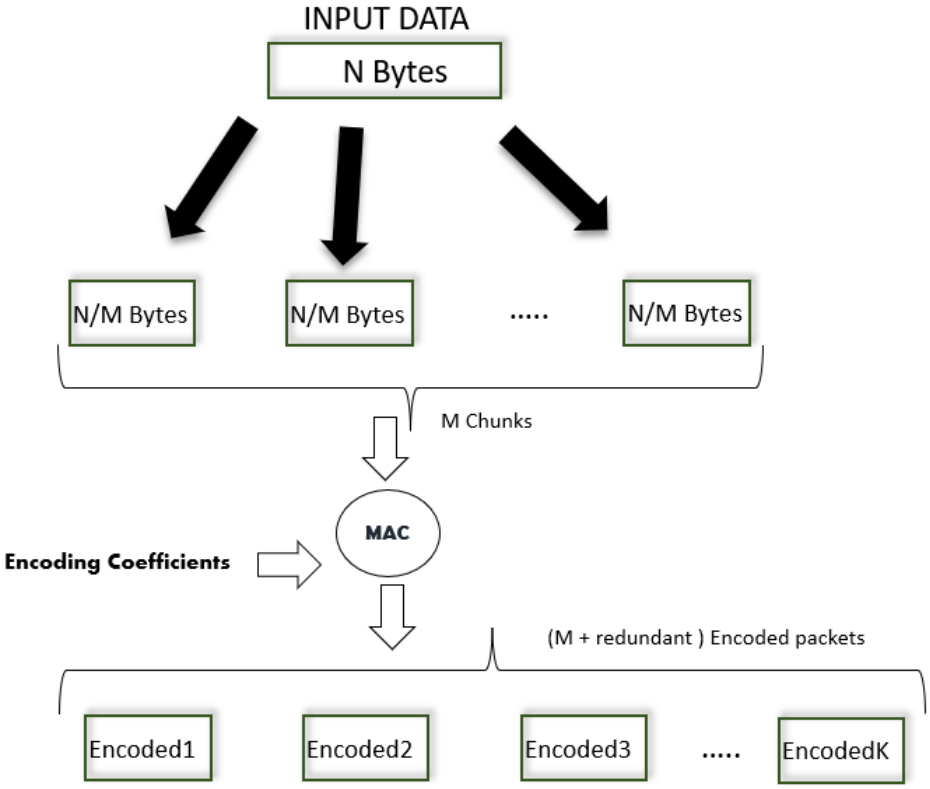

Figure 21 Encoding Mechanism

2. **Decoding**
   The packets that are produced from the encoding process are linear combinations of the pseudo random coefficients and the chunks of the input data. To decode those packets, gaussian elimination is performed on the system of equations that was created from the encoding process. Once the equations have been solved, the fragments of the data have been recovered. At the final phase of the decoding

process, the recovered data slices are glued together and the original input data is formed.

To analyse the arithmetic intensity of the encoding and decoding algorithms the roofline model for those two tasks was generated as depicted in Figure 22 below.

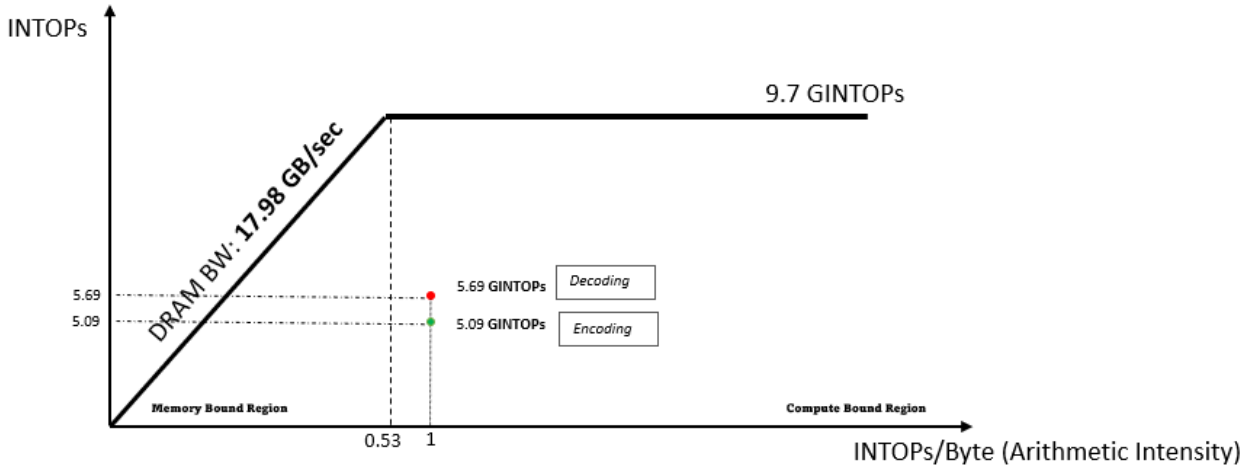

Figure 22 Roofline model of the encoding and decoding tasks

The operations for the encoding and decoding task are performed in the GF(2^8) finite field and hence only 8 bit integer values are used. Based on the roofline model one integer operation is performed per input byte. The arithmetic intensity of 1 INTOP/Byte is higher than the ratio ***peak performance / peak bandwidth*** and hence the algorithms can be characterized as compute bound applications.

At the next step, the scenario of applying the encoding and decoding algorithms on 100 KB input data was tested for two reference architectures (x86 and ARM). Figure 23 and Figure 24 depict the execution time breakdown for the two architectures for the encoding algorithm while Figure 25 and Figure 26 exhibit the same analysis for the decoding part. It is noted that all the following experiments were executed 10 times on the corresponding architectures and the average execution times were kept.

The encoding process consists of two parts. At first, the coding coefficients are generated through a pseudo random generator and then the encoding task that performs the multiplications on the Galois Field is invoked. Based on those figures most of the time is spent on the encoding process while only 0.037 and 0.12 msec are spent on the task of generating the coefficients for the x86 and ARM architecture respectively.

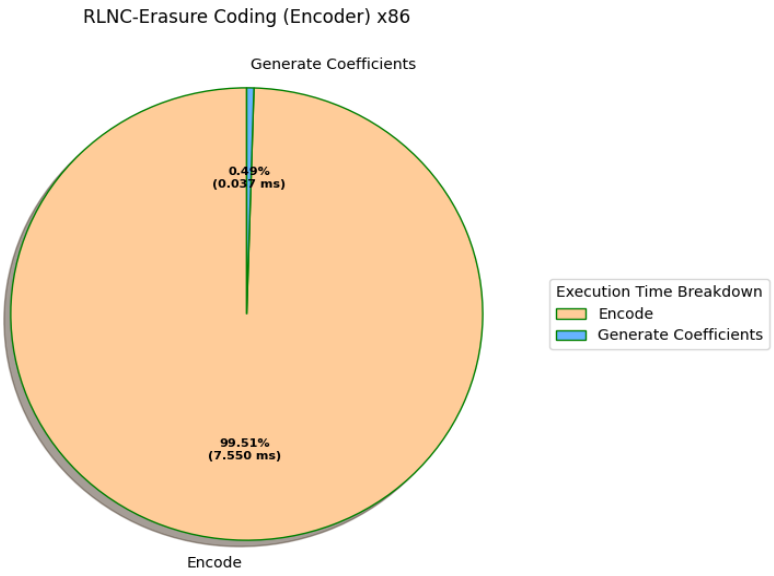

Figure 23 RLNC encoding (x86) execution time

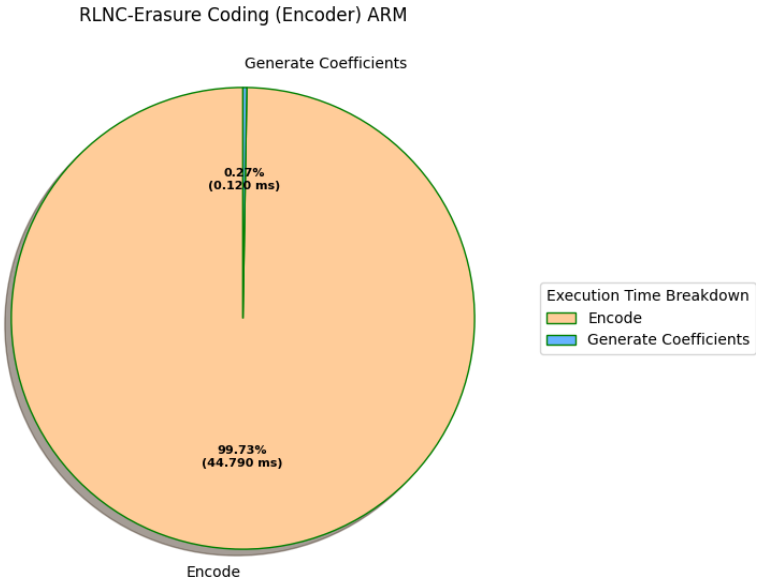

Figure 24 RLNC encoding (ARM) execution time

Similarly for the decoding part. The time that is required for performing the gaussian elimination and extracting the encoded data fragments is responsible for at least the 98% of the algorithm's latency. On the contrary, the process of gluing the extracted slices together and recovering the original data takes only 1.4 % and 0.31 % of the total execution time when this task is executed on an ARM and on a x86 system.

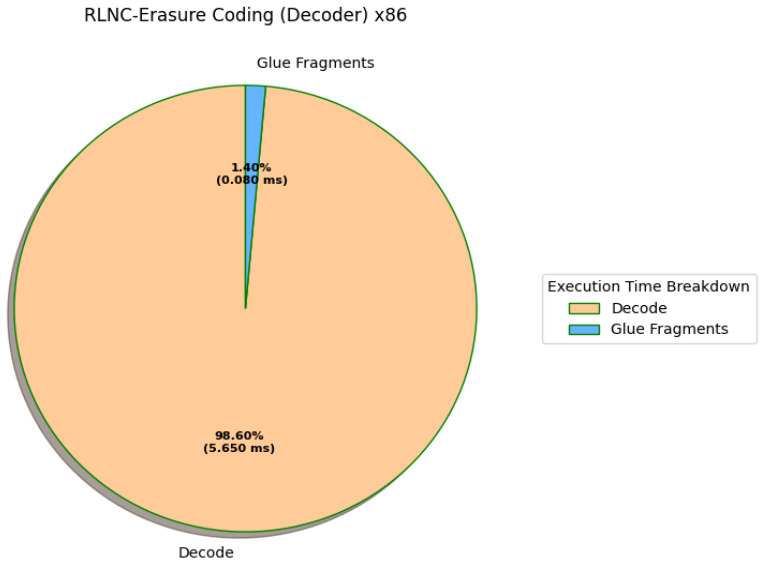

Figure 25 RLNC decoding (x86) execution time

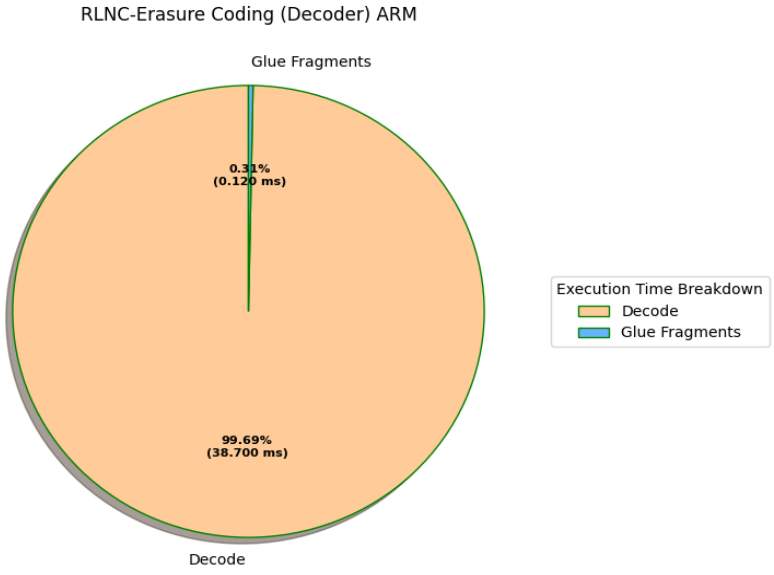

Figure 26 RLNC decoding (ARM) execution time

At the last step of the profiling phase, the total latency of the encoding and decoding processes was measured for three different granularities. Figure 27 and Figure 28, show the overall execution time for 1KB, 10KB and 100KB input data for the x86 architecture.

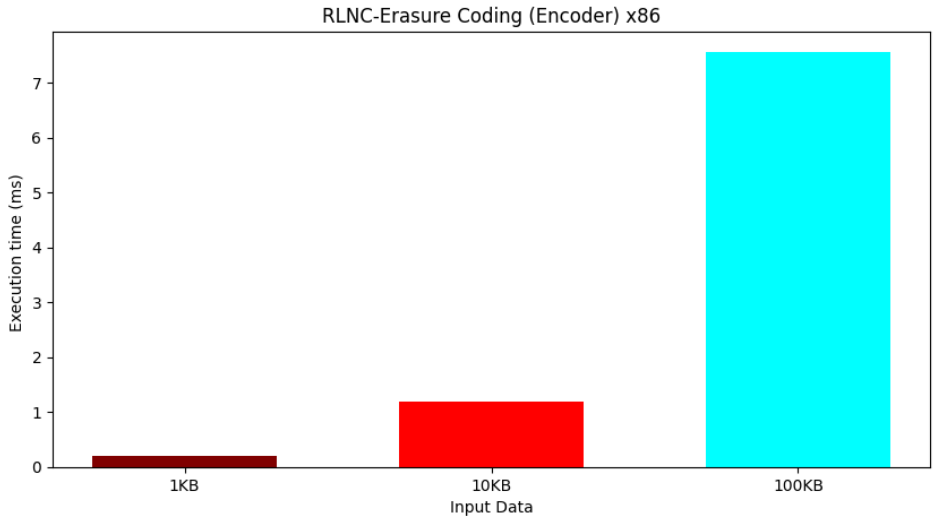

Figure 27 Encoder overall execution time (x86)

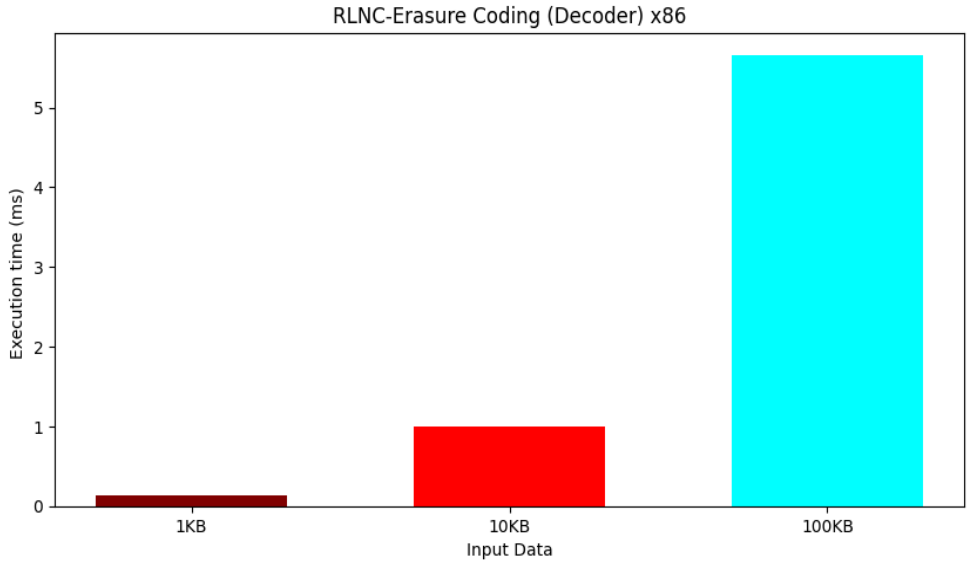

Figure 28 Decoder overall execution time (x86)

Similarly, Figure 29 and Figure 30 show the overall execution times for an ARM architecture.

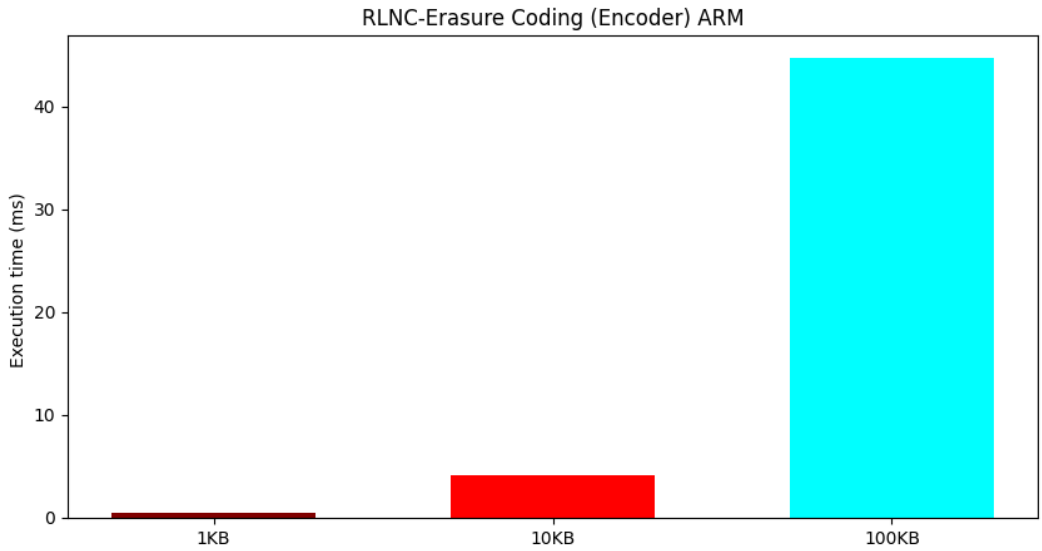

Figure 29 Encoder overall execution time (x86)

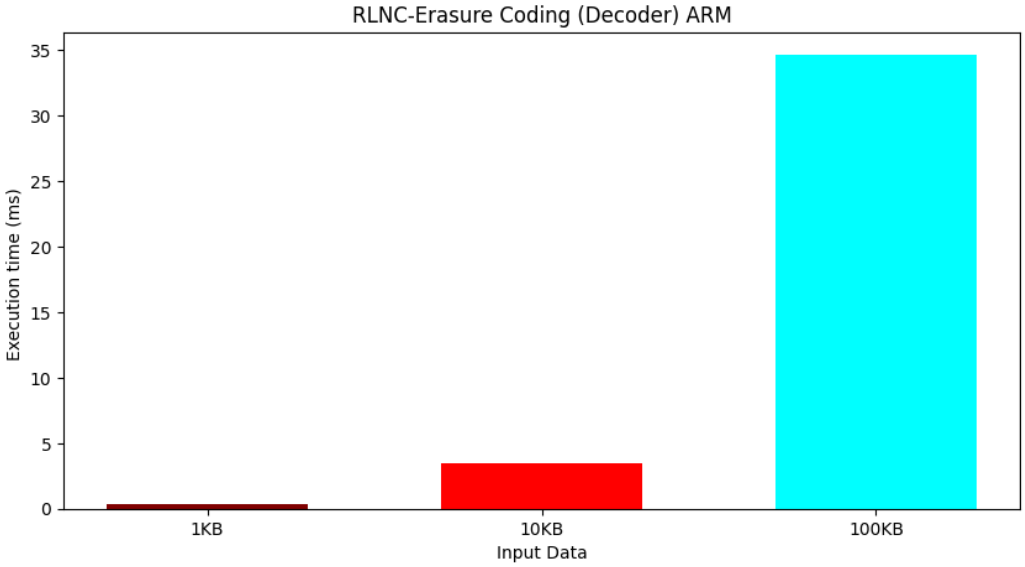

Figure 30 Decoder overall execution time (ARM)

Based on the profiling analysis, the encoding task that performs the finite field arithmetic operations and the decoding part that performs the gaussian elimination for solving the system of equations and recovering the encoded data chunks, have potential for acceleration. Those two algorithmic parts can be accelerated on hardware platforms.

4.2.2 Fintech Analysis

Savitzky-Golay Filter

The Savitzky-Golay filter is a digital filter that is frequently used for smoothing time series data by fitting adjacent data points with low-degree polynomials. The number of the adjacent data points is described as the filter's ***window.***

The filter's coefficients are solely dependent on the desired polynomial order and on the window size. Hence, for a given polynomial order and a window the filter's coefficients are fixed.

The smoothing process is performed by applying discrete convolution between the signal's coefficients and the *window* data points. Once, the convolution for a specific *window* has been completed the *window* is shifted and the convolution is performed again for the next set of data. The below figure depicts a signal smoothed by applying the Savitzky-Golay filter with a polynomial order of three and a *window* size of 10 on a time series composed of 200 data points.

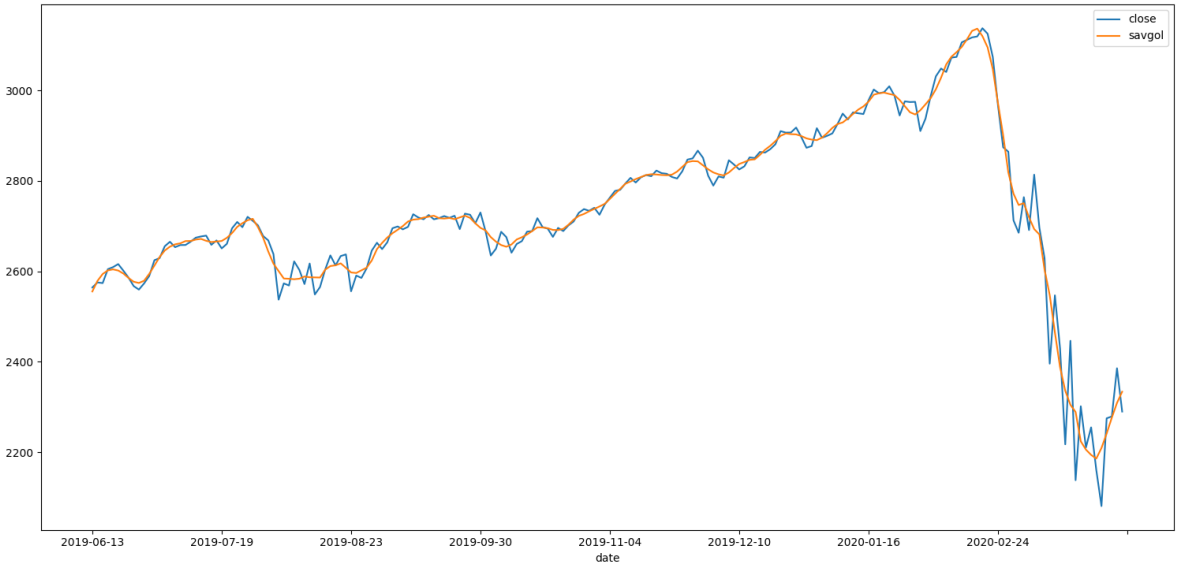

Figure 31 Smoothing a time series

The Savitzky-Golay filter consists of two algorithmic parts.

1. Firstly, the filter's coefficients are calculated based on the filter's polynomial order and on the *window* size.
2. Secondly, a discrete convolution is performed between the input signal and the filter's coefficients. At first, the *window* is placed on the 1st input value and the filter's coefficients are multiplied with the first *half window* values gradually by shifting the window by one after each iteration. A zero padding is applied on the input signal in order to perform convolution for the first *half window* iterations. Subsequently, to perform

convolution for the last *half window* iterations, a padding with the last input value is applied.

The described process is illustrated in the example Figure below.

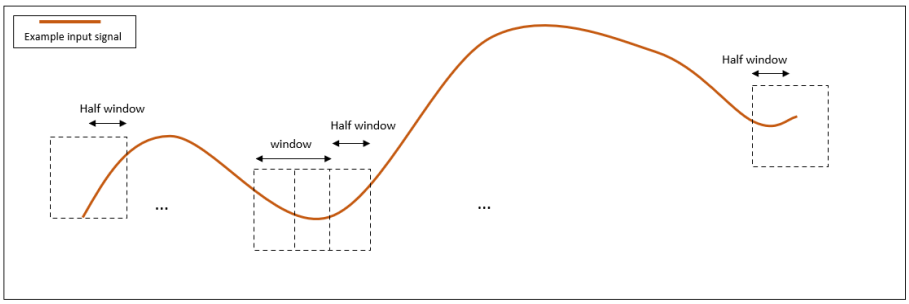

Figure 32 Moving window convolutions

To analyse the specific algorithm, characterize its arithmetic intensity and determine the algorithmic regions that should be accelerated on hardware platforms, a three-step profiling process is followed as described below.

At first, a roofline analysis is performed and the filter's roofline model is extracted as illustrated in Figure 33.

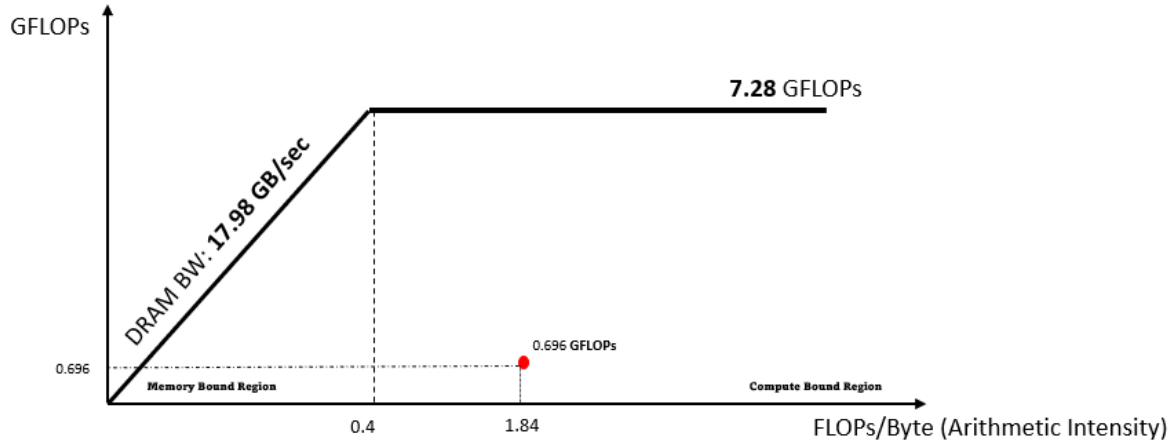

Figure 33 Savitzky-Golay filter roofline model

The roofline model shows the arithmetic intensity for the convolution stage. In this part of the algorithm's execution most of the operations are performed and the output signal values are calculated, therefore this algorithmic region is depicted with a red dot in Figure 33 above. The arithmetic intensity of 1.84 indicates that more operations are performed per input byte and that this part of the algorithm's execution is computationally intensive.

The second step of the profiling phase includes running the algorithm on different reference architectures and measuring its execution time for a given input signal. The pie charts in Figure 34 and Figure 35 below show the filter's execution time breakdown for two computing architectures (x86 and ARM). The results in the following figures were obtained using one

input signal composed of 20000 data points which translates to applying the Savitzky-Golay filter on a single financial asset.

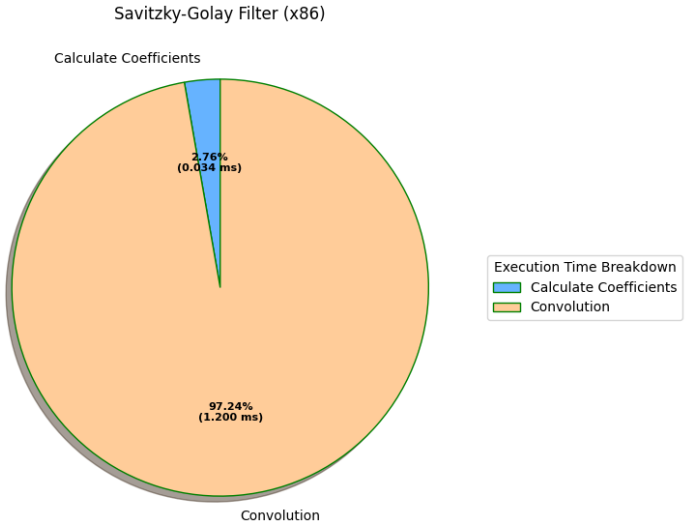

Figure 34 Filter's execution time (x86)

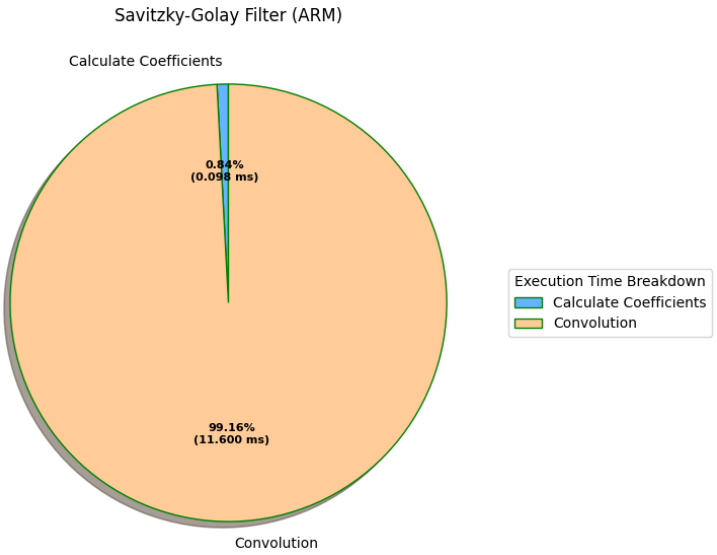

Figure 35 Filter's execution time (ARM)

It is shown on the pie charts above that most of the algorithm's execution time is spent on the convolution operations. The calculation of the filter's coefficients is dependent on the filter's parameters (window size and polynomial order). Hence, for a given type of Savitzky-Golay filter the latency of this algorithmic part is fixed. Therefore, the latency overhead from calculating the filter's coefficients should be taken into account in the case that different types of filters are applied on few small input signals.

At the third part of the profiling phase the overall's algorithm execution time was measured for the two reference architectures and for multiple assets. In particular, the scenarios of applying the Savitzky-Golay filter on 10, 20 and 100 different signals, each signal composed of 20000 values, were tested. The extracted execution times are depicted in Figure 36 and Figure 37 below.

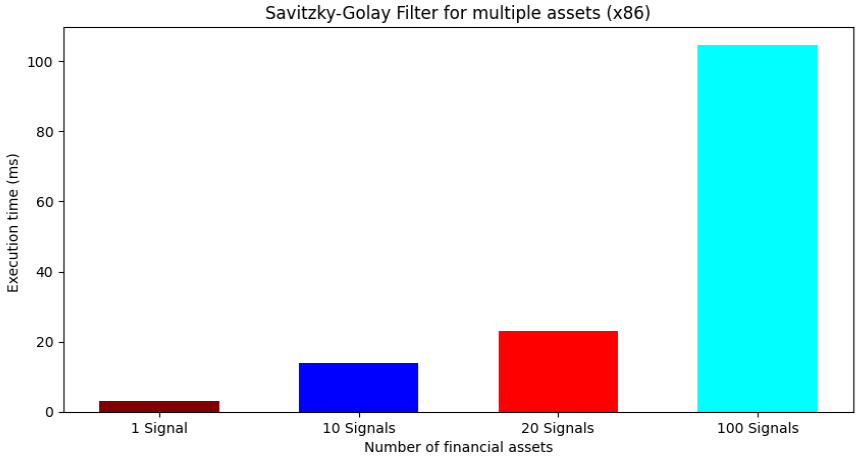

Figure 36 Execution time for multiple assets (x86)

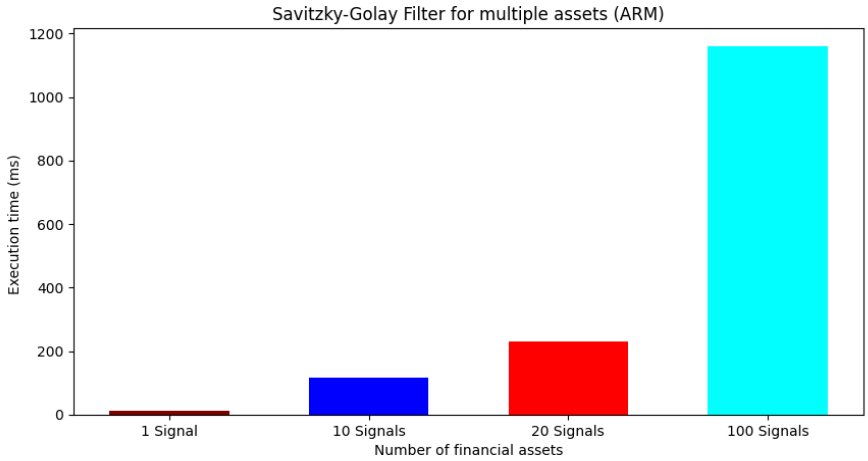

Figure 37 Execution time for multiple assets (ARM)

The application of the Savitzky-Golay filter on multiple financial assets has a latency that varies from 1.2 to 112 ms in the baseline x86 architecture and from 11.6 to 1158 ms for the ARM system respectively. Based on the above analysis, AUTH will accelerate on hardware platforms the scenarios of executing the Savitzky-Golay filter for multiple financial assets, performing optimizations that target to minimize the overall latency on edge (ARM) and on cloud (x86) devices.

Kalman Filter

The Kalman filter, also referred to as linear quadratic estimation, is used to estimate unknown variables based on a series of noisy measurements. It has a wide range of applications in

guidance, navigation and finance. Kalman filtering is based on a linear dynamical system discretized in the time domain.

Suppose there is a process model given by x_k:

$$x_k = F * x_{k-1} + B * u_{k-1} + w_{k-1}$$

where k is the discrete time step, while the measurement model is:

$$y_k = H * x_k + v_{k-1}$$

where w and v are the noise terms, which are added into the system. The purpose of the filter is to estimate the state value x_k that is represented as $xbar_k$. To describe how the Kalman filter works, the following parameters should be defined:

• Q = cov(w), R = cov(v)

• P represents the error covariance between x_k and $xbar_k$

• K represents Kalman gain

• H and F are constant terms

The Kalman filtering calculates the following three lines of code for each data point.

$$K = \frac{P * H}{H^2 * P + R} \quad Kalman\ gain\ update$$

$$xbar_k = xbar_k + K * (x_k - H * x_k) \quad x_k\ approximation$$

$$P = (1 - K * H) * P + Q \quad Error\ estimation\ update$$

Figure 38 shows the filter's output when we use as input signal the closing prices dataset provided by InbestMe.

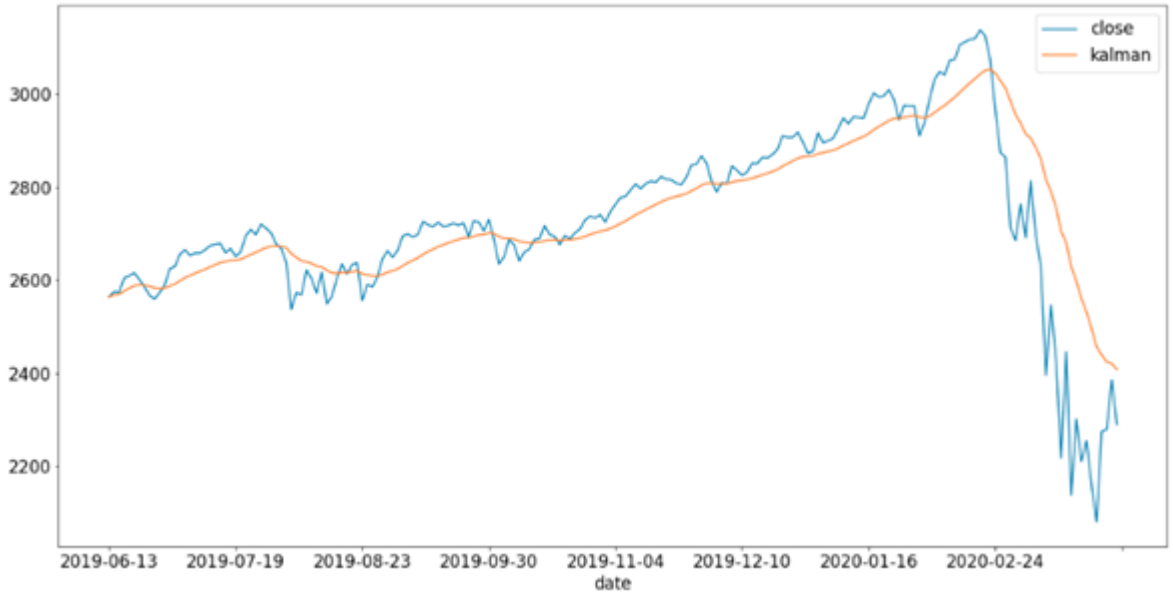

Figure 38 Kalman filter's output using InbestMe's closing prices dataset

In order to identify the computationally intensive parts of the filter, which are going to be implemented on hardware, the aforementioned methodology was followed. Figure 39 demonstrates the roofline diagram created using Intel Advisor.

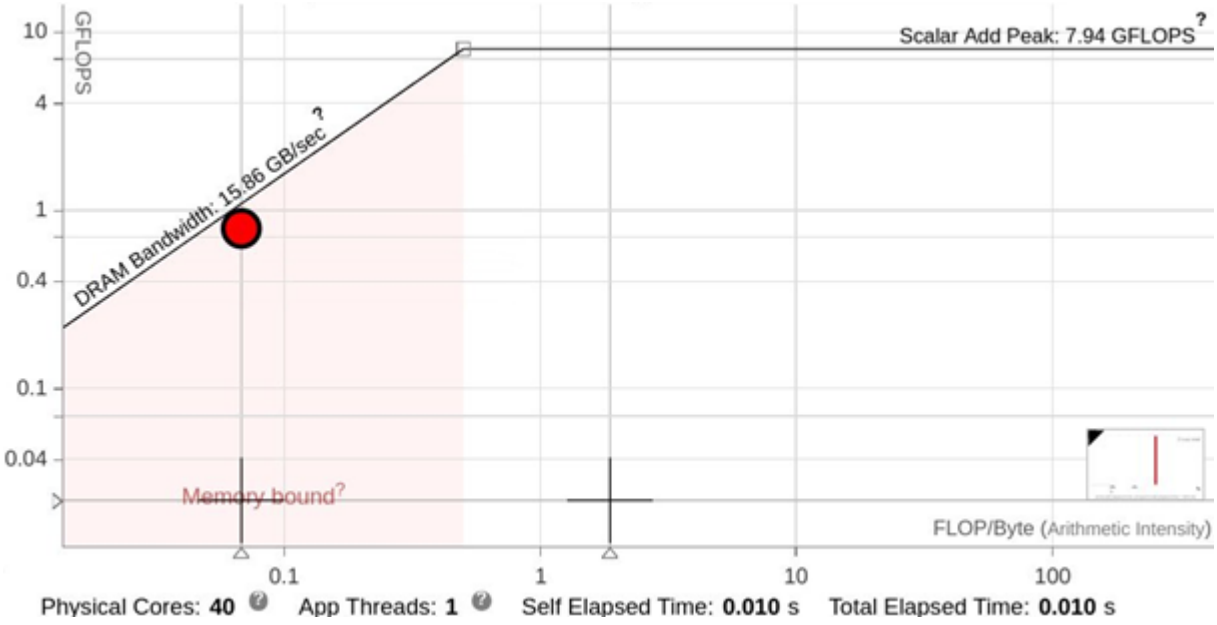

Figure 39 Roofline model for Kalman filter

The roofline model shows the arithmetic intensity for the Kalman filtering computation loop which is around 0.068 FLOPs/Byte. The Kalman filter is a memory bound algorithm, as depicted from the red circle in the plot, meaning that the rate at which the filter processes input is limited by the amount of memory available and the speed of the memory accesses.

The second step of the profiling phase includes running the algorithm on different reference architectures and measuring the execution time of read/write operations as well as the execution time of the main operations of the filter for a given input signal. The results of the following figures were obtained using a single financial asset (20000 data points).

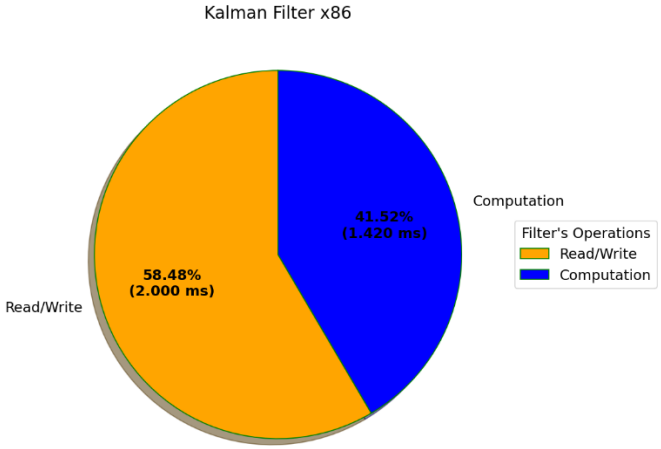

Figure 40 Kalman filter execution time breakdown for x86 architecture

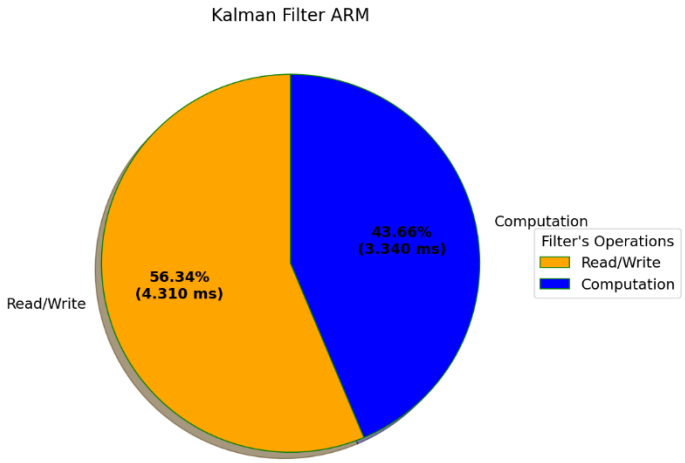

Figure 41 Kalman filter execution time breakdown for ARM architecture

Figure 40 and Figure 41 show that on both reference architectures the algorithm requires at least 56% of the overall execution time on read/write operations, verifying the algorithm's memory bound nature.

Finally, the execution time of the Kalman filter is measured for x86 and ARM architecture, for 1, 10, 20 and 100 different assets. Each asset is a signal composed of 20000 values just like in the case of Savitzky-Golay filter. The extracted execution times are depicted on the following figures.

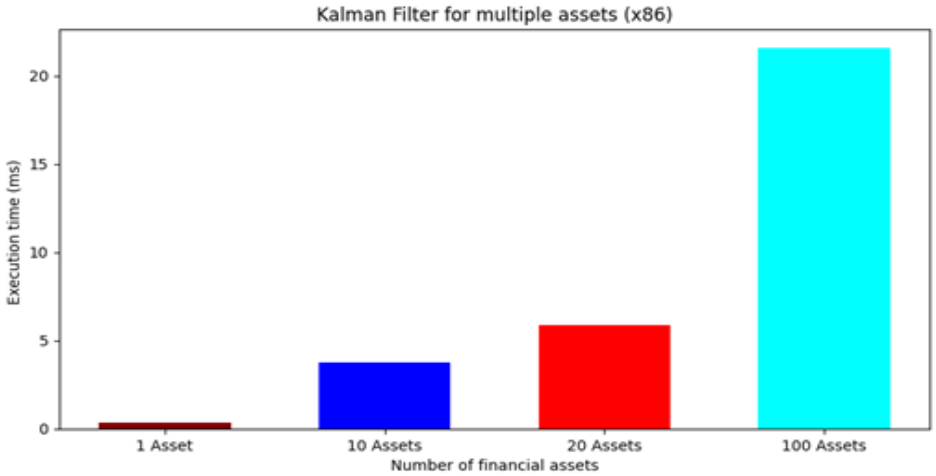

Figure 42 Execution time for multiple assets for x86 architecture

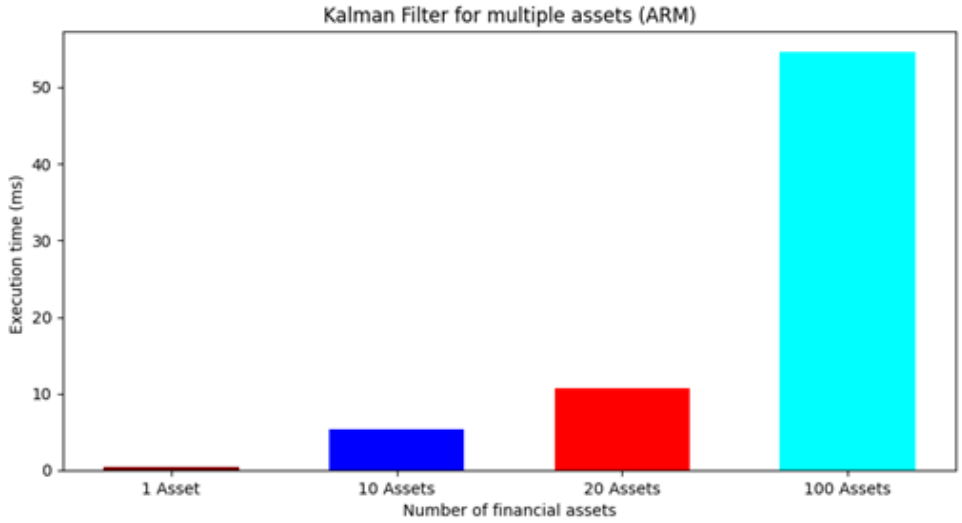

Figure 43 Execution time for multiple assets for ARM architecture

The application of the Kalman filter on multiple financial assets has an execution time that varies from 0.37 ms to 21.55 ms in the baseline x86 architecture and from 0.52 ms to 54.53 ms for the ARM system respectively. Based on the aforementioned observations, AUTH will accelerate the algorithm in order to minimise its latency when multiple assets are used.

Wavelet Filter

Discrete time wavelet transforms have found engineering applications in computer vision, pattern recognition, signal filtering and most widely in signal and image compression. A wavelet is a waveform of effectively limited duration that has an average value of zero and nonzero norm. In numerical analysis and functional analysis, a discrete wavelet transform (DWT) is any

wavelet transform for which the wavelets are discretely sampled. As with other wavelet transforms, a key advantage it has over Fourier transforms is temporal resolution: it captures both frequency and location information (location in time).

As in the previous subsections, we make use of the Intel Advisor tool in order to extract the use case's roofline model that is depicted in Figure 44. The roofline depicts 3 dots that correspond to 3 functions of the body of wavelet application. We are interested in the red dot (dwt_sym_strid function) that constitute the most time-consuming task of the wavelet filtering. From its roofline model, wavelet's dwt_sym_strid function constitutes a computation bounded time consuming part that needs CUDA acceleration.

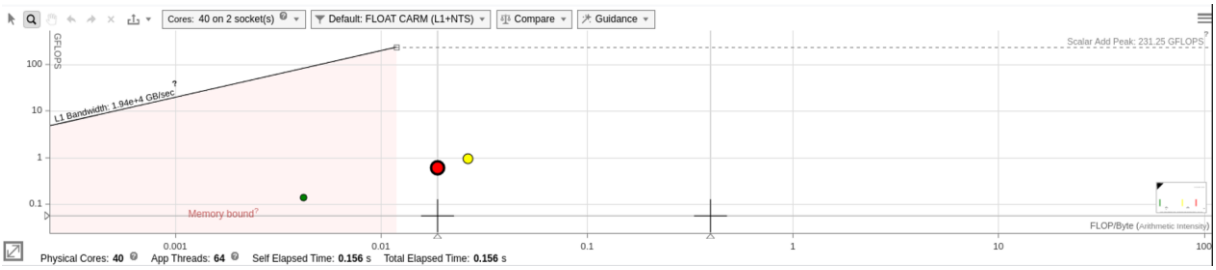

Figure 44 Roofline model for wavelet transform

Additionally, we evaluate our use case on both SERRANO's x86 and ARM CPU architectures for a given input signal of 10000000 data points, equivalent to 500 assets of 20000 data points (maximum size of dataset according to InBestMe) in order to estimate the execution latency of the compute intensive part of this use-case.

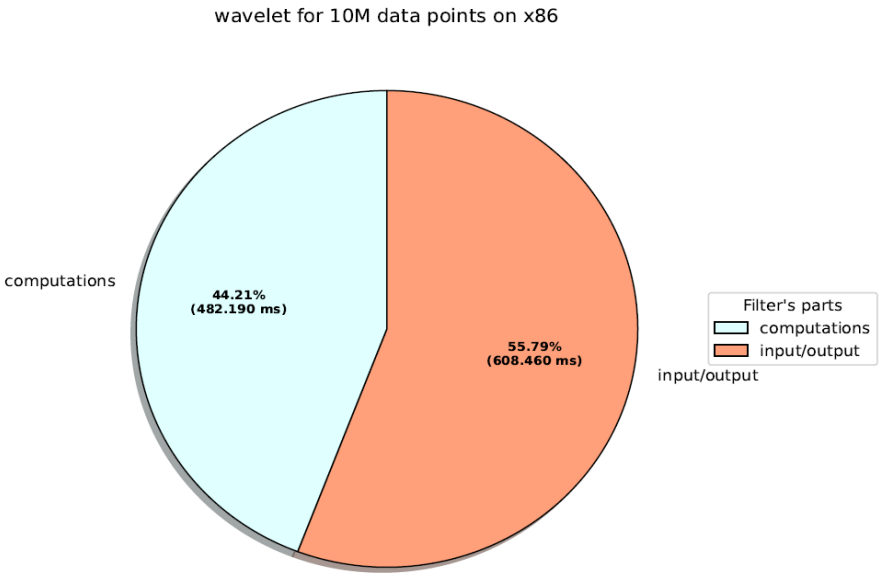

Figure 45 Wavelet filter execution time breakdown on x86 CPU

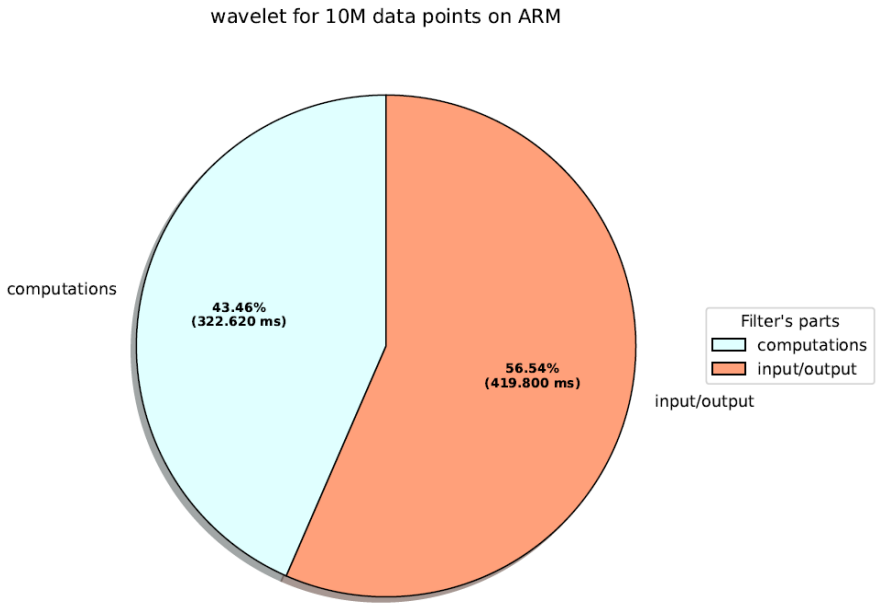

Figure 46 Wavelet filer execution time breakdown on ARM CPU

It is, therefore, clear that the wavelet's computationally intensive function dwt_sym_stride () consumes big part of the total execution latency (44% and 43% for the x86 and the ARM CPUs respectively) that needs acceleration.

4.2.3 Anomaly Detection in Manufacturing Settings

Density-based spatial clustering of applications with noise

Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu in 1996. It is a density-based clustering non-parametric algorithm, meaning that given a set of points in some space, it groups together points that are closely packed together, marking as outlier points that lie alone in low-density regions. DBSCAN is one of the most common clustering algorithms and also most cited in scientific literature.

The density of instances is a key concept in the DBSCAN algorithm. The intuition is that the most common patterns will be close to each other, whereas the anomalous ones will be far from the nominal ones. By computing the pairwise distance measures of the instances, the areas that fulfil some criteria are identified and the instances are grouped. These criteria are specified by two parameters that define when two instances are considered close as well as the minimum quantity of close instances needed to consider an independent cluster. There is also the possibility for DBSCAN to not classify certain instances in a particular cluster, if they are too far from the others.

The hypothesis behind the use of this algorithm is that, in general, the cycles carried out in the machine are going to be, somehow, similar (with light variations). These are considered normal cycles. If an anomalous cycle is carried out, the pattern that characterises the backlash

will be different, so the distance between the other instances is going to be too high in order to be considered part of the cluster that constitutes the normal instances. Therefore, it is not going to be grouped in the "normal" cluster. If the failure continues for a long time, and the anomalous patterns are similar between them, a new cluster is going to be created, with the last anomalous patterns. As the elements are deteriorating, more anomalous patterns will be monitored, and there may be more clusters that represent different degradation levels.

Figure 47 shows the output of the DBSCAN algorithm using the twenty-two signals provided by IDEKO. The algorithm creates two distinct clusters, one for the regular signals (indicated as *not anomaly*), where the majority is located, and one for the anomalous ones (indicated as *anomaly*). In this way, the UC provider will be able to identify malfunctions and repair the machine before failure.

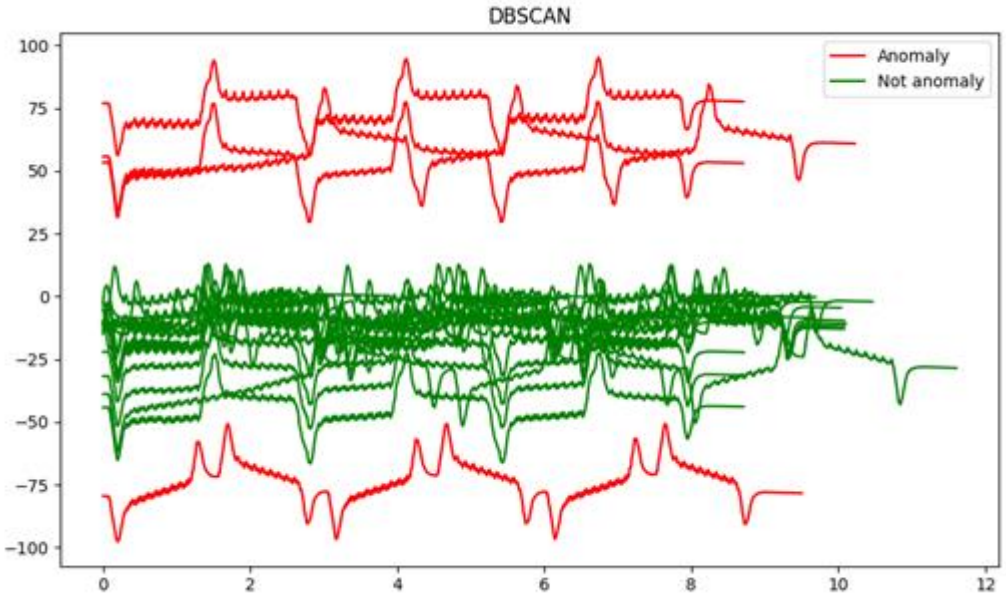

Figure 47 DBSCAN's output in IDEKO's UC

The provided python source code uses a library that implements the DBSCAN algorithm. This implementation takes as input, besides the algorithm parameters, the distances of all the different signals. After executing the original application, we verified that most of the algorithm's execution time is consumed on the distance calculation. In particular, the calculation of all the distances takes 165 sec which is 99.9% of the overall execution time. For this reason, we focused on the acceleration of the dynamic time warping algorithm which is used for measuring the distance of the available signals.

Dynamic Time Warping

Dynamic time warping (DTW) is an algorithm for measuring similarity between two temporal sequences, which may vary in speed. For instance, similarities in walking could be detected using DTW, even if one person was walking faster than the other, or if there were accelerations and decelerations during the course of an observation. DTW has been applied to temporal

sequences of video, audio, and graphics data. A well-known application has been automatic speech recognition, to cope with different speaking speeds. Other applications include speaker recognition and online signature recognition.

In general, DTW is a method that calculates an optimal match between two given sequences (e.g., time series) with certain restriction and rules:

- Every index from the first sequence must be matched with one or more indices from the other sequence, and vice versa.
- The first index from the first sequence must be matched with the first index from the other sequence (but it does not have to be its only match).

- The last index from the first sequence must be matched with the last index from the other sequence (but it does not have to be its only match).
- The mapping of the indices from the first sequence to indices from the other sequence must be monotonically increasing, and vice versa, i.e., if j >i are indices from the first sequence, then there must not be two indices l>k in the other sequence, such that index i is matched with index l and index j is matched with index k, and vice versa.

The optimal match is denoted by the match that satisfies all the restrictions and the rules and that has the minimal cost, where the cost is computed as the sum of absolute differences, for each matched pair of indices, between their values.

To identify the computationally intensive parts of the DTW algorithm, which are going to be implemented on the FPGA device, the aforementioned methodology was followed. Figure 48 demonstrates the roofline diagram created using Intel Advisor.

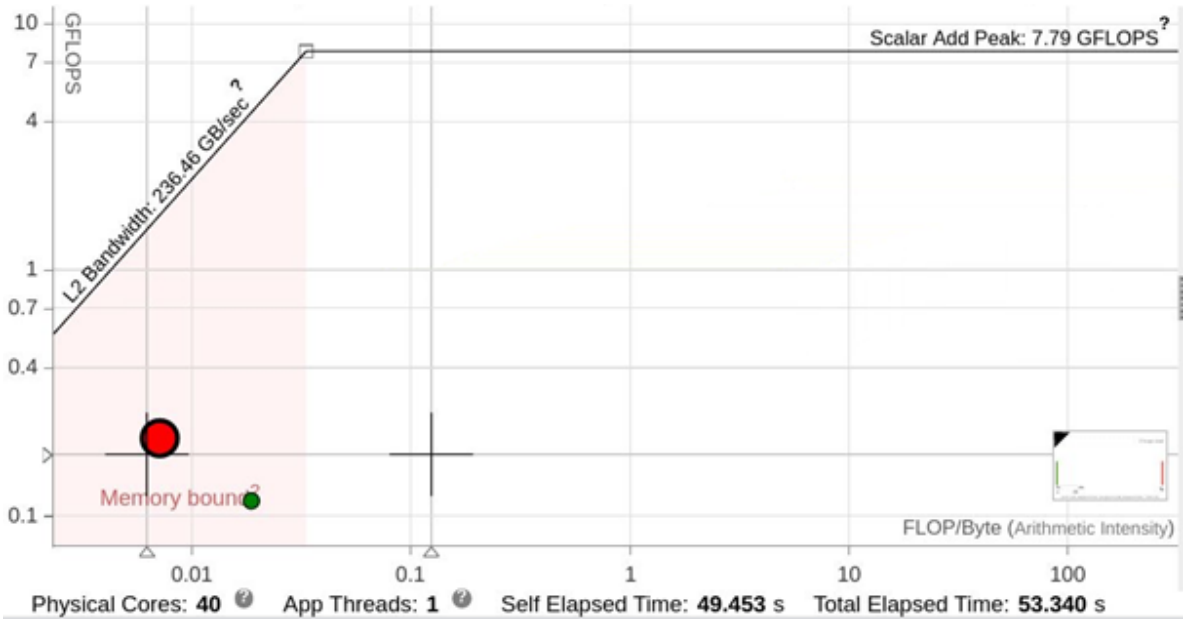

Figure 48 Roofline model for DTW

The roofline model shows the arithmetic intensity for the DTW computation loop which is around 0.01 FLOPs/Byte. The DTW algorithm is a memory-bound algorithm, as depicted from the red circle in the plot. As a second step, the algorithm was executed on different reference architectures and the execution time of each of the algorithm's main operations was measured. The results of the following figures were obtained using all the available input signals provided by IDEKO.

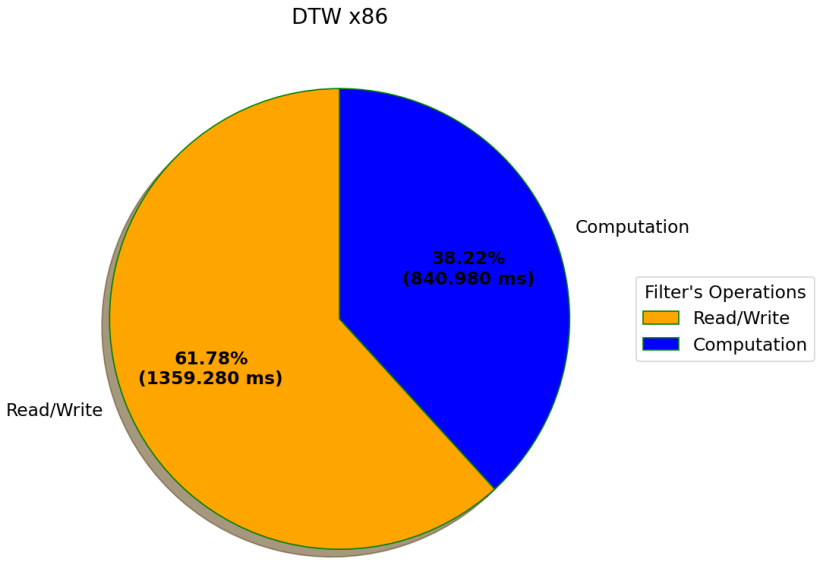

Figure 49 DTW x86 execution time

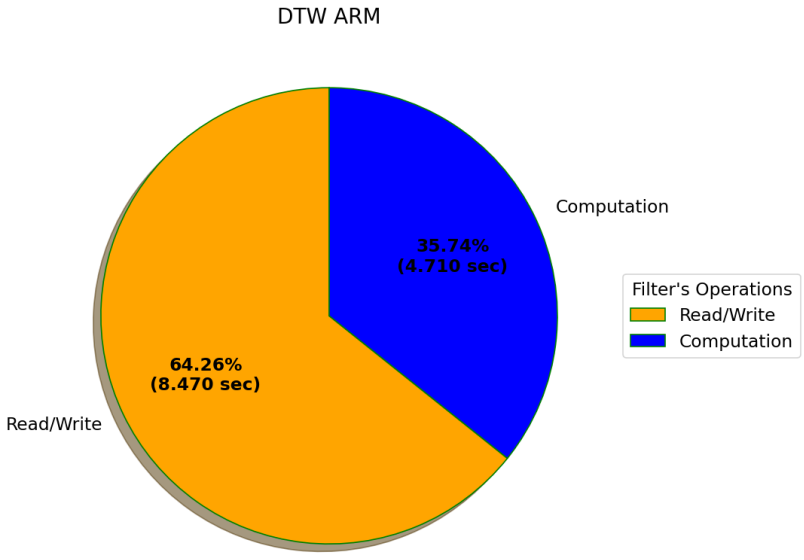

Figure 50 DTW ARM execution time

Figure 49 and Figure 50 show that on both reference architectures the algorithm requires at least 61% of the overall execution time on read/write operations, verifying that DTW is a memory bound algorithm.

5 HW/SW IPs library for acceleration of computationally intensive kernels

Acceleration of AES GCM

AES-GCM consists of 3 main algorithmic parts. The encryption and the decryption computational tasks and the read/write memory intensive task. Even though the memory intensive task takes the same time with both the encryption and decryption computations, due to the accelerators' nature we can only boost the performance of the compute intensive algorithmic parts.

In order to accelerate the encryption and the decryption methods on GPU with CUDA programming model, at first, we convert their serial C written loops to parallel CUDA kernels. We allocate and transfer the AES block array and the key array in the pageable GPU memory through cudaMalloc() and cudaMemcpy() functions and finally we execute the CUDA kernel. We launch both the encryption and decryption CUDA kernels with totally N threads organized in blocks of 1024 threads where N is the AES' block number. Finally, in both kernels we make usage of shared memory in order to achieve more efficient implementations.

We finally compile our CUDA file and we execute the binary:

```
./AES-cuda file_32MB.txt key.txt encrypt.txt decrypt.txt
```

Where file_32MB.txt is the file to be encrypted, the key.txt is the key file, the encrypt.txt is the filename of the encrypted file and the decrypt.txt is the filename of the decrypted file.

5.1 Acceleration of RLNC Erasure Coding

The acceleration of RLNC Erasure Coding involves two parts:

I. Accelerating the computationally intensive part of the **encoder**.
II. Accelerating the computationally intensive part of the **decoder**.

Encoder

The encoding task can be accelerated by parallelizing the calculations that are performed between the encoding coefficients and the input chunks of data.

At the first stage, the input data fragments and the encoding coefficients were transferred through burst memory transactions to the FPGA and stored in local memory units, in order to minimize the latency that is induced by moving data between the global memory and the hardware platform. At the next step, using the corresponding HLS directive 20 identical circuits were instantiated on the platform. Those components operate in parallel to one another and

perform finite field arithmetic operations between the coefficients and the bytes of each chunk. As a result, the latency of the computations is reduced and approaches the latency that would appear in the scenario that each chunk of data had 20 times fewer bytes. In order to perform those computations completely in parallel; the encoding coefficients, the bytes of a single chunk and the finite field multiplication matrices should be placed in different local memories, allowing multiple read and write operations to be performed simultaneously. To achieve that, the array partition HLS directive with a factor of 20 was applied to the input data, splitting the data chunks to multiple local memories and hence allowing the parallel execution of multiple operations. Figure 51 below illustrates the described mechanism.

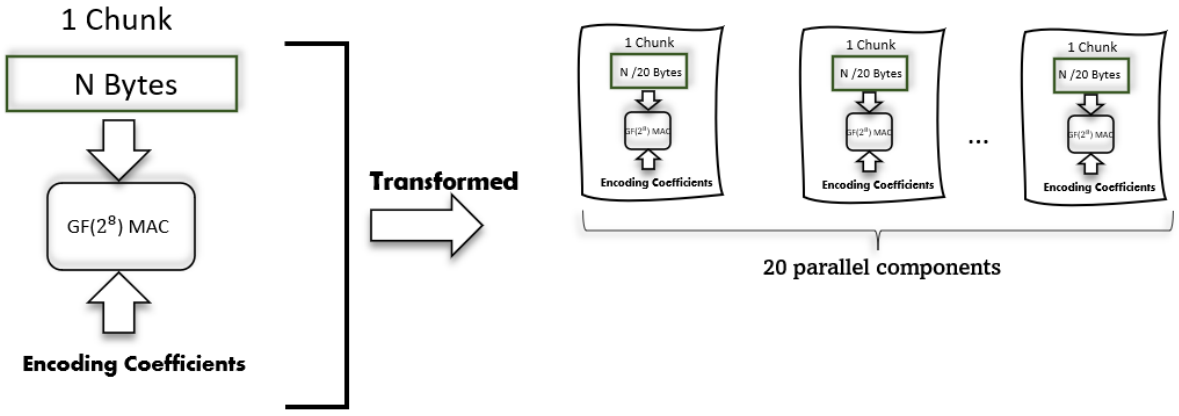

Figure 51 Unrolling the finite field computations by instantiating multiple parallel components

At the next stage, the same process was applied again on a higher level. More specifically, multiple circuits were instantiated in order to perform the above mechanism in parallel for 5 chunks. To achieve that, the corresponding HLS directive was applied, and more parallel components were instantiated on the hardware platform.

Decoder

To accelerate the decoding algorithm, computations that are involved in the gaussian elimination were parallelized. Firstly, a similar approach to the encoding task was followed and the encoded packets and coefficients were transferred to the FPGA's local memory. After that, using the HLS unroll directive 20 components that would perform finite field arithmetic operations and would eliminate the encoded fragments of the bottom and upper triangle were instantiated on the FPGA. The purpose of this technique was to execute this task in parallel, achieving a latency similar to the latency that would appear in the gaussian elimination if each encoded fragment of the coded packets had smaller size.

Besides the aforementioned optimizations that target to implement an algorithm that exhibits task level parallelism, the optimization of instantiating multiple compute units and connecting them to different memory banks was applied to both the encoding and the decoding algorithms. Each compute units implements the 20 parallel components that were described

in the paragraph above and parallelize the encoding and decoding tasks. This optimization leads to a coarse-grained parallel execution by encoding or decoding simultaneously multiple chunks of data. Figure 52 below illustrates the described technique.

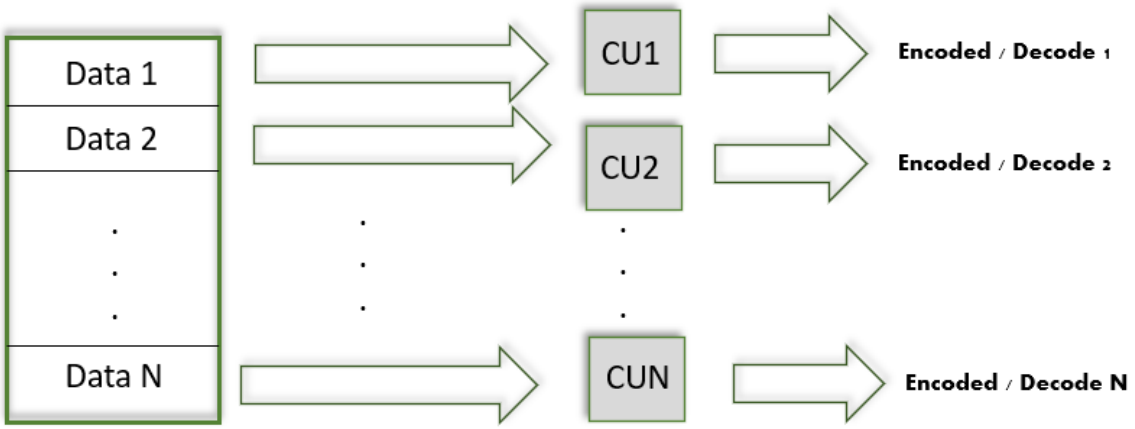

Figure 52 Using multiple compute units for parallel encoding - decoding

5.2 Acceleration of Savitzky-Golay Filter

Based on the profiling analysis of the filter, the discrete convolution algorithm has a variable execution time that depends on the size of the input signal while the computations for generating the filter's coefficients are fixed for a given type of filter. Hence, the acceleration optimizations were performed on two axes (algorithmic and memory level) for a bespoke filter implementation.

The fintech use case scenario uses a third order polynomial curve and a window size of 11 for fitting the adjacent data points and performing the Savitzky-Golay (SAVGOL) filtering. To eliminate the need of calculating the filter's coefficients every time that the smoothing algorithm is invoked two tasks were performed. At first, the coefficients were calculated offline for a third order polynomial SAVGOL filter and subsequently those values were mapped as constants on a Block Memory (BRAM) unit on the FPGA. Table 3 below shows the values of the filter's coefficients.

Coefficients (window = 11, polynomial order = 3)					
1	2	3	4	5	6
0.20745	0.19580	0.16083	0.10256	0.02097	0.08391

Table 3 Filter's coefficients

Algorithmic optimizations

The applied algorithmic optimizations target to minimize the convolution's execution time for minimal utilized resources and hence energy expenses.

A dataflow mechanism that is composed of three distinct sub-units (**read, convolve, write**) was designed for that purpose, performing the memory read, write operations and the convolution computations in a pipelined manner. At first, a memory read operation transfers the first *window* data points from the global memory to the FPGA's local BRAM resources. By reading the first *window* input values and storing them in local memory, we allow the module that performs the convolution operations to begin its calculations without waiting for the whole input dataset to become available first. Once the memory read transactions are concluded the dataflow mechanism is invoked. This mechanism is illustrated in Figure 53 and the operation of its components is described below:

- **read:** The purpose of this module is to perform memory read transactions through the interface port and store the input's signal values in a First-In-First-Out (FIFO) stream. The FIFO stream has a depth of two and buffers the input elements, allowing the **convolve** module to perform operations continuously without stalling its operation for reading data from the global memory.

- **convolve**: This sub-component performs the convolution task and begins its operation once the first *window* input values have been stored in an array in the local memory. The convolution is performed in pipelined loops with each loop executing a number of *window* multiplications between the input signal and the filter's coefficients in parallel. Moreover, a shifting mechanism is used to access the next available input data. Specifically, when all the *window* input values that are stored locally have been convolved, then the elements of this array are shifted, and the first element of the FIFO stream is moved in the last address of the local array. This process is repeated until all the input data elements have been moved in the local memory array and have been convolved. The output of each convolution is stored in a FIFO stream that is connected to the **write** module.

- **write**: This module performs the memory write transactions by writing the output data elements that are stored in the FIFO stream, provided by the **convolve** unit, to the global memory.

  In order to perform the convolutions' multiplications in parallel, complete array partition was performed, using the corresponding array_partition HLS pragma on the BRAM arrays that contain the filter's coefficients and the ***window*** number input data elements.

Memory based optimizations

To minimize the number of accesses to the global memory, a custom data type of 512 bits was created to represent the input and output data values, increasing the memory interface bandwidth to 512 bits and hence minimizing the number of memory transactions.

In order to parallelize the filter's execution for many financial assets, multiple identical compute units were deployed on the FPGA and connected to different memory banks. Each compute unit is responsible to perform the smoothing algorithm through the described dataflow mechanism for the input signal that is currently available and is provided by the host application. It is noted that the host application is agnostic regarding the compute unit that will be assigned to each input signal. This optimization methodology that is depicted in Fig. 54, leads to a coarse-grained parallelism, with each unit performing a single filtering process.

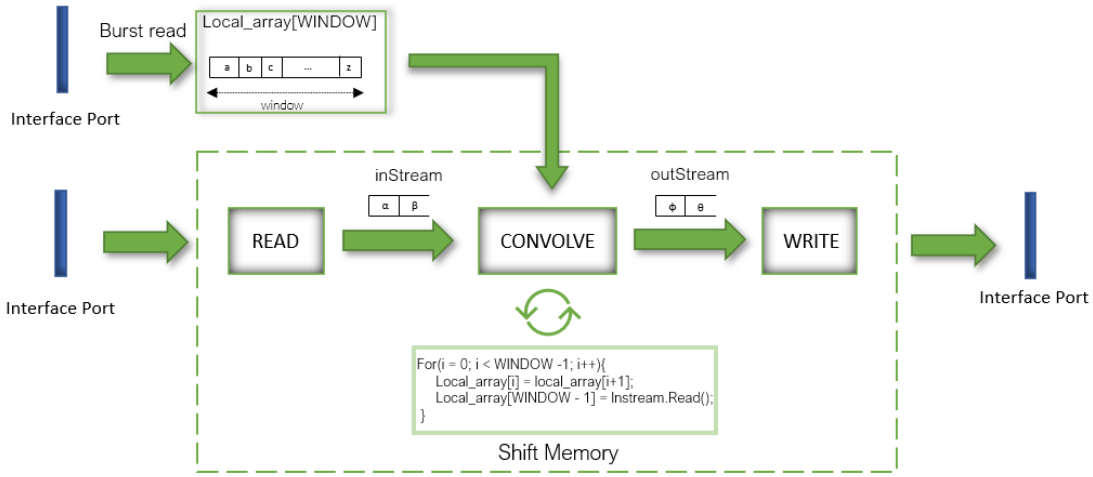

Figure 53 Designed acceleration mechanism

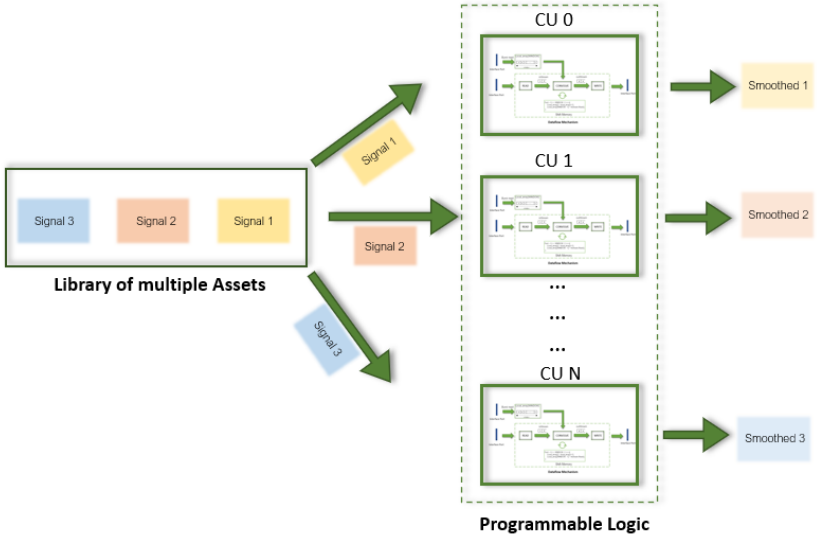

Figure 54 Execution of multiple signals using many compute units

5.3 Acceleration of Kalman Filter

The Kalman filter is an inherently sequential algorithm due to the dependency of each estimated value from the previous one. To accelerate a sequential application the original code should be restructured in a way that parts of the execution flow can be parallelized. In the Kalman filter's case, we split the initial signal into multiple batches and perform filtering in parallel to each of these sub-signals. Although this parallelization strategy breaks the dependency, it is evident that it approximates the original algorithm's output.

As a first step we need to identify the way this parallelization strategy affects the algorithm's output. The batch size is the most important parameter in our approach as the more batches we use the higher parallelization we can achieve. To further investigate this parameter, we perform batched Kalman filtering for different batch sizes and measure the score i.e., the number of output values that are equal to the original. As the input signal consists of floating-point values, we consider that two numbers are equal when their absolute difference is below 0.001. To better highlight the batch size effect, we used the largest available input signal which is composed of around 100 assets. The results of the experiment are depicted in the following figure.

Figure 55 shows that as the batch size increases, and hence less batches are used, the algorithm's output becomes more like the original. For a batch size equal to 32 i.e., 65536 batches, the batched Kalman filter's score is 18.8% while for a batch size equal to 4096 i.e., 512 batches, the score is equal to 99.3%. It is also observed that when the batch size is bigger than 512, the algorithm's score is above 94.3% meaning that the algorithm's parallelism can be significantly increased without losing accuracy. It is possible to further increase the parallelism in batched Kalman filtering and hence use even smaller batch sizes. Figure 56 depicts the mean error percentage for the same batch sizes. As shown, even for a batch size equal to 32 where the score is below 20%, the mean error percentage is equal to 0.64%, meaning that the approximate Kalman filter's output does not significantly differentiate from the original output.

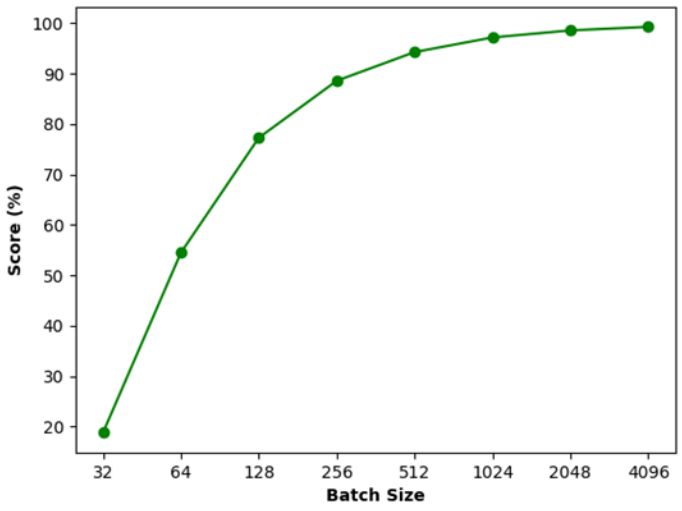

Figure 55 Batched Kalman filter's score for different batch sizes

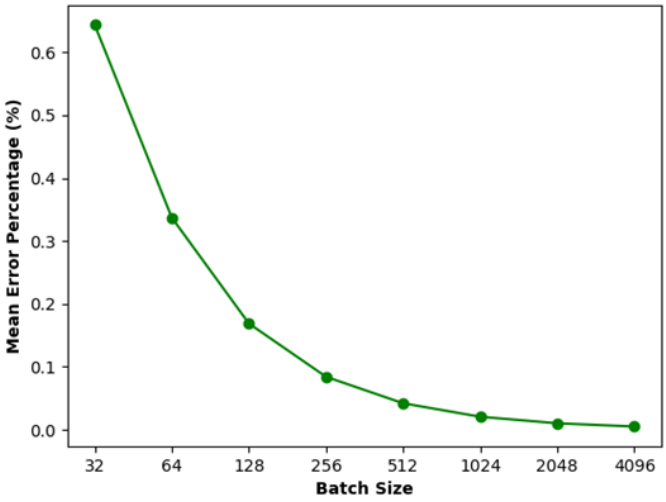

Figure 56 Batched Kalman filter's MEP for different batch sizes

The parallelization opportunities that arise using batching, allow an easy mapping to a FPGA. Our approach was to create as many as possible identical compute units (CU) on the device, split the original signal into equal parts and assign them to a different CU. In addition, each CU performs batching, referred as internal batching, to the sub-signal it is assigned. Table 4, shows the compute units (CU) that were used as well as the number of internal batches for each of the available devices.

Configuration \ Device	**Alveo U50**	**MPSoC ZCU102**
Compute Units	4	1
Internal Batches	32	128

Table 4 Batched Kalman filter's configuration per device

To calculate in parallel the batches of a CU, an array with size equal to the internal batch size is created for all the Kalman loop variables. The loop that iterates over the batches was fully unrolled using the corresponding HLS directive. To fully unroll the loop, it is essential to have parallel memory accesses to each of the arrays. This property can be implemented in HLS by partitioning them completely. Finally, custom data types were used to create an efficient design with the minimal resources.

As the optimization process is device specific, different techniques were used for each FPGA. For the Alveo U50 the available HBM resources were used to minimise the data transfer latency between the host and the device. The full AXI width was also utilised, for both Alveo U50 and MPSoC ZCU102, to decrease the data transfers between the global FPGA memory and the kernel. These optimizations are essential especially for memory bound applications where the data transfers require most of the execution time.

5.4 Acceleration of Wavelet Transformation

In the case of wavelet application, we accelerate on an NVIDIA GPU the dwt_sym_stride() function, the wavelet's most time consuming algorithmic part. As in the case of the AES workload, we launch our kernel with N total threads, where N is the length of the input array that refερσ to the input signal to be processed. Therefore, each thread corresponds to an array element that transforms its corresponding neighboring pieces of the input signal in parallel. Such as in the case of AES-GCM, wavelet's input and output arrays are allocated on the pageable GPU memory through cudaMalloc() function and are transferred from host to device memory and vice versa through cudaMemcpy() function.

5.5 Acceleration of DBSCAN

In section 4, the *density-based spatial clustering of applications with noise* algorithm was analysed and the computationally intensive parts were identified. The time-consuming task of DBSCAN is the *dynamic time warping* distance calculation for all the available input signals provided by IDEKO. For this reason, our acceleration attempts target the DTW calculations that occur before the actual DBSCAN execution. Our parallelization strategy relies on the observation that the calculations of the DTW distances are independent and hence can be executed in parallel. Our approach was to create as many as possible identical compute units (CU) on the device and assign different signal calculations.

To perform the DTW distance calculation in a CU, changes have been made to the original source code. DTW is a dynamic programming (DP) algorithm that calculates the optimal signal alignment and hence the distance, in a bottom-up manner storing the intermediate results in a two-dimensional array. Suppose that the input signal lengths are N and M respectively. This approach consumes *O (N * M)* memory, meaning that for two input signals with ~5000 floating point elements each, the algorithm will create an array with ~25 million floating point elements. Due to its extensive length the DP algorithm could not be mapped to the FPGA as there were not enough BRAMs in the programmable logic (PL). The idea behind the code restructuring was that if the optimal signal alignment is not required (which is computed using backtracking), the DTW distance can be calculated in the exact same way by only storing *O (N + M)* floating point elements. This modification allows an easy mapping to a FPGA.

All of the kernel loops were pipelined using the corresponding HLS directive. The target initiation interval (II) was set to one (II=1). Nevertheless, it was not possible to achieve this goal to all the available loops as the calculation of the element of each iteration depends on the calculation of former elements. Finally, custom data types were used to further decrease the available resources and create a more efficient design. As it was previously mentioned, the optimization process is device specific meaning that different techniques were used for each FPGA. For instance, in the case of the Alveo U50 the available HBM resources were used to minimise the data transfer latency between the host and the device. Finally, different number of compute units (CU) were created to each of the available devices. In particular, for the Alveo U50, 8 CUs were created while for the MPSoC ZCU102 only two.

6 Evaluation

6.1 Evaluation of AES GCM

Our accelerated encryption application was deployed and executed on both cloud and edge NVIDIA GPUs of the SERRANO's infrastructure. We evaluated both encryption and decryption tasks with a 32 MB custom input text file to encrypt and we present the experimental results through the figures below. Note that all GPUs latencies include the time needed for the data to be transferred from host to device and vice versa.

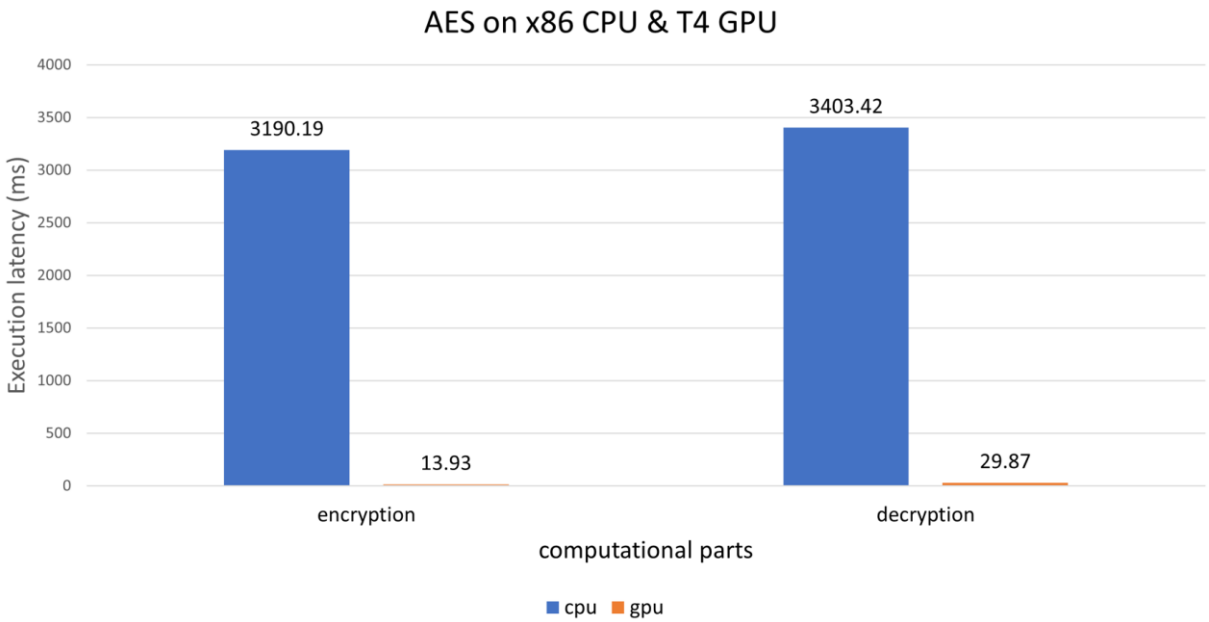

Figure 57 AES-GCM acceleration on T4 GPU

As depicted in Figure 57, our implementation achieves huge speedups for both applications on T4 NVIDIA GPU in comparison with a x86_64 80-core Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz. More specifically we achieve a **x229** and a **x113.14** speedup for encryption and decryption tasks respectively.

Additionally, we evaluate our implementation on Jetson AGX Xavier Developper Kit. AGX Xavier occupies a 6-core A57 ARM @ 2GHz and a 256-core Pascal GPU @ 1.3GHz. Figure 58 depicts the comparison between the CPU and our GPU implementation.

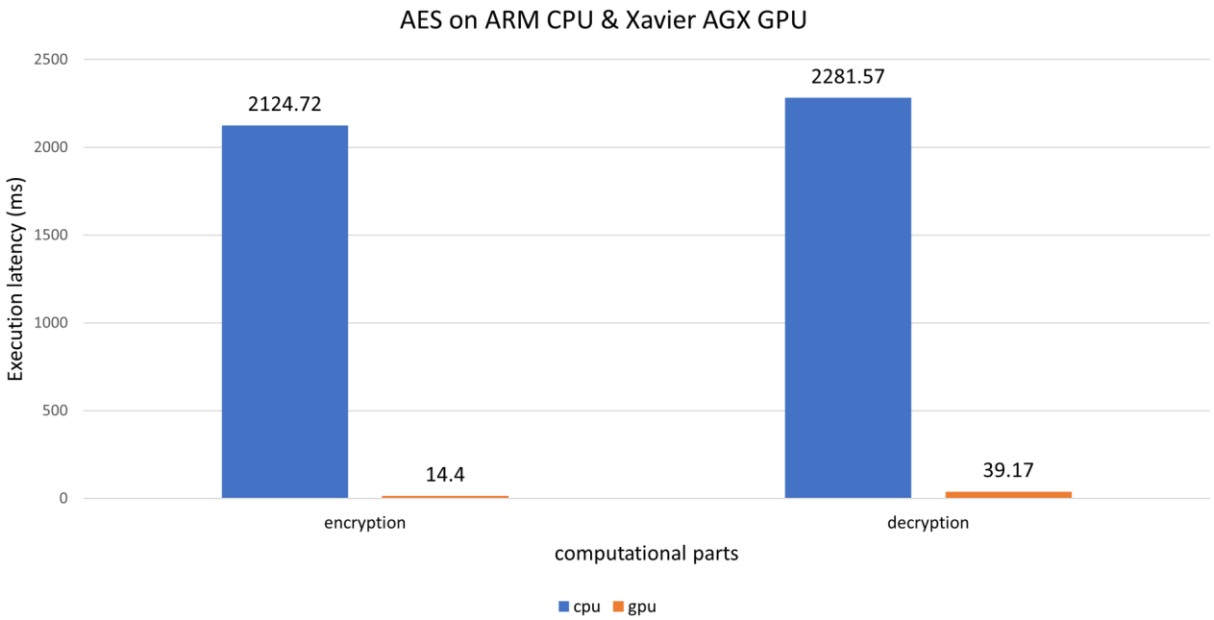

Figure 58 AES-GCM acceleration on Xavier AGX GPU

In this case, we achieve a **x147.55** and a **x58.24** speedup for encryption and decryption tasks respectively. In addition to execution latency, we measured the power that our accelerated applications consumed on both GPUs. The power consumed for both edge Xavier AGX and cloud T4 GPU is **3.9 Watt** and **26 Watt** accordingly disclosing the low power nature of the embedded Xavier AGX GPU in comparison with the cloud T4 GPU.

6.2 Evaluation of RLNC Erasure Coding

The proposed encoding and decoding algorithms were implemented on cloud and edge FPGAs of SERRANO Alveo U50 and MPSoC ZU102. Experiments conducted on encoding and decoding four chunks of data, each chunk of 100KB and their results are presented in the section below. A clock frequency of 300 MHz was set as target operating frequency on both devices. It should be noted that all experiments were executed 10 times and the average execution times and speedups are listed. For the implementation on the Alveo U50 four compute units were instantiated on the platform and connected to different HBM memory banks. Table 5 below shows the overall resource utilization and power consumption for both devices.

Table 5 RLNC erasure coding Encoder

Resource\Device	**Alveo U50 (4 CUs)**	**MPSoC ZCU102 (1 CU)**
BRAM (%)	52 %	40 %
DSP (%)	1 %	~ 0 %
FF (%)	~ 0 %	1 %
LUT (%)	2 %	3 %
Power (Watt)	5.96	2.3

Table 6 RLNC Erasure coding Decoder

Resource\Device	Alveo U50 (4 CUs)	MPSoC ZCU102 (1 CU)
BRAM (%)	44 %	36 %
DSP (%)	1 %	1%
FF (%)	~ 0 %	~ 0 %
LUT (%)	2 %	3 %
Power (Watt)	6	2.1

The pie charts below (Figure 59 and Figure 60) depict the total execution time of the encoding and decoding algorithms on the Alveo U50 FPGA. In the case of the encoding task the time that is required for the kernel to be executed is 1.55 ms while the tasks of transferring data from and to the global memory have an overall latency of 0.5 ms. Similar are the results for the decoding task with the kernel's execution being responsible for 84% of the overall execution time.

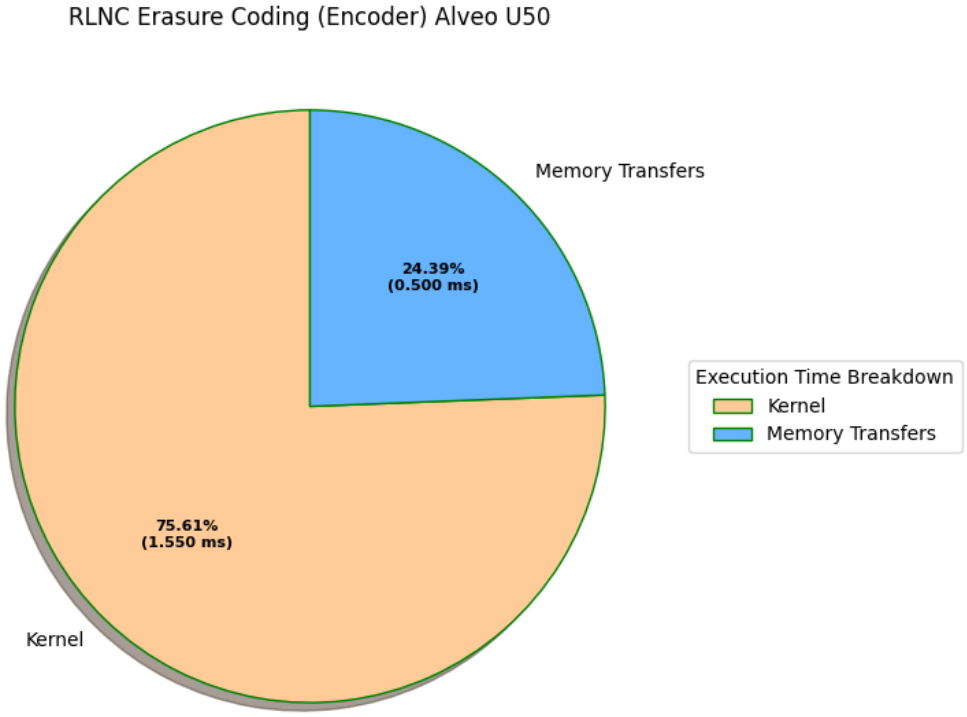

Figure 59 RLNC encoding U50 execution time

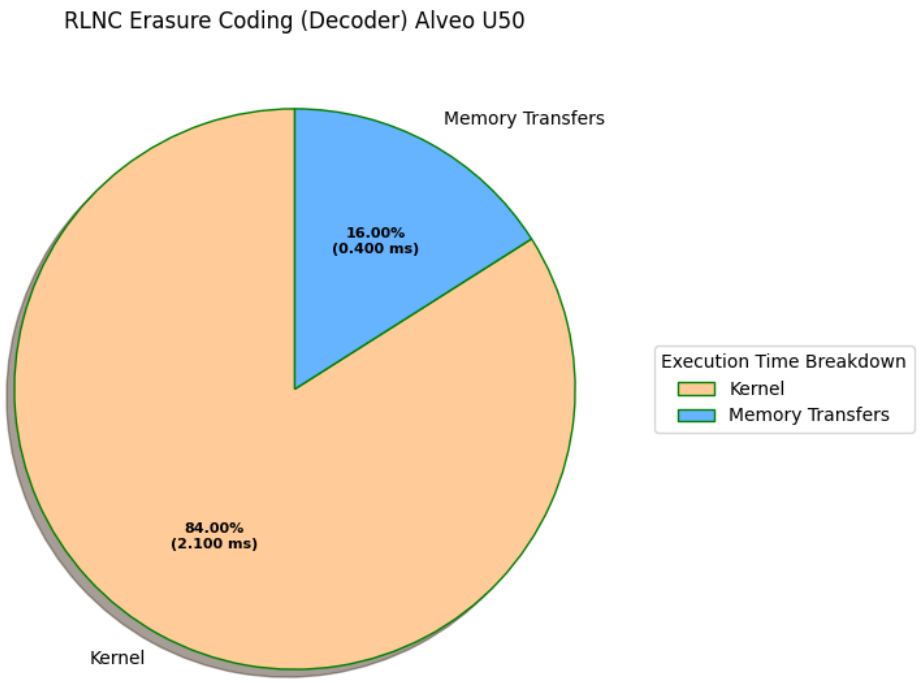

Figure 60 RLNC decoding U50 execution time

Figure 61 and Figure 62 depict the same analysis for the ZCU102 edge device. In this experiment the encoding algorithm is executed in 5.06 ms while the decoding part is required 11.63 ms for its execution.

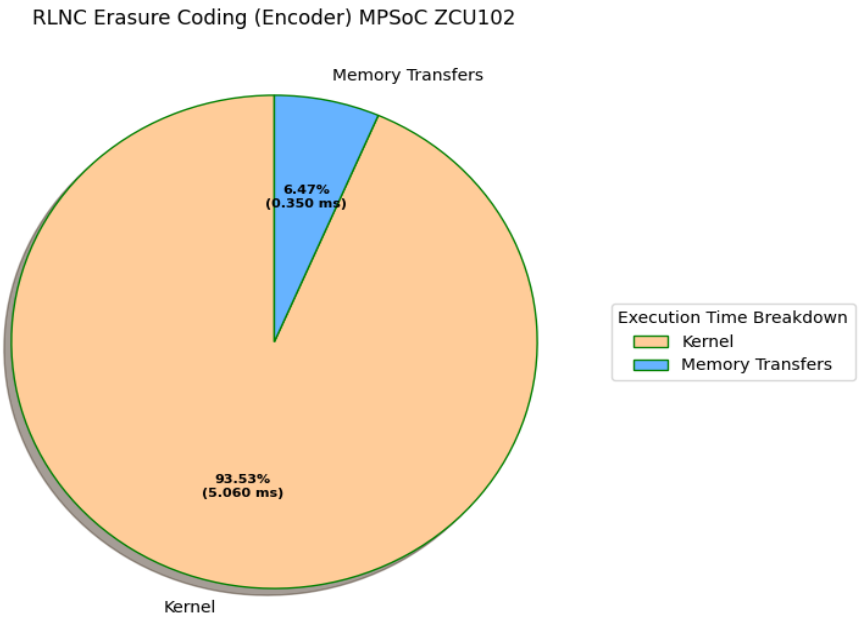

Figure 61 RLNC encoding ZCU102 execution time

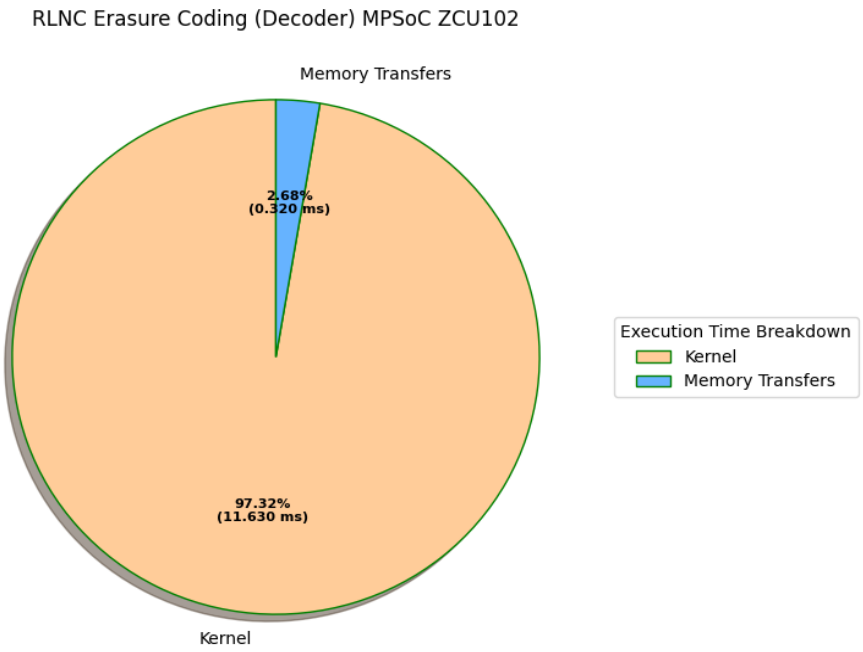

Figure 62 RLNC decoding ZCU102 execution time

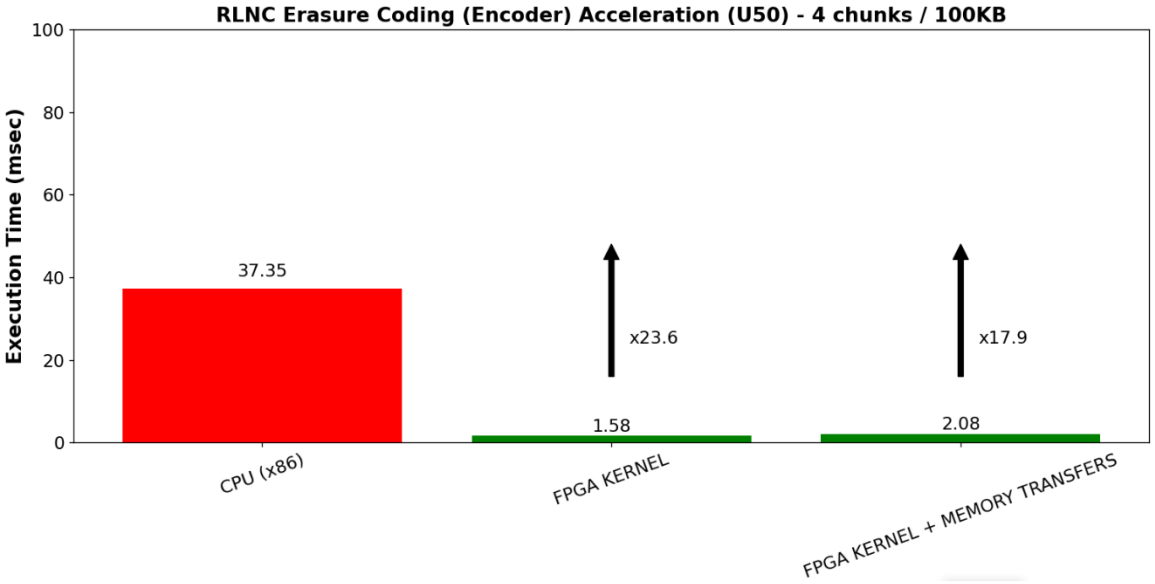

Figure 63 RLNC encoder (U50) acceleration

The bar plots depicted in Figure 63 and Figure 64 show the acceleration of the encoder and decoder when those kernels are executed on the Alveo U50 platform and are compared with our baseline x86 architecture.

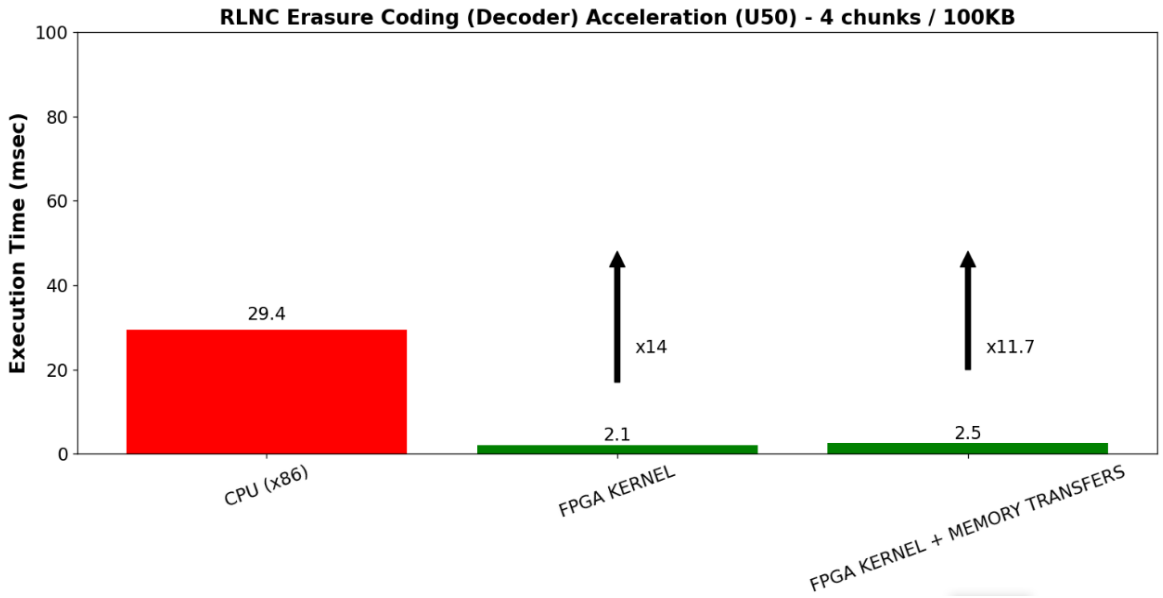

Figure 64 RLNC decoder (U50) acceleration

For the encoding kernel, a speedup of x23.6 is appeared when the kernel's execution is compared to the time that is required for the execution of the same application on the x86 architecture. The total speedup is reduced from x23.6 to x17.9 when the time that is required for moving data from and to the global memory is considered.

Similarly, for the decoding kernel. The decoder's execution time on the U50 FPGA is about 14 times faster than its execution on the x86 processor. Taking into account the latency induced by the memory transfer operations the overall speedup is reduced to x11.7.

The bar plots shown in Figure 65 and Figure 66 depict the acceleration of those two algorithms on the ZCU102 when the algorithmic latency is compared to the execution time on the baseline ARM architecture.

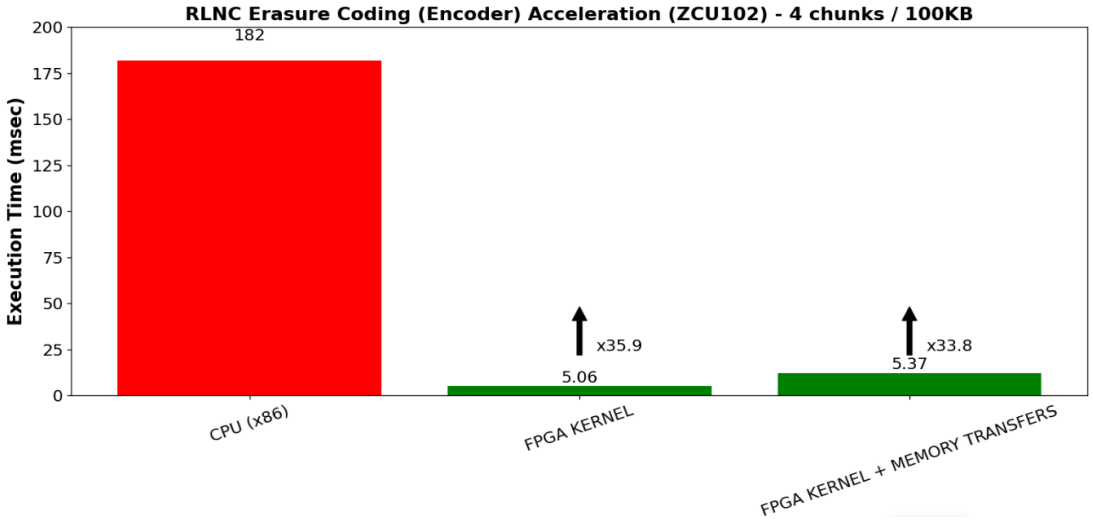

Figure 65 RLNC encoding (ZCU102) acceleration

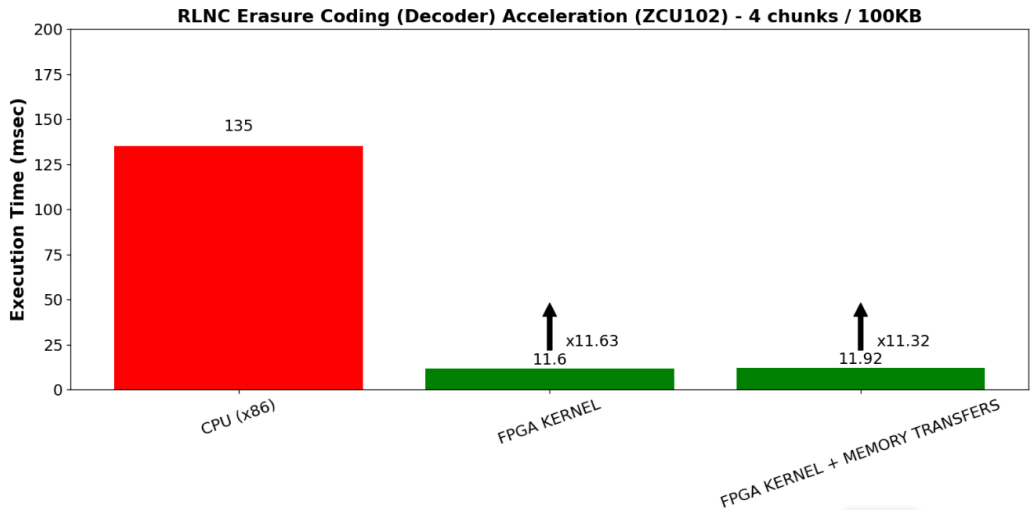

Figure 66 RLNC decoding (ZCU102) acceleration

Based on the above figures the encoder shows a speedup of x35.9. The time that is required for the memory transactions is 0.29 ms. Similarly, the decoding kernel is executed x11.63 faster on the ZCU102 than on the ARM processor.

6.3 Evaluation of Savitzky-Golay Filter

The designed filter was deployed and executed on the cloud and edge FPGAs of the SERRANO's infrastructure. The scenario of performing the SAVGOL filtering for 10 different financial assets -each asset consists of 20000 prices- was tested for all platforms. Table 7 shows the power consumption and resource utilization for all hardware platforms. It is noted that a clock frequency of 300 MHz was set as the operating frequency. For the Alveo U50 acceleration card, 10 identical compute units were instantiated on the programmable logic to achieve coarse-grained parallelism, with each compute unit assigned to a specific HBM memory, while for the edge MPSoC devices a single compute unit is used.

Table 7 Savitzky-Golay filter's consumed resources for all the available designs

Resource\Device	**Alveo U50 (10 CUs)**	**MPSoC ZCU102 (1 CU)**
BRAM (%)	11.6 %	5 %
DSP (%)	9.2 %	1 %
FF (%)	~ 0 %	4 %
LUT (%)	12.3 %	11 %
Power (Watt)	9.1	2.2

The pie charts (Figure 67 and Figure 69) depict the filter's execution time per platform. It is noted that the experiments and the figures presented in the following paragraphs were executed 10 times and the average execution and data transfer times are displayed.

The execution of the application consists of the tasks of transferring data from the host application to the device's global memory and running the filter. In the case of the U50 device the accelerated kernel has a latency of 0.52 ms on average after 10 runs, while the processes of moving data from and to the global memory have a latency of 0.048 ms and 0.083 ms respectively.

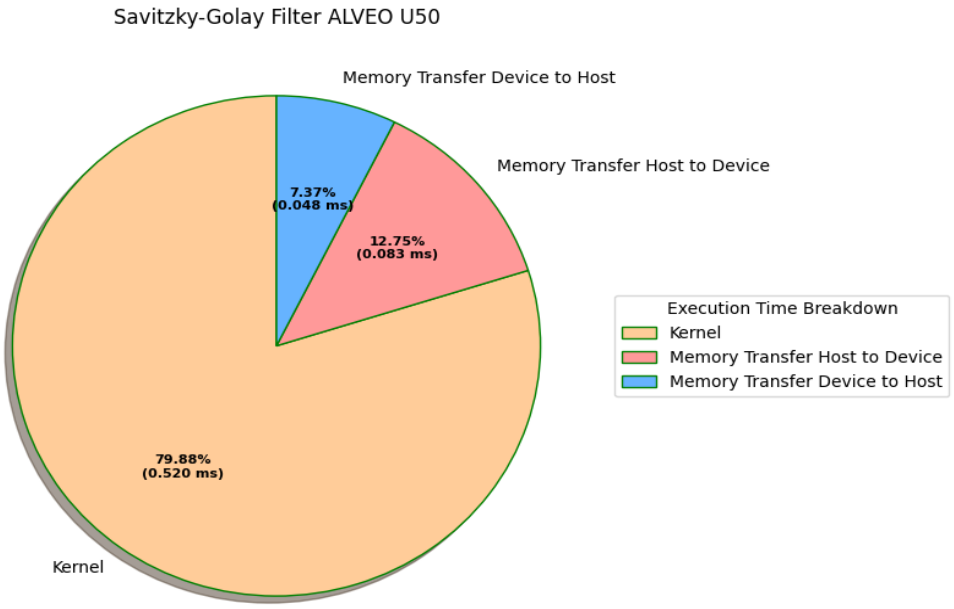

Figure 67 Filter's execution on U50

Figure 68 below shows the application's acceleration for the U50 acceleration card with the overall execution time compared to the x86 baseline architecture.
The designed filter on the FPGA leads to an execution time that is 21.9 times faster than the corresponding execution on a x86 architecture. Considering the time that is needed for moving data from the host side to the hardware platform the total execution time is 17.5 times faster.

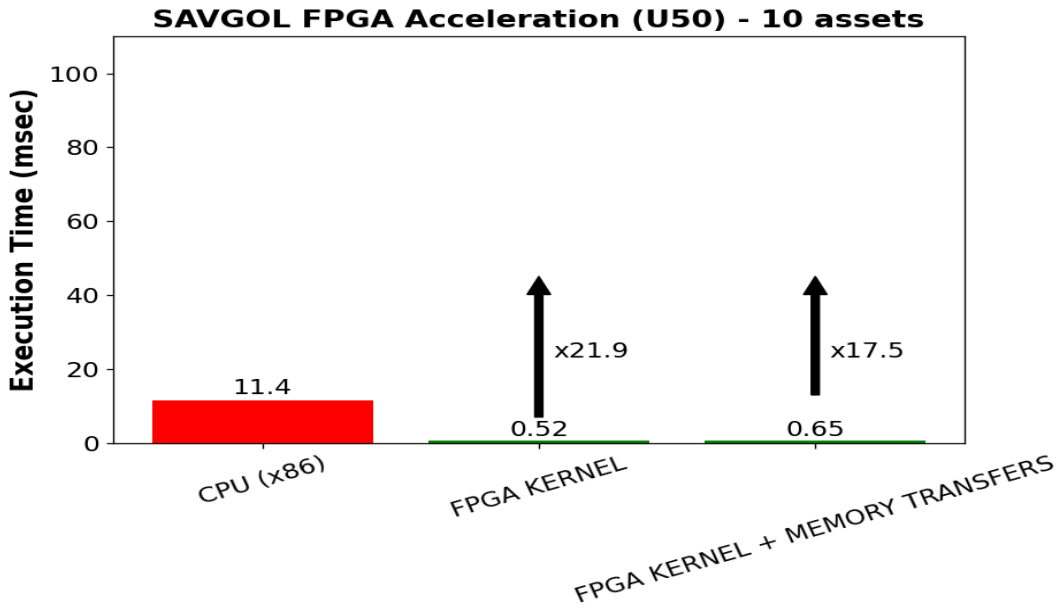

Figure 68 Filter's acceleration on U50

Figure 69 depicts the execution time breakdown for the ZCU102 edge device. In the presented case, the kernel's execution time is 5.8 ms and about 10 times slower that the execution time of the same kernel on the U50 device. For the specific device, the time that is required for transferring data from and to the global memory is 0.48 ms and 0.2 ms correspondingly.

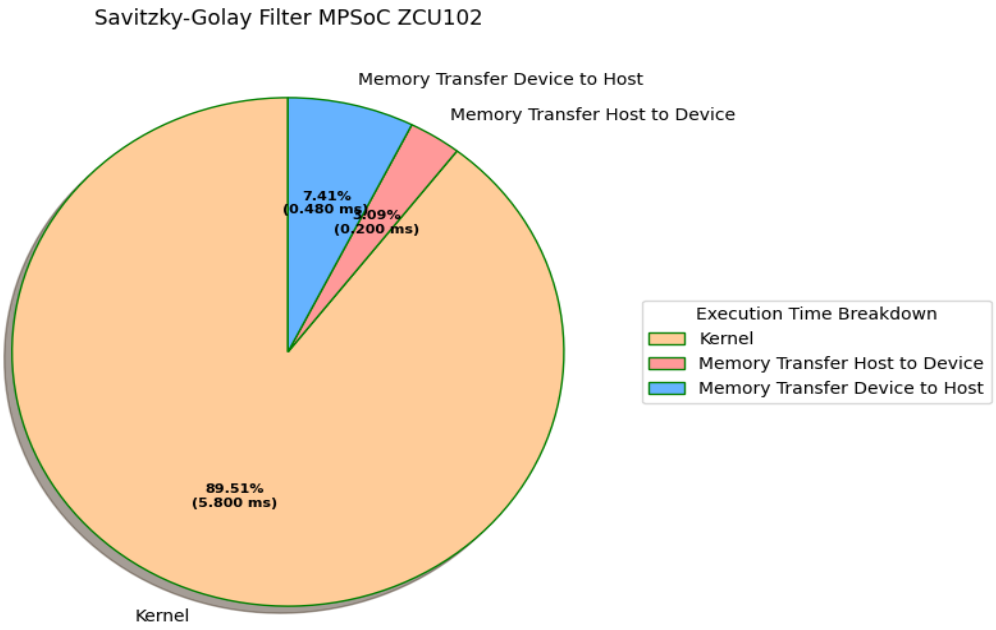

Figure 69 Filter's execution on ZCU102

Figure 70 shows the acceleration results for the ZCU102 device. Specifically, a speedup of 19.9 is noticed when the kernel's execution time is compared with the filter's execution time on an ARM baseline architecture. The total speedup is reduced from x19.9 to x17.8 if we take account the latency that is induced from the memory transfer operations.

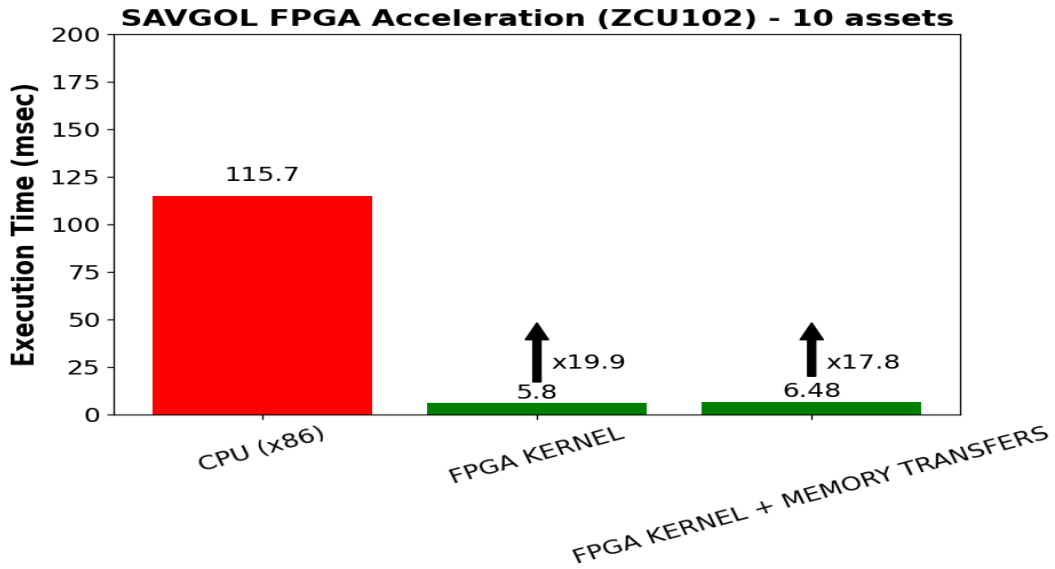

Figure 70 Filter's acceleration on ZCU102

6.4 Evaluation of Kalman Filter

After applying batching to the original application, as well as the device specific optimizations described in section 5, the batched Kalman filter was executed on different FPGA devices available in the SERRANO platform. This evaluation aims to highlight the impact of our acceleration strategy in terms of performance, power, and consumed resources. In addition, by analysing these results we will also be able to identify whether batched Kalman filter is an application that is suitable for acceleration on the edge, cloud, or both.

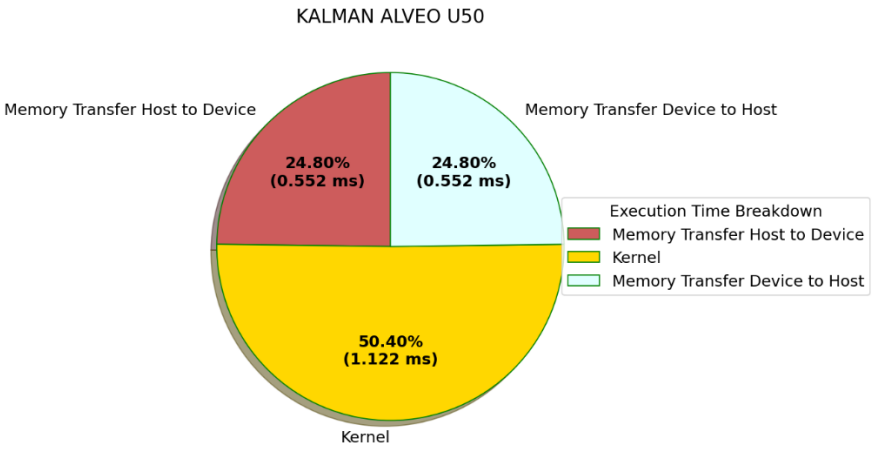

Figure 71 Batched Kalman filter's execution time breakdown on ALVEO U50

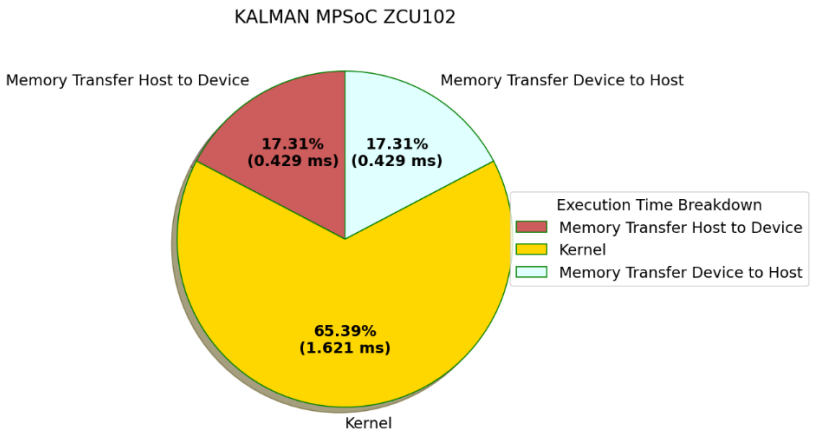

Figure 72 Batched Kalman filter's execution time breakdown on ZCU102

As it was mentioned in section 3, the overall execution time of an application that uses hardware for acceleration, consists of moving the input data from host to device, executing the hardware function and moving the output data from the device to host. To quantify the duration of each of these distinct steps, the batched Kalman filtering was executed for the largest available signal which is composed of around 20 assets for Alveo U50 and 5 assets for MPSoC ZCU102. The input data size is differentiated due to the fewer resources of edge devices. It is noted that a clock frequency of 300 MHz was set as the operating frequency. Figure 71 shows the execution time breakdown of batched Kalman filtering for Alveo U50. As the figure depicts, the kernel requires 50.4% of the overall execution time i.e., 1.122 msec while the memory transfers require 24.8% each i.e., 0.522 msec. Figure 72 depicts the same diagram for MPSoC ZCU102. The results are similar, as 65.4% of the overall execution time is required from the kernel and 34.6% is consumed on memory transfers.

After breaking down the execution time of the accelerated function, the overall execution time is compared with the corresponding CPU baseline. For instance, accelerators targeted for Alveo FPGAs (e.g., U50 and U200) were compared with the x86 baseline while accelerators targeted for MPSoC FPGAs (e.g., ZCU102 and ZCU104) were compared with the ARM baseline. Figure 73 and Figure 74 show the CPU baseline, the kernel and overall FPGA execution times as well as the corresponding speedups for the two examined devices.

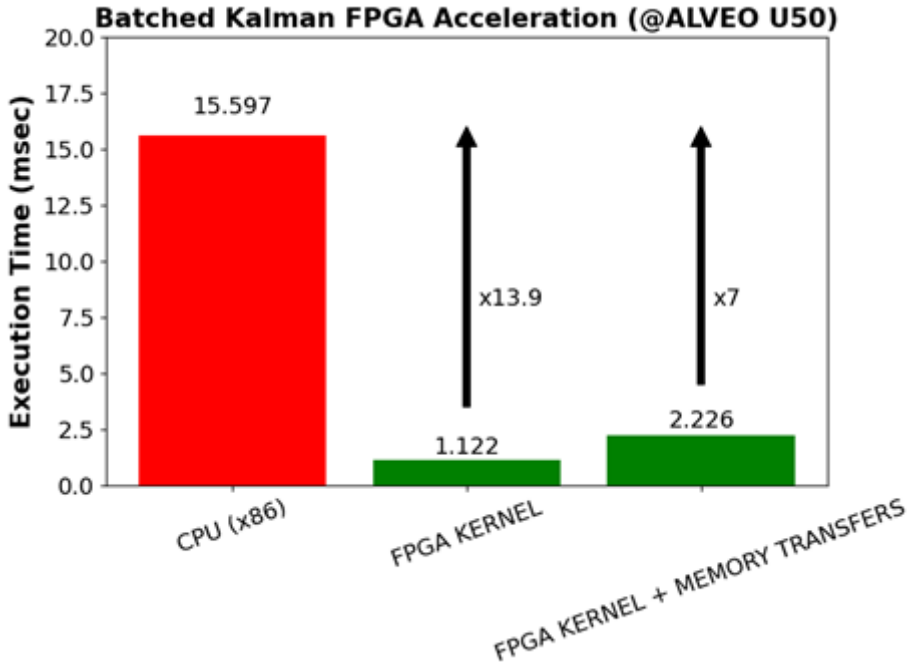

Figure 73 Batched Kalman filter's execution time speedups on ALVEO U50

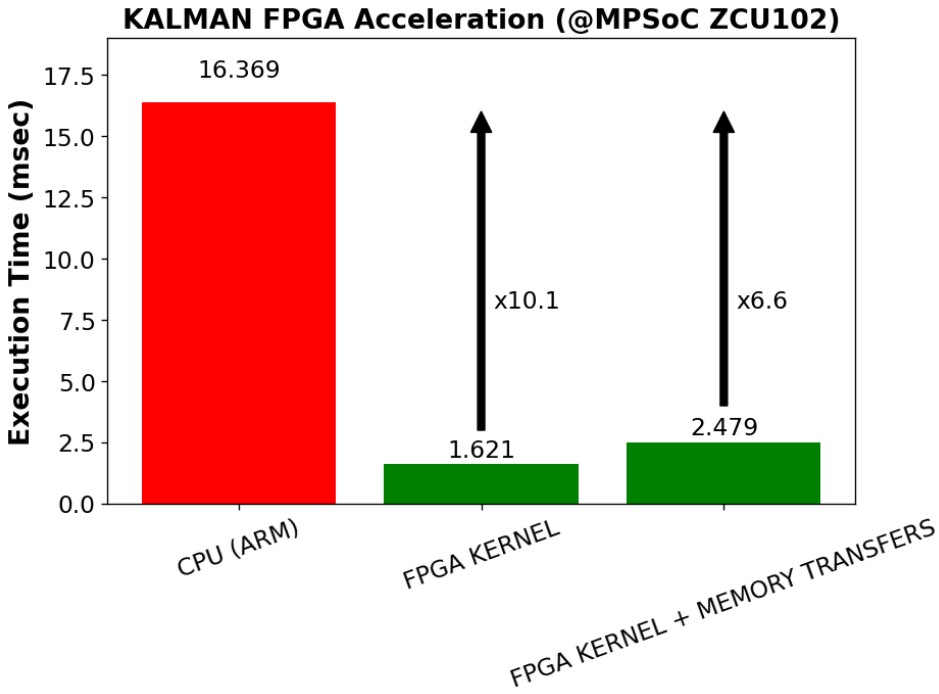

Figure 74 Batched Kalman filter's execution time speedups on MPSoC ZCU102

Our acceleration strategy leads to a x13.9 faster execution without considering the memory transfers and to a x7 overall speedup on an Alveo U50. The design that targets the MPSoC ZCU102 leads to a x10.1 faster execution and to a x6.6 overall speedup.

Another important aspect of the final designs is the device resources and the power they consume during the FPGA execution. Table 8 shows the allocated resources of our designs in all the available devices. For the Alveo FPGAs we present the consumed resources in one Super Logic Region (SLR).

Table 8 Batched Kalman filter's consumed resources for all the available designs

Resource\Device	**Alveo U50**	**MPSoC ZCU102**
BRAM (%)	49.72 %	17 %
DSP (%)	4.32 %	10 %
FF (%)	~ 0 %	43 %
LUT (%)	20.52 %	75 %
Power (Watt)	7.422 W	5.8 W

6.5 Evaluation of Wavelet Transformation

We deployed and evaluated our accelerated version of wavelet filter on both cloud and edge NVIDIA GPUs as well. In Figure 75 and Figure 76, we depict the resulted latencies for a 10 million data points 1D array as an input. Note that both the recorded latencies of CPU and GPU implementations concern the total latencies of the dwt_sym_stride. In our case, this function is called and executed 4 times. Therefore, in the following figures we depict the total sum of the 4 corresponding latencies. Additionally, every latency is presented, constitutes both the kernel execution and the memory copies latencies needed for the computations.

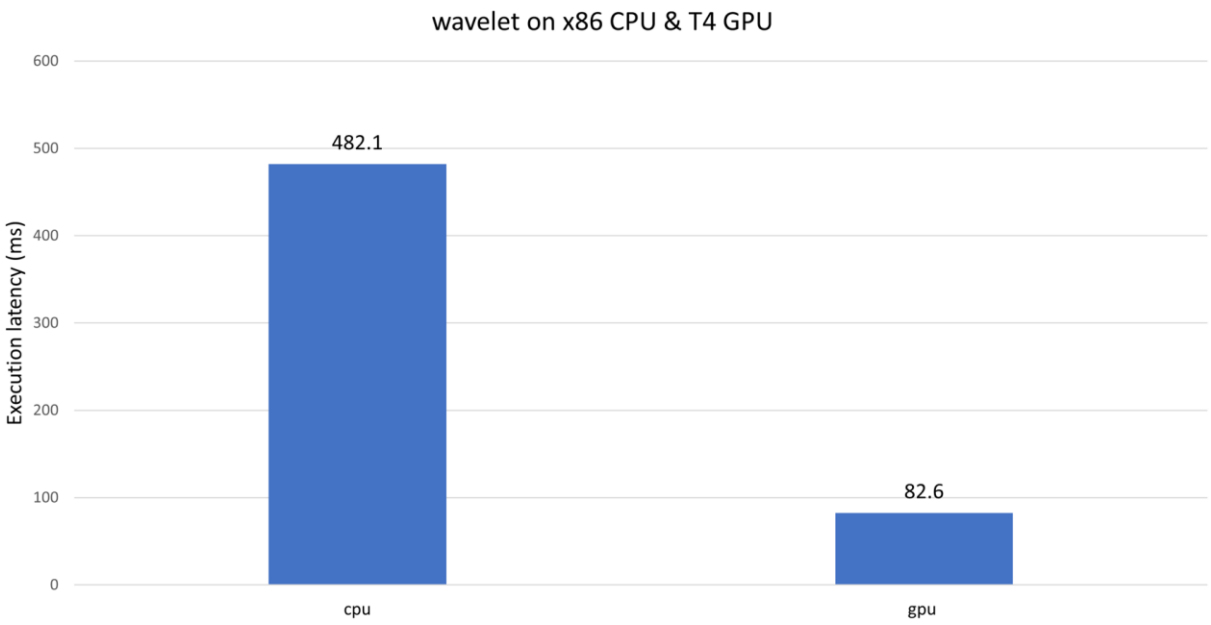

Figure 75 Wavelet acceleration on T4 GPU

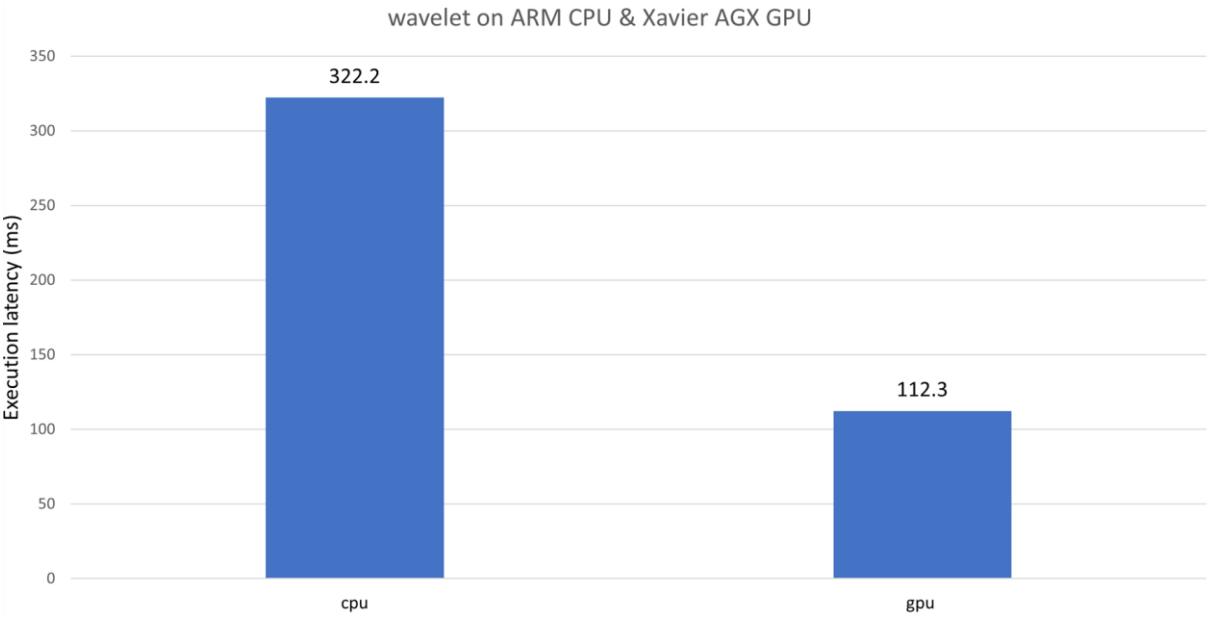

Figure 76 Wavelet acceleration on Xavier AGX GPU

As presented, in the case of wavelet filter we achieve high accelerations with **x5.83** and **x2.86** speedups for the cloud T4 GPU and the edge Xavier AGX GPU accordingly. Additionally, the final CUDA accelerated wavelet consumes **3.11 Watt** and **26 Watt** on the edge Xavier AGX and the cloud T4 GPU respectively. Xavier AGX presents in the wavelet's case a very power efficient GPU in comparison with the cloud T4 GPU.

6.6 Evaluation of DBSCAN

After restructuring the original application, the *dynamic time warping* algorithm was executed on different devices. By evaluating our strategy in terms of performance, power, and resources, opportunities for further acceleration can be identified. In addition, by analysing these results we will also be able to identify whether *dynamic time warping* is an application that is suitable for acceleration on the edge, cloud, or both.

As a first step, the duration of data movement from host to device and device to host, as well as the duration of the hardware function were measured. To quantify the duration of each of these distinct steps, the application was executed for all the input signals provided by IDEKO. It is noted that a clock frequency of 300 MHz was set as the operating frequency.

Figure 77 shows the execution time breakdown of DTW application for Alveo U50. As the figure depicts, the kernel requires 99.45% of the overall execution time i.e., ~253 msec while the memory transfers require 0.55% each i.e., 1.411 msec. Figure 78 shows the same diagram for MPSoC ZCU102. The results are similar, as 99.84% of the overall execution time is required from the kernel and only the 0.16% is consumed on memory transfers.

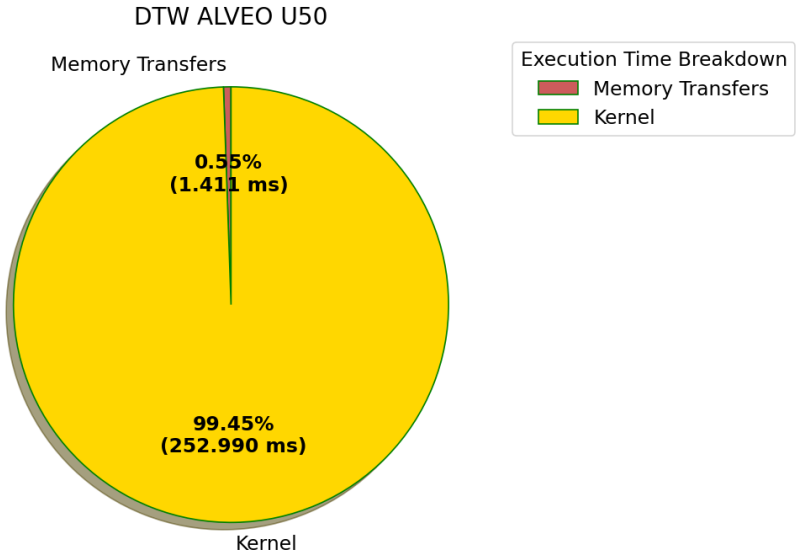

Figure 77 DTW execution time breakdown on ALVEO U50

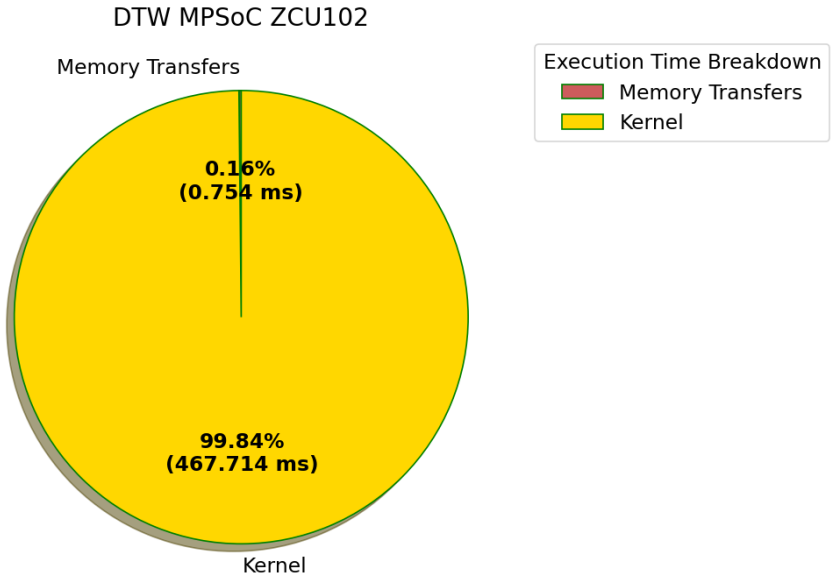

Figure 78 DTW execution time breakdown on ZCU102

After breaking down the execution time of the accelerated function, the overall execution time is compared with the corresponding CPU baseline. Figure 79 shows the CPU baseline, the kernel and overall FPGA execution times as well as the corresponding speedups. Our acceleration strategy leads to a x7.66 faster execution without considering the memory transfers and to a x7.62 overall speedup on an Alveo U50. Similarly, the kernel is executed x8.98 faster on the MPSoC ZCU102 than on the ARM processor. The overall speedup is almost the same (x8.96) as the memory transfers require a very small amount of the overall execution time.

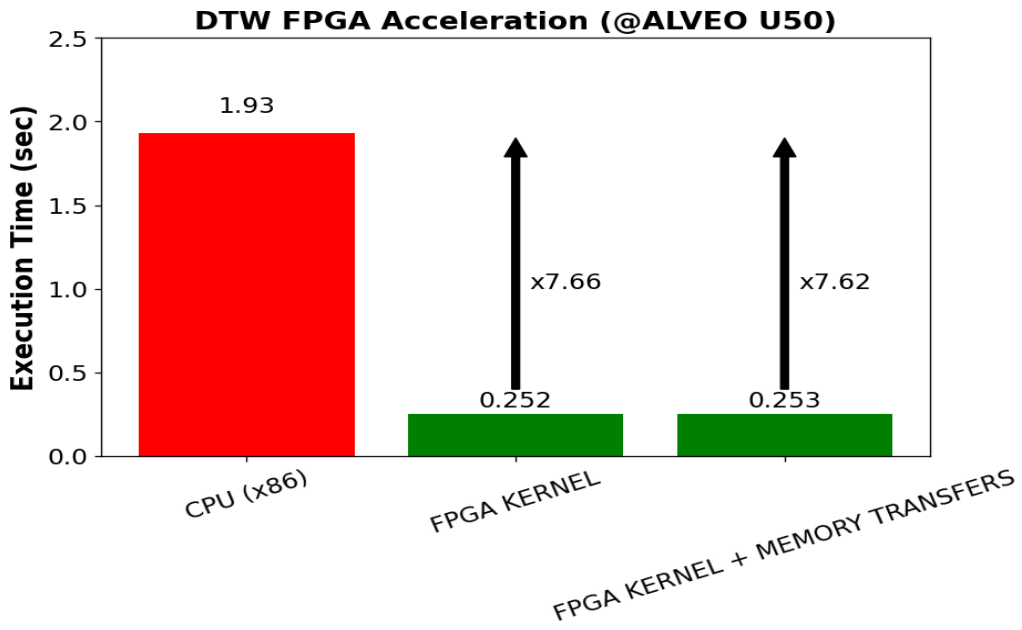

Figure 79 DTW's execution time speedups on ALVEO U50

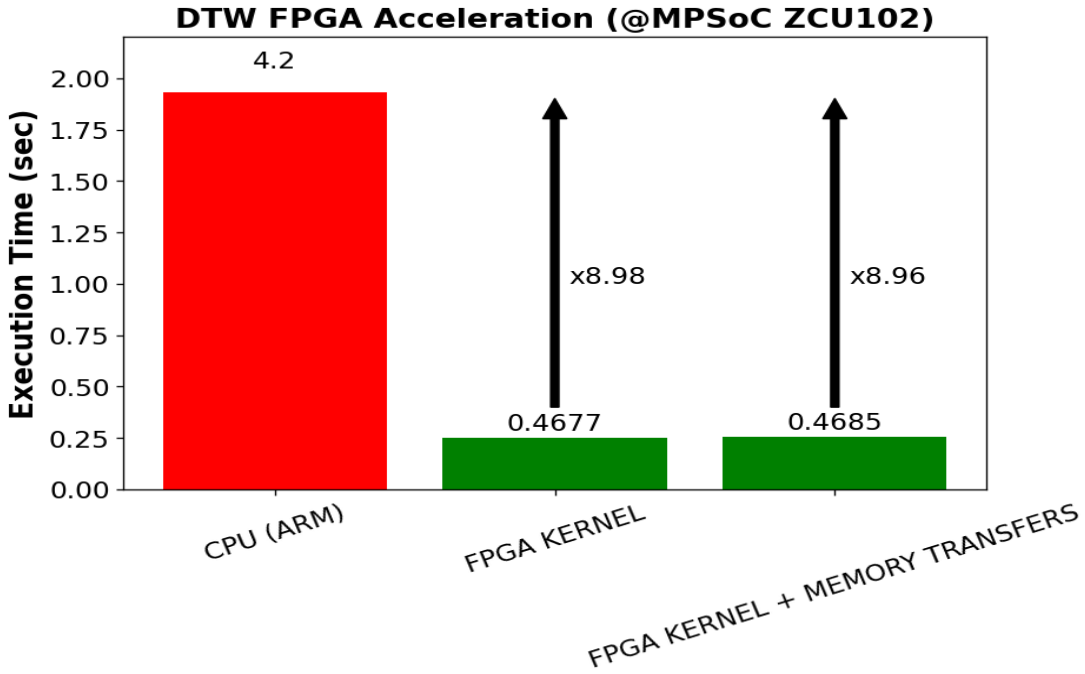

Figure 80 DTW's execution time speedups on MPSoC ZCU102

Finally, Table 9 shows the allocated resources of our designs in all the available devices as well as the power consumption.

Table 9 DTW's consumed resources for all the available devices

Resource\Device	**Alveo U50**	**MPSoC ZCU102**
BRAM (%)	15.44 %	4 %
DSP (%)	~0.0 %	0 %
FF (%)	~0.0 %	2 %
LUT (%)	5.44 %	6 %
Power (Watt)	6.895 W	2 W

7 Conclusion

Nowadays, there is an ever-increased number of applications deployed over Edge, Cloud and HPC infrastructures. This explosion of computing devices across the computing continuum poses new challenges in terms of providing a power-efficient, secure and automatic way for the deployment of different applications in such heterogeneous environments. Moreover, the need for performance efficient deployments within such environments makes imperative the utilization of hardware accelerators throughout the entire computing stack.

In this work, we evaluate different acceleration approaches for Edge and Cloud infrastructures. We describe various programming models that enables us to accelerate the use-cases and we further examine a set of major optimization techniques. We also provide an initial version of the accelerated use-cases focusing on 6 different algorithms required by them. Finally, we apply and evaluate our accelerations on various hardware devices, showing that we achieve up to x229 performance speedup compared to baseline CPU execution.