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

Grant Agreement no. 101017168

Deliverable D4.4
Final Release of the SERRANO Cloud and Edge Acceleration Platforms and Tools

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:	HLRS
Due date:	30/06/2023
Actual submission date:	30/06/2023

Dissemination level:	Public
Revision:	1.0

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

Revision History

Date	Editor	Status	Version	Changes
09.03.23	Javad Fadaie Ghotbi	Draft	0.1	Initial ToC
15.05.23	Javad Fadaie Ghotbi	Draft	0.15	Added task 4.1
20.05.23	Javad Fadaie Ghotbi	Draft	0.18	Added task 4.2
01.06.23	Javad Fadaie Ghotbi, Kamil Tokmakov	Draft	0.2	Added VVUQ framework
15.05.23	Gabriel Iuhasz	Draft	0.3	Added 3.3
01.06.23	Argyrios Kokkinis	Draft	0.4	Added updates regarding tasks 4.1, 4.2 and 4.3
01.06.23	Aggelos Ferikoglou	Draft	0.5	Added task 4.3
11.06.23	Anastassios Nanos	Draft	0.6	Added task 4.4
19.06.23	Kamil Tokmakov	Draft	0.7	Edit Draft
23.06.23	Javier Martin, Yoray Zack	Draft	0.8	Review Draft
28.06.23	Javad Fadaie Ghotbi, Kamil Tokmakov	Draft	0.9	Integrate review comments into draft
28.06.23	Javad Fadaie Ghotbi, Kamil Tokmakov	Final	1.0	Final version

Author List

Organization	Author
USTUTT/HLRS	Javad Fadaie Ghotbi, Kamil Tokmakov
AUTH	Dimitrios Danopoulos, Aggelos Ferikoglou Ioannis Oroutzoglou, Kostas Siozios, Argyris Kokkinis, Dimosthenis Masouros, Stylianos Siskos, Spyridon Nikolaidis, George Zervakis, Zoi Agorastou, Florentia Afentaki, George Margaritis, Christina Panagiotopoulou
NBFC	Anastassios Nanos
UVT	Gabriel Iuhasz

Internal Reviewers

IDEKO
MLNX

Abstract: This deliverable (D4.4) presents the outcomes of WP4. The SERRANO platform successfully addresses the challenges of accelerating kernels in an HPC environment by incorporating the power of HPC systems and parallelization techniques. The platform offers flexibility in performance and energy efficiency tradeoffs by integrating GPU- and FPGA-accelerated versions. Additionally, the platform incorporates transprecision and approximation computing techniques to enhance performance, and optimise the utilisation of compute and memory resources.

To manage the uncertainties introduced into the SERRANO platform, a Verification, Validation, and Uncertainty Quantification (VVUQ) framework has been developed. The VVUQ framework quantifies uncertainties and suggests optimal parameters to improve performance and energy efficiency.

Plug&Chip framework facilitates the development of FPGA and GPU accelerators with automatic optimization. The framework ensures enhanced performance without manual intervention and emphasises memory-efficient designs for FPGA applications.

Furthermore, the vAccel framework, enhanced by NBFC and partners, enables the execution of hardware-accelerated operations as serverless functions in isolated contexts, expanding the platform's capabilities.

Keywords: SERRANO architecture, SERRANO platform, HPC services, FPGA, GPU accelerator, approximation and transprecision computing, VVUQ, vAccel, Plug&Chip

Table of Contents

List of Images

List of Tables

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
DSP	Digital Signal Processing
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
HPO	Hyper-Parameter Optimization
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

The SERRANO platform has successfully developed an HPC service at HLRS that aims to accelerate kernels proposed by use case providers within an HPC environment. This service incorporates the power of HPC systems and parallelization techniques using OpenMP/MPI, enabling efficient processing of large volumes of data and compute-intensive applications with minimal runtime.

AUTH has successfully developed GPU- and FPGA-accelerated versions for all edge and cloud devices available on the SERRANO platform. These accelerated versions, along with the HPC versions, consider various performance and energy efficiency tradeoffs. As a result, they provide the orchestration framework developed in WP5 with multiple degrees of freedom, allowing for flexible optimization options.

In response to the challenges, such as limited computing and memory resources, HLRS has integrated transprecision and approximation computing techniques into the kernel implementation within the HPC service. This integration enables the execution of the kernel with varying data precision and allows for minimised and adaptable computations within the kernel.

AUTH has also applied approximation techniques to FPGA-accelerated application versions, such as precision scaling, approximate memoization, and loop perforation. These techniques contribute to increased energy efficiency and enrich the range of available UC applications.

The utilisation of these techniques introduces parameters and uncertainties. Verification, Validation, and Uncertainty Quantification (VVUQ) is designed to quantify uncertainty and effectively manage the trade-off between accuracy and execution runtime. By suggesting optimal parameters for kernel execution, the VVUQ framework aims to maximise performance and improve energy efficiency.

To estimate execution time and energy consumption for different data batches, the VVUQ framework employs the Gradient Descent method. This method enables the development of a non-linear formula that provides accurate estimations, facilitating efficient resource allocation and planning within the HPC service.

The Plug&Chip framework has been introduced into the SERRANO platform by AUTH, which enables the development of FPGA and GPU accelerators. This framework incorporates automatic optimization techniques, allowing for performance enhancements without requiring manual intervention. AUTH also focuses on developing a methodology for creating memory-efficient accelerators specifically for FPGA applications.

NBFC and the involved partners used and enhanced the vAccel framework to allow an arbitrary hardware-accelerated operation to be executed as a serverless function in an isolated context.

2 Introduction

This deliverable constitutes the final release of the SERRANO platforms, frameworks, and tools developed in WP4 and demonstrates the progress achieved during the project period second iteration of the SERRANO implementation plane(M16-M30). In the scope of WP4, the SERRANO project progressed in different dimensions, such as the development of the accelerated kernels, the application of approximation techniques, the introduction of Plug&Chip, and vAccel frameworks.

During the reporting period, the SERRANO platform enriched a range of accelerators, from energy-efficient devices at the network edge to high-performance, massively parallel devices in the cloud and HPC. Different applications' versions, including HPC, GPU, and FPGA-accelerated versions, have been developed for various devices on the platform, considering performance and energy efficiency tradeoffs. These options provide flexibility to the orchestration framework developed in WP5.

Moreover, in order to overcome challenges such as limited computing and memory resources, transprecision and approximation computing techniques have been integrated into the kernel implementation of the HPC service. This approach allows for flexible execution with different data precisions and minimised computations. To address uncertainties and trade-offs between accuracy and execution runtime, a Verification, Validation, and Uncertainty Quantification (VVUQ) framework has been developed, suggesting parameters for kernel execution to optimise runtime and energy consumption. Additionally, approximation techniques such as precision scaling, approximate minimisation, and loop perforation have been employed in FPGA-accelerated application versions to enhance energy efficiency and expand the library of UC applications. Furthermore, work on algorithmic transprecise adaptation for distributed streaming applications edge/cloud computing systems were performed, aiming to reduce network latency and increase bandwidth.

WP4 also focused on meeting the increasing demands for high-performance computing and energy-efficient designs. It addresses the challenge of efficiently designing and deploying compute-intensive applications on accelerated platforms such as GPUs and FPGAs. The project developed FPGA and GPU accelerators using the Plug&Chip framework, which enables the automatic optimization of kernels for performance without human intervention and also introduces a methodology for memory-efficient accelerators on FPGAs.

Scaling the execution of hardware accelerated operations was addressed using the vAccel framework. This framework exposes hardware acceleration functionality to isolated serverless functions, allowing efficient execution and device selection. Enhancements are made to the vAccel framework in SERRANO to support arbitrary function execution, device selection, and remote execution through virtual or TCP sockets, thus enabling seamless deployment of hardware-accelerated applications in multi-tenant environments.

2.1 Document Structure

The document is structured as follows:

- This section (Section 1) introduces the activities performed during the reporting period.
- Section 2 provides an overview of kernel acceleration techniques in HPC, FPGA, and GPU devices. It presents the achieved speedup and energy gain resulting from these acceleration methods.
- Section 3 describes the transprecision and approximation computing techniques employed in kernel acceleration. It shows how these techniques contribute to improving execution time and reducing energy consumption.
- Section 4 demonstrates the Plug&Chip framework and the essential tools required for developing FPGA and GPU accelerators.
- Section 5 showcases the enhanced vAccel framework, which enables the execution of arbitrary hardware-accelerated operations as serverless functions within isolated contexts.
- Section 6 concludes the document, summarizing the developments in WP4.

3 HW/SW Acceleration Techniques & SERRANO Infrastructure Characterization

In this section the design, implementations and the evaluation results of the execution of the kernels of the UCs with FPGA, GPU and HPC accelerators for the cloud, edge and HPC resources are described. The accelerators that are described in this section compose the library of the accurate accelerators that are used in the SERRANO platform, as opposed to the approximation techniques described in Section 3.

The SERRANO platform features accelerators ranging from energy-efficient devices at the edge (e.g., NVIDIA Jetson and Xilinx MPSoC) to high-performance, massively parallel devices in the cloud (e.g., NVIDIA T4 and Xilinx Alveo), as well as Hawk supercomputer at HLRS. For more information on the devices used, refer to Deliverables D4.1 and D4.2. NVIDIA GPUs along with the programming model CUDA was used, while High Level Synthesis was used to create the designs for the FPGAs. For more information on the FPGA and GPU optimizations used by the team at AUTH, see Deliverable D4.1, which provides detailed explanations of each optimization.

With respect to HPC, the SERRANO platform incorporates HPC services that accelerate the kernels proposed by the use case providers in an HPC environment. The framework has been developed at HLRS, and the kernels were optimised to be executed on the Hawk supercomputer (Hawk). With the help of thousands of compute nodes, the kernels can process large volumes of data requiring intensive computation at a considerably faster rate.

The HPC kernels were accelerated by applying parallelization techniques, such as parallelization in shared and distributed memory using OpenMP/MPI. The MPI [1] communication library provided us with the protocol to communicate data across different processes that had disjoint memory address space, while OpenMP [2] provided us with parallelization in shared memory using multithreading. By leveraging these frameworks, we were able to speed up the execution time of the kernels and minimise energy consumption across different input data sizes provided by the use case providers.

3.1 Acceleration of the Secure Storage (UC1, Chocolate Cloud) Algorithms

Table 1 summarises the algorithms used in the workflow of the secure storage UC1. As baseline for execution time and the energy consumption, the metrics that are obtained by executing these algorithms on x86 and ARM based processor architectures are considered.

Table 1: Secure Storage (UC1) algorithms

Algorithm	Description
Encoder (EC)	Erasure coding encryption algorithm
Decoder (EC)	Erasure coding decryption algorithm
AES Encryption	AES-GCM 256 bits encoding algorithm
AES Decryption	AES-GCM 256 bits decoding algorithm

3.1.1 Erasure-Coding (EC) Encoder

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 the encrypted data before dispatching them in multiple secure locations. For more details on erasure-coding, refer to D4.1.

3.1.1.1 Design and Implementation

Two implementations of the accelerators for the EC encoder were developed. One exploits the computational resources of the acceleration cards (Alveo U50 and Alveo U200) and enables encoding to be performed in data chunks of up to 10MB each. This approach reduces the memory transfer time between the card's global memory and the compute region. The second implementation is deployed on the MPSoC FPGAs (ZCU102 and ZCU104) and performs encoding in smaller data chunks, limited by the platform's on-chip memory resources, with a maximum size of 100KB each. Consequently, frequent off-chip memory transactions are necessary when encoding large files. As described in D4.1, the computationally intensive kernel in the EC encoding algorithm involves multiplications and accumulations of large 2D arrays over a Galois Field. The number of columns in the arrays primarily depends on the size of the input data that will be encoded.

3.1.1.1.1 Alveo FPGA Acceleration Cards

An acceleration approach similar to loop-tiling was adopted to design the accelerator. Initially, the input data matrix is divided into N parts, with each part representing a portion of the initial input matrix containing input bytes stored in a number of TILE rows. Consequently, based on the design parameter TILE, the number of sub-matrices that constitute the input data is determined.

A MAC FPGA accelerator is designed and optimised to perform matrix multiplication on TILE-sized arrays. This optimization is achieved using the HLS unroll directive, resulting in the generation of multiple multipliers executed in parallel to calculate each output row. This optimization methodology is illustrated in Figure 51 in D4.1.

The optimised MAC accelerator calculates the output sub-matrix for the first TILE-sized array and transfers the result back to the card's global memory. It then reads the next TILE bytes and performs MAC operations on the subsequent sub-matrix. This process continues until all input sub-matrices have been processed.

For the design of the Alveo FPGA cards, the TILE parameter was set to 2000, and the number of parallel executed multipliers was set to 20. Figure 1 below illustrates the design of this accelerator.

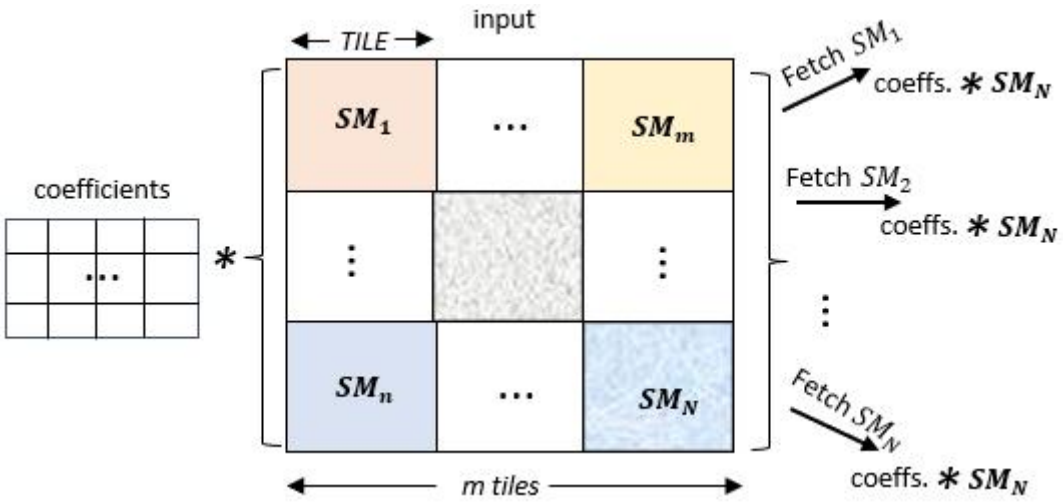


Figure 1: Acceleration design

Additionally, to enhance the overall acceleration, multiple compute units (CUs) were developed to perform the aforementioned operations in parallel.

For the design executed on the Alveo U50 card, 6 compute units were instantiated. On the other hand, the design implemented on the Alveo U200 card utilises 8 compute units.

3.1.1.1.2 Xilinx MPSoC FPGAs

The approach described in the previous subsection can be applied to design similar accelerators on the MPSoC platforms by adjusting the design parameters, such as TILE, the number of parallel executed multipliers, and the number of compute units. However, reducing the TILE parameter leads to more frequent off-chip memory transactions. Since the MPSoC platforms lack high-speed memories like the HBM memories available on the Alveo U50 card, this can result in significant execution overhead.

Therefore, for the design of the EC encoder accelerators on MPSoC FPGAs, the approach outlined in D4.1 is followed. Here is a brief summary:

On the MPSoC ZCU104 platform, two compute units are instantiated. Each compute unit is highly optimised for performing MAC operations on matrices of 100KB. These optimizations involve storing the input data matrix in multiple on-chip memories (BRAMs) and enabling

parallel multiplications. The input data is divided into chunks of 100KB each and processed by the two compute units.

A similar design is implemented on the ZCU102 platform, but in this case, 6 compute units are used in parallel for improved performance.

It is important to note that in these designs, although the number of memory transactions is higher compared to the Alveo implementations, the accelerators are aggressively optimised for the 100KB data sizes. Therefore, the overhead from frequent off-chip communication is not considered a bottleneck to the overall acceleration.

3.1.1.2 Evaluation Results

The results that are presented in the sections below show the application's execution and energy gains when it is executed on the selected FPGA platforms. Note that the application time consists of the time that is also required to set the execution environment, initialise the platform's buffers, perform all the memory transactions, and store the results in the host's memory space.

3.1.1.2.1 Alveo FPGA Acceleration Cards

Figure 2 shows the execution time speedup and the energy gains when those accelerators are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 2x and 2.1x for the U50 and U200 , while the energy gains are 3.1x and 1.7x respectively.

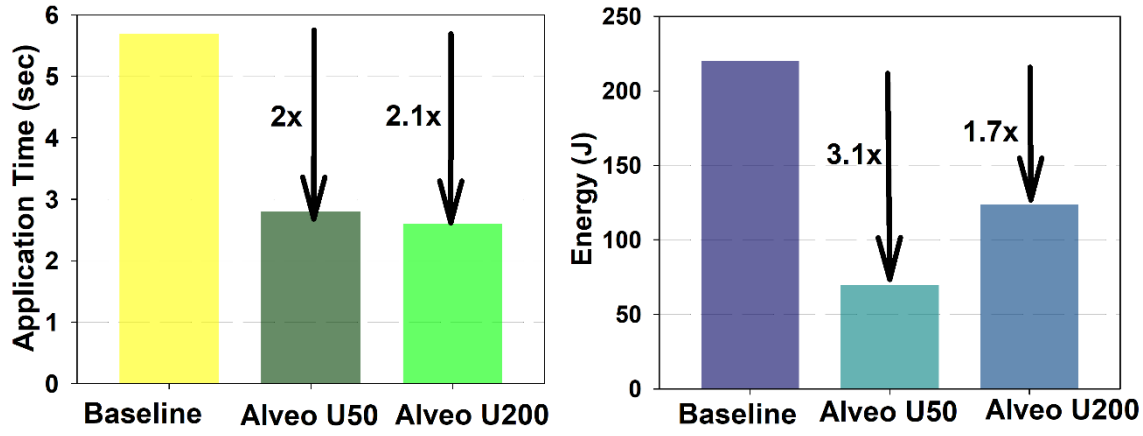


Figure 2: Execution time speedup and energy gains on the Alveo Xilinx acceleration cards

3.1.1.2.2 Xilinx MPSoC FPGAs

Figure 3 shows the execution time speedup and the energy gains when those accelerators are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 5.8x and 6.6x for the ZCU102 and ZCU104, while the energy gains are 10.6x and 7x respectively.

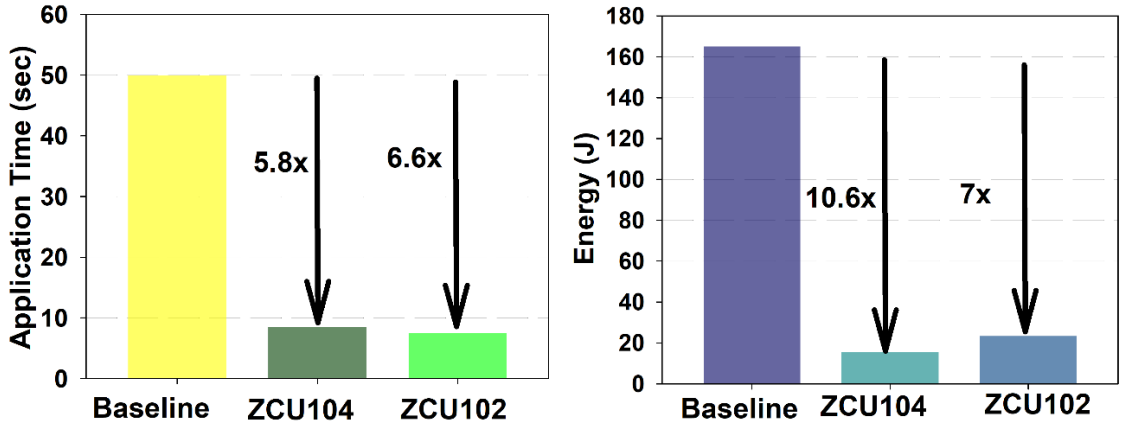


Figure 3: Execution time speedup and energy gains on the Xilinx MPSoC FPGAs

3.1.2 Erasure-Coding (EC) Decoder

The EC decoder algorithm takes input the coding coefficients, the encoded data and performs gaussian elimination on the encoded matrix. D4.1 provides more details on the algorithm.

3.1.2.1 Design and Implementation

Similar to the encoder's computationally intensive kernel, the core of this algorithm is the 2D matrix multiplication over the Galois Field. The FPGA designs are analogous to the ones that are mentioned in the section before (i.e the EC encoder) and perform the decoding in chunks of data.

3.1.2.1.1 Alveo FPGA Acceleration Cards

The designs for the Alveo U50 and U200 acceleration cards perform the decoding task in chunks up to 10MB each. The accelerator's ports that communicate with the global memory and transfer the encoded data, the coding coefficients and the decoded output were mapped to different memory banks in order to avoid latencies induced by a shared communication channel.

To enable a parallel computation of the elements that compose the decoded matrix, the *ARRAY_PARTITION* HLS directive was used on the encoding coefficients to store them in multiple on-chip memories and perform memory read and write operations in parallel.

Finally, to enable a task level parallelism scheme 10 compute units are instantiated both on the Alveo U50 and Alveo U200 card and are executed in parallel.

3.1.2.1.2 Xilinx MPSoC FPGAs

Similar to the MPSoC EC encoder accelerators, the accelerators developed for the decoder operate on chunks up to 100KB each. To optimise the accelerators' performance the *HLS UNROLL* directive was used to generate multiple GF multipliers that operate in parallel.

3.1.2.2 Evaluation Results

3.1.2.2.1 Alveo FPGA Acceleration Cards

Figure 4 shows the execution time speedup and the energy gains when those accelerators are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 1.8x and 2x for the U50 and U200, while the energy gains are 3.1x and 1.6x respectively.

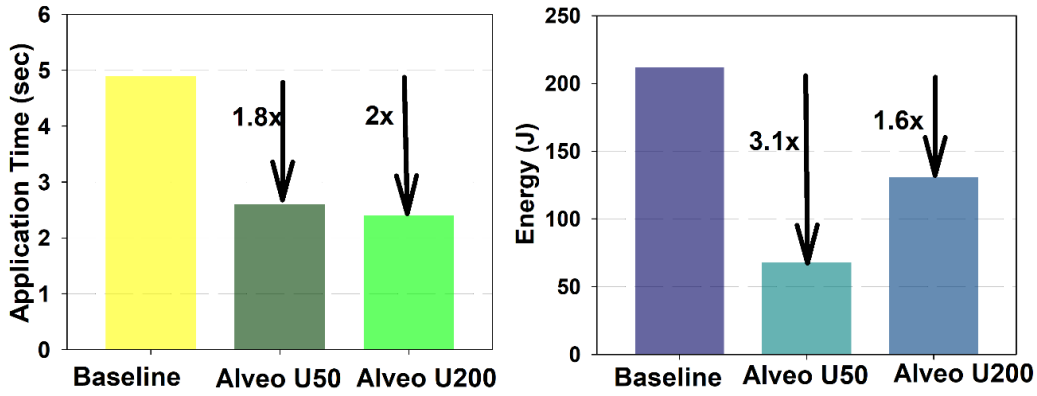


Figure 4: Execution time speedup and energy gains on the Alveo Xilinx acceleration cards

3.1.2.2.2 Xilinx MPSoC FPGAs

Figure 5 shows the execution time speedup and the energy gains when those accelerators are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 4.1x and 5.1x for the ZCU102 and ZCU104, while the energy gains are 6.1x and 5.9x respectively.

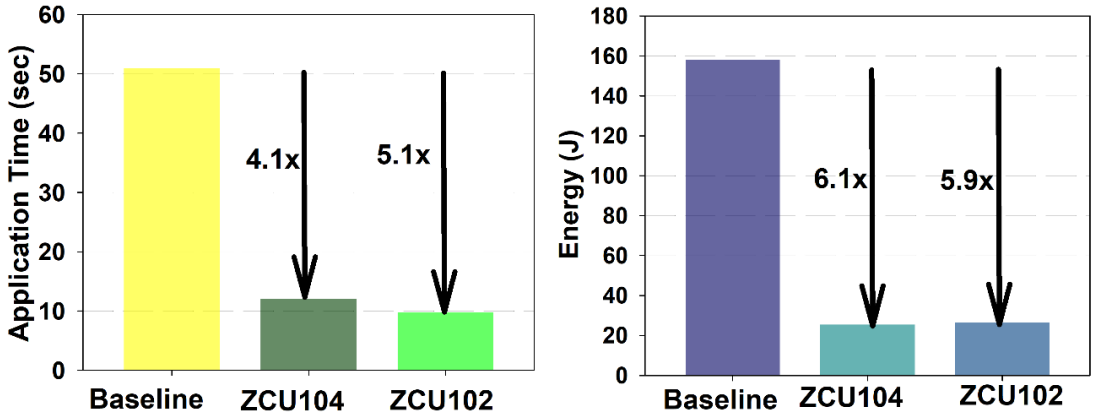


Figure 5: Execution time speedup and energy gains on the Xilinx MPSoC FPGAs

3.1.3 AES-GCM Encryption

AES-GCM is an encryption scheme based on Galois Message Authentication Code (GMAC.) It consists of two main functions, block cipher encryption and multiplication and is preferred for the high speed of authenticated encryption and data integrity that it provides. AES-GMC

Encryption is suitable to be employed in communication or electronic applications. For more information about AES-GCM Encryption, refer to D4.1.

3.1.3.1 Design Implementation

The AES-GCM encryption method was accelerated on GPU with CUDA programming model, following the methodology described in Deliverable D4.1. The implemented acceleration consists of three main parts. The first part is to read the AES block array and the key array and transfer them to the pageable GPU memory, using the *cudaMalloc()* and *cudaMemcpy()* methods. The second part, which is the core of the acceleration, is to define and execute the AES-GCM kernel. In practice we converted the serial C written loops to parallel CUDA kernels. The final third step is to transfer the encrypted result from the GPU memory to the CPU memory and write in a file the encrypted result. The shared memory was also adopted and thus a more efficient implementation was reached. Finally, we launched the encryption kernel with totally N threads organised in blocks of 1024 threads where N is the AES' block number.

3.1.3.1.1 NVIDIA Tesla T4 GPU

The acceleration for the NVIDIA Tesla T4 GPU is, as mentioned above, based on the three basic steps: (i) the Host to Device memory copy, (ii) the kernel execution, (iii) and the Device to Host memory copy. The *cudaMemcpy()* method was adopted for the first and last steps. For the kernel part, the serial for loops of the AE S-GCM encryption was parallelized using the CUDA programming model. For the parallelization and acceleration, the kernel was launched with totally of N threads(N:AES' block number) separated at blocks with 1024 threads per block that all had access to the shared memory, which was also adopted to optimise the implementation.

As depicted in Figure 6, shared memory is accessed from all the threads that are placed in the same block. All the threads from the same block can access data loaded from the global memory to the shared memory, and thus, the resulting kernel is more efficient. Finally, due to the fact that data were shared between threads, the danger of race conditions existed. Theoretically, all threads from a block run in parallel, but in practice, it is infeasible that all the threads will finish their execution synchronously. Thus, the threads have to be synchronised. For this purpose, *__syncthreads()* function was adopted, to be able to synchronise the threads

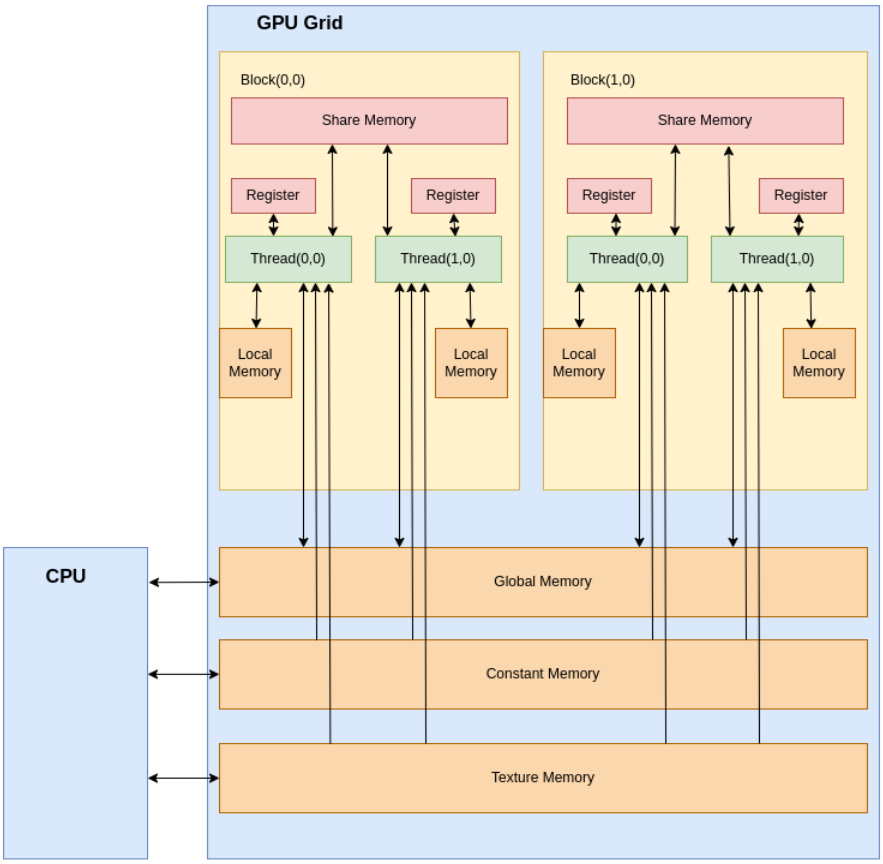


Figure 6: GPU Grid for NVIDIA Tesla T4 GPU

3.1.3.1.2 NVIDIA Jetson AGX

Similar to the Tesla T4 acceleration, the implementation for the NVIDIA Jetson AGX also had the same structure of the three basic stages, host to device memory copy, kernel execution, and device to host memory copy. Again, the adopted block size was 1024 with total of N threads (AES' block number). Finally, also in this case shared memory and threads synchronisation were also adopted to increase the efficiency of the acceleration.

3.1.3.2 Evaluation Results

The evaluation of the AES-GCM encryption acceleration was implemented on a 32MB custom input text file.

3.1.3.2.1 NVIDIA Tesla T4 GPU

Figure 7 shows the execution time speedup and the energy gains when this accelerator is executed on the Nvidia Tesla T4 GPU. The execution time speedup is 229.02x, while the energy gain is 301.8x.

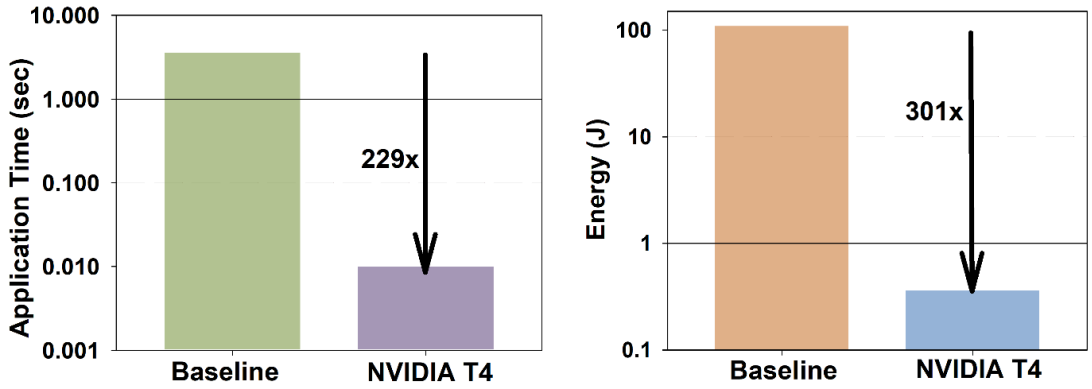


Figure 7: Execution time speedup and the energy gains on the Nvidia Tesla T4 GPU

3.1.3.2.2 NVIDIA Jetson AGX

Figure 8 shows the execution time speedup and the energy gains when this accelerator is executed on the Nvidia Jetson AGX GPU. The execution time speedup is 147.55x, while the energy gain is 45.7x.

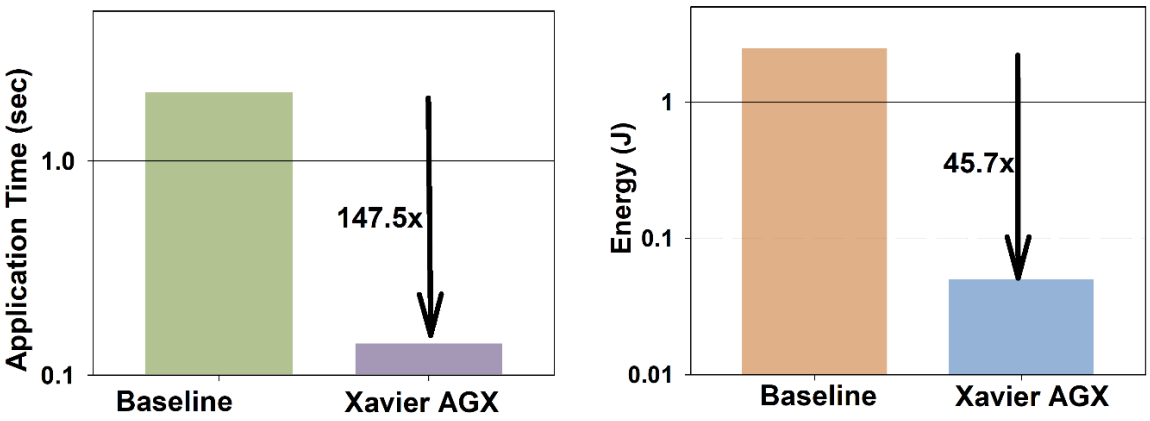


Figure 8: Execution time speedup and the energy gains on the Nvidia Jetson AGX GPU

3.1.4 AES-GCM Decryption

Similar to the AES-GCM Encryption, the AES-GCM Decryption scheme is also based on Galois Message Authentication Code (GMAC) and consists of block cipher encryption and multiplication operations. For more information, refer to D4.1.

3.1.4.1 Design Implementation

The AES-GCM decryption acceleration is similar to the implementation of the encryption. Again, it consists of three major steps, where the first and last one are for the data copy from the host to the device and from the device to the host, respectively. Also, the second step is the cuda kernel execution, where the for loop parallelization is implemented.

3.1.4.1.1 NVIDIA Tesla T4 GPU

For the Nvidia Tesla T4 GPU acceleration of the AES-GCM decryption, the above mentioned three basic steps were implemented. Additionally, the kernel launched totally N threads (AES' block) organised at blocks with block size 1024 threads per block. Finally, shared memory was once again leveraged to develop an efficient decryption acceleration.

3.1.4.1.2 NVIDIA Jetson AGX

Similar to the T4 acceleration, the acceleration for the Nvidia Jetson AGX GPU, was also implemented following the above three described steps (host to device memory copy, kernel execution and device to host memory copy). Finally, the same grid size and block size with the above T4 implementation were adopted. Analytically, a total number of N threads (N is the AES' block) was adopted and organised in blocks where the block size was set at 1024 threads per block. Similarly, we took advantage of shared memory, as described in Figure 6.

3.1.4.2 Evaluation Results

For evaluation purposes, a 32MB custom text file was firstly used, for encryption, with the encryption kernel which was described at the previous section and then the output was used to be decrypted with the implemented accelerated decryption function.

3.1.4.2.1 NVIDIA Tesla T4 GPU

Figure 9 shows the execution time speedup and the energy gains when this accelerator is executed on the Nvidia Tesla T4 GPU. The execution time speedup is 113.94x, while the energy gain is 155.25x.

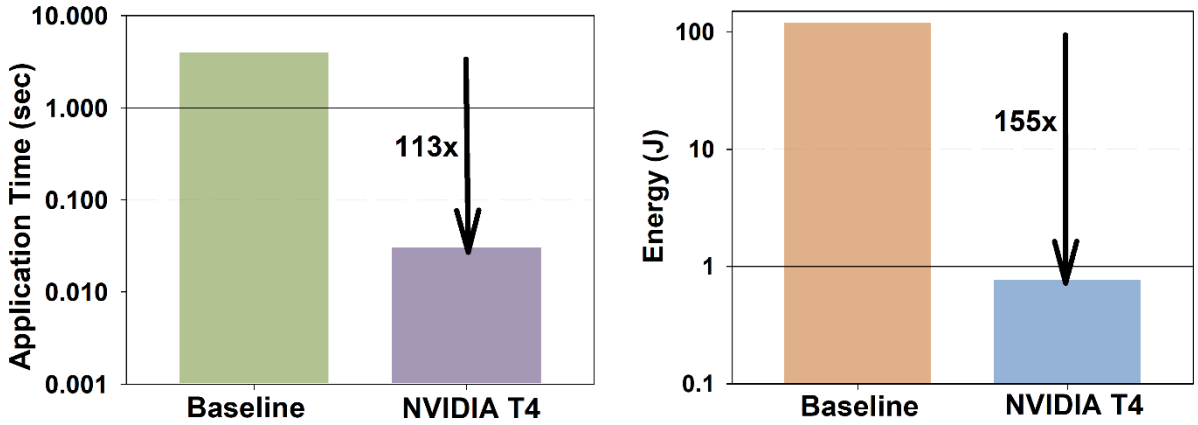


Figure 9: Latency and energy gains of AES decryption on T4 GPU

3.1.4.2.2 NVIDIA Jetson AGX

Figure 10 shows the execution time speedup and the energy gains when this accelerator is executed on the Nvidia Jetson AGX GPU. The execution time speedup is 58.25x, while the energy gain is 14.43x.

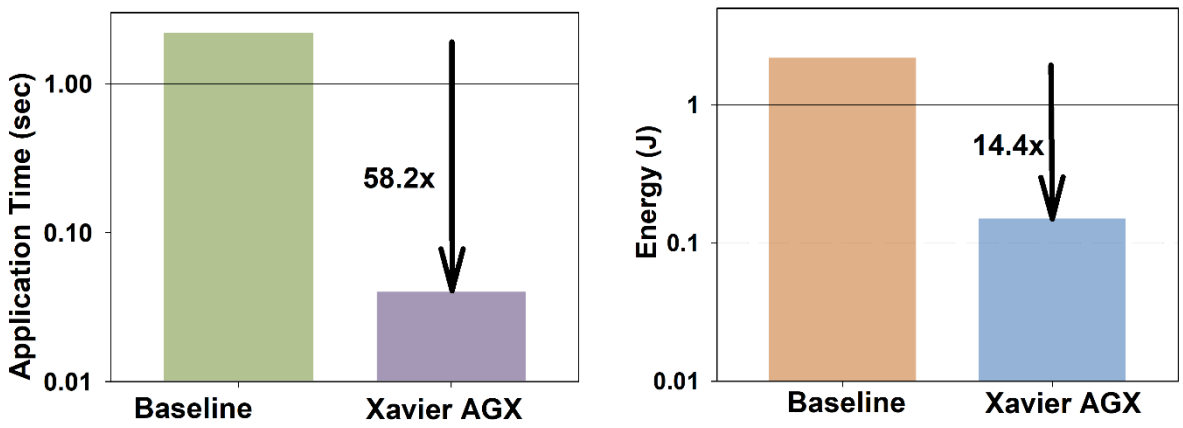


Figure 10: Latency and energy gains of AES decryption on Xavier AGX GPU

Figure 11 below summarises the results for all the FPGA and GPU designs for the UC1 algorithms.

Application	UC	Platform	Device	Latency (ms)	BRAM	DSP	FF	LUT	Grid size	Block size	Power (W)	Energy gain	Speedup
EC Encoder	CC	FPGA	U50	2847	83%	2.3%	6.5%	10.3%	–	–	24.8	3.1x	2x
EC Encoder	CC	FPGA	U200	2654	63.8%	2%	1.7%	8.6%	–	–	46.8	1.7x	2.1x
EC Encoder	CC	FPGA	ZCU104	12146	53.3%	2%	1.3%	11.7%	–	–	1.8	10.6x	5.8x
EC Encoder	CC	FPGA	ZCU102	9843	62.8x	0.3%	5.3%	13%	–	–	2.7	7x	6.6x
EC Decoder	CC	FPGA	U50	2631	84%	9.1%	6.5%	24.1%	–	–	25.8	3.2x	1.8x
EC Decoder	CC	FPGA	U200	2438	42%	0.2%	18.6%	34%	–	–	54.1	1.6x	2x
EC Decoder	CC	FPGA	ZCU104	8581	68%	6.1%	4.1%	9%	–	–	2.1	6.1x	4.1x
EC Decoder	CC	FPGA	ZCU102	7561	62.8%	0.3%	5.3%	13%	–	–	2.7	5.9x	5.1x
AES Encryption	CC	GPU	Tesla T4	13.9	–	–	–	–	2029	1024	26	301x	229x
AES Encryption	CC	GPU	Xavier AGX	14.4	–	–	–	–	2029	1024	3.9	45x	147x
AES Decryption	CC	GPU	Tesla T4	29.8	–	–	–	–	2029	1024	26	155x	113x
AES Decryption	CC	GPU	Xavier AGX	39.1	–	–	–	–	2029	1024	3.9	14.4x	58x

Figure 11: UC1 FPGA and GPU designs

3.2 Acceleration of the Fintech Analysis (UC2, InBestMe) Algorithms

Table 2 summarises the algorithms used in the workflow of the fintech UC2. As baseline for execution time and the energy consumption the metrics that are obtained by executing those algorithms on x86 and ARM based processor architectures are considered.

Table 2: Fintech Analysis (UC2) algorithms

Algorithm	Description
Savitzky-Golay	A moving window digital filter used for smoothing time-series
Kalman	A digital filter used for smoothing time-series
Wavelet	A db4 wavelet filtering transformation used for smoothing time-series
Black-Scholes	A mathematical formula for the calculation of the European call and put options

3.2.1 Savitzky-Golay (SAVGOL) Filter

The Savitzky-Golay [3] filter is a powerful tool in digital signal processing utilised for smoothing experimental data sets and reducing signal noise. By applying polynomial functions and considering neighbouring data points, this filter effectively removes high-frequency components from signals while preserving their overall shape and features. Details on the Savitzky-Golay filter can be found in Deliverable D4.1.

3.2.1.1 Design Implementation

Different accelerators for the execution of the Savitzky-Golay filter were implemented for its deployment at cloud and edge FPGA and GPU devices as well as and for HPC platforms. For the acceleration on FPGA and GPU devices, a design methodology has been developed [4].

3.2.1.1.1 Alveo FPGA Acceleration Cards

The designs for the Alveo U50 and U200 acceleration cards were developed following the design methodology that is described in Deliverable D4.1. Briefly, a dataflow mechanism that is composed of three distinct subunits was designed. This mechanism performs the memory read, write and the filter's moving window computations in a pipelined manner. In addition, to achieve a task level parallelism 10 compute units are instantiated. The execution of the SAVGOL filter on the UC time-series is performed on batches of 10, allowing the 10 compute units to work independently on the calculations of 10 different signals.

3.2.1.1.2 Xilinx MPSoC FPGAs

The dataflow design methodology that was developed for the FPGA acceleration of the SAVGOL filter leads to the generation of designs that utilise few computational resources, therefore the same design can be implemented on the MPSoC devices as well. However, in this case due to the platforms' limited resources 5 compute units are instantiated. This means that the UC timeseries are fed into the accelerators in batches of 5.

3.2.1.1.3 NVIDIA Tesla T4 GPU

For the implementation of the Savitzky-Golay filter acceleration on the Nvidia Tesla T4 GPU, we took advantage of the unified memory scheme that this GPU supports. Analyticity, instead of using different system (CPU) and device (GPU) memory we used the unified memory that is accessible from both CPU and GPU targeting to reduce the communication time cost and simplify the total implementation. The unified memory scheme is depicted in Figure 12. The acceleration flow in this case started with unified memory allocation using the *cudaMallocManaged()* method, and then, the kernel execution took place. The parts of memory copy at the beginning and at the end of the classical CUDA flow, in this case do not exist and thus, a better acceleration is reached. The grid size that we adopted in this case was ceil(N/Block Size) blocks, where N is the total number of data points of the time series and the

block size that we adopted was 32 threads per block. Also, some extra function for the data reading and writing were defined. All the data points for all the time series of the input dataset were read at the beginning of the execution and similarly, all the output data points for all the time series were written at the end of the execution.

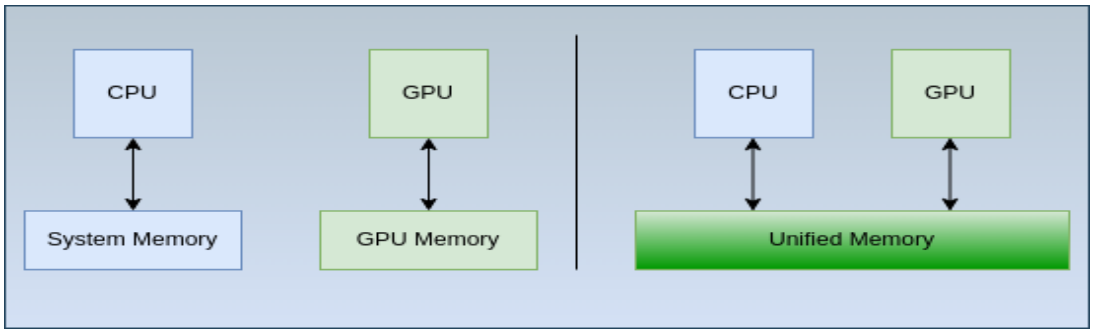


Figure 12: Unified memory scheme for CPU and GPU

3.2.1.1.4 NVIDIA Jetson Orin and Nano GPUs

The acceleration of the Savitzky-Golay filter for the Nvidia Jetson Orin, Nano and Xavier NX GPUs is based on the classical CUDA acceleration flow. First the data is transferred from the system (CPU) memory to the device (GPU) memory using malloc*()* and *cudaMalloc()* to allocate CPU and GPU memory respectively, and *cudaMemCpy()* method to copy the data. Then the kernel execution takes place and finally, the data transfer from the device (GPU) memory to the system (CPU) memory is implemented. For the kernels' implementation, the main task of the kernel is the parallelization of the for loop of SAVGOL filter. In practice the kernel was launched at a grid with block size at 32 threads per block for the Orin and 64 for the Nano and Xavier NX and grid size at ceil (N/Block Size), where N is the total number of data points of the input time series. Totally, N threads were used for the kernel launch.

3.2.1.1.5 HPC

In our collaboration with INBestMe, we received input data in CSV format, specifically asset prices. However, before utilising this data in the HPC environment, preprocessing was necessary. To accomplish this, we employed a data converter tool that has been developed alongside the HPC system. This tool converts the input data into a binary format, that is memory efficient compared to the original CSV format. Moreover, the data converter tool has the capability to generate data in various data precisions, including lower precision formats, such as float. Furthermore, the tool generates signals in a disjointed format with consecutive indexes. This format facilitates the uniform distribution of signals across multiple processes within the HPC system. By dividing the signals into disjoint segments, each process can handle a specific subset of the data independently, enabling parallel processing and enhancing overall performance and efficiency.

Once the data preprocessing with the data converter tool is complete, the original CSV data is transformed into disjoint segment signals in binary format. The task parallelization strategy can be employed in this scenario. By distributing the workload among multiple processes, each process can handle a specific subset of the data independently. This enables parallel processing, where multiple tasks are executed simultaneously. By utilising the Savitzky-Golay filter locally in each process, the signals can be filtered individually. This localised filtering approach can significantly enhance overall performance and efficiency.

3.2.1.2 Evaluation Results

3.2.1.2.1 Alveo FPGA Acceleration Cards

Figure 13 shows the execution time speedup and the energy gains when those accelerators are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 2.1x and 2.5x for the U50 and U200, while the energy gains are 2.1x and 1.3x respectively.

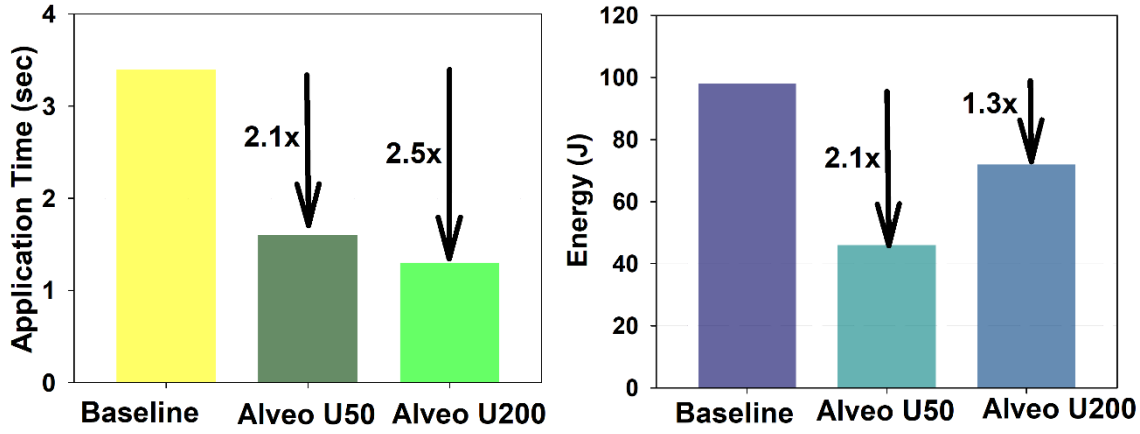


Figure 13: Latency and energy gains of SAVGOL on Alveo FPGAs

3.2.1.2.2 Xilinx MPSoC FPGAs

Figure 14 shows the execution time speedup and the energy gains when those accelerators are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 1.9x and 2.1x for the ZCU102 and ZCU104, while the energy gains are 1.8x and 2.2x respectively.

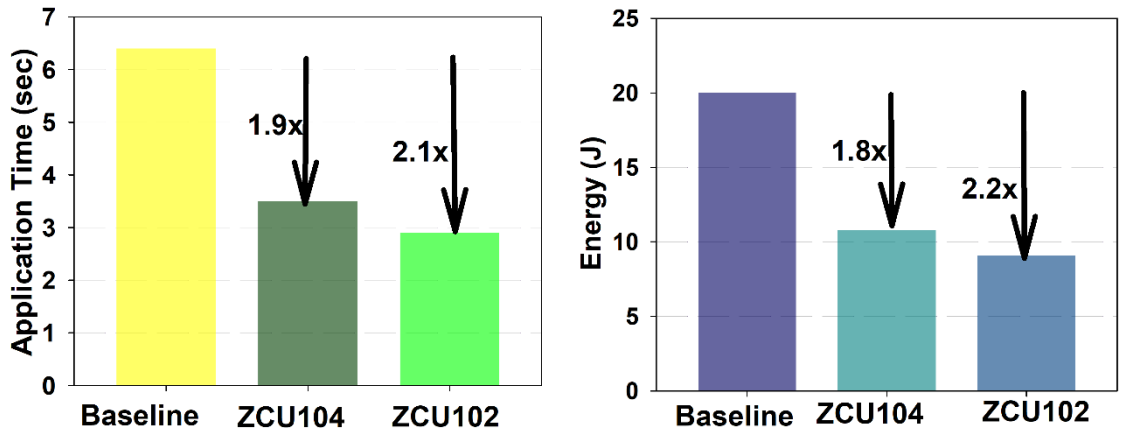


Figure 14: Latency and energy gains of SAVGOL on MPSoC FPGAs

3.2.1.2.3 NVIDIA Tesla T4 GPU

Figure 15 shows the execution time speedup and the energy gains when this accelerator is executed on the NVIDIA Tesla T4 GPU. The execution time speedup is 4.21x, while the energy gain is 4.48x.

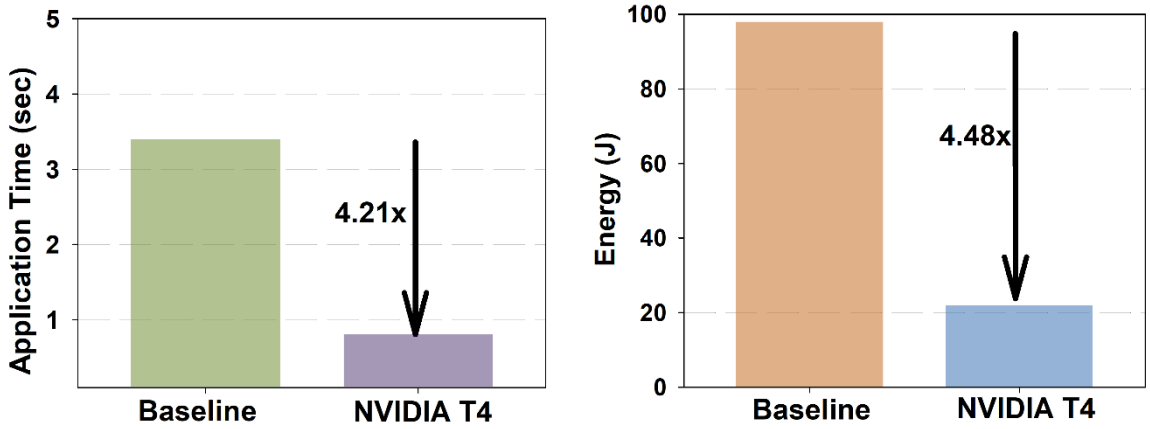


Figure 15: Latency and energy gains of SAVGOL on T4 GPU

3.2.1.2.4 NVIDIA Jetson Orin and Nano GPUs

Figure 16 shows the execution time speedup and the energy gains when those accelerators are executed on the NVIDIA Jetson Orin and NVIDIA Jetson Nano GPUs. The execution time speedups are 2.6x and 1.1x for the Orin and the Nano, while the energy gains are 3.94x and 2.99x respectively.

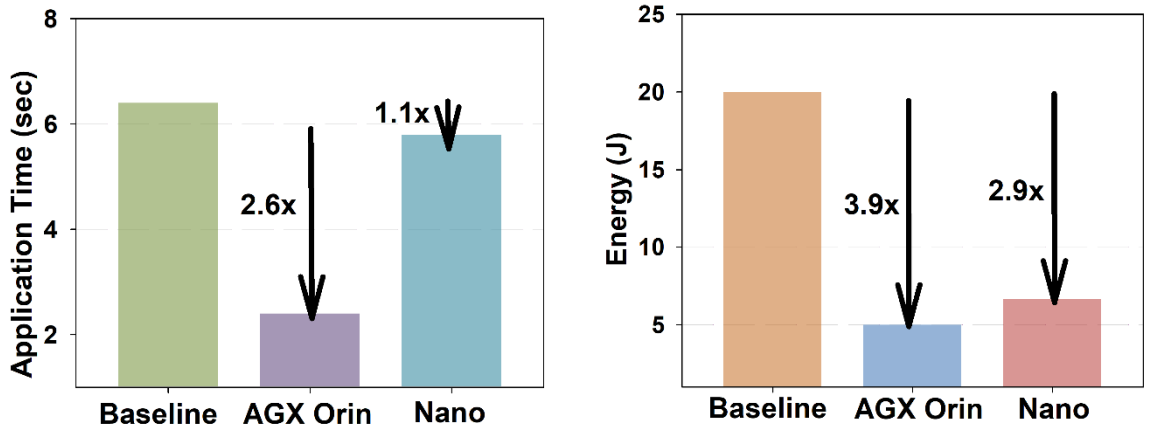


Figure 16: Latency and energy gains of SAVGOL on Orin and Nano GPUs

3.2.1.2.5 HPC

InBestMe supplied multiple data batches containing 100, 500, 1000, 2000 and 4879 asset prices. We conducted tests on the execution time and energy consumption of the kernel filters using these data sets on the Hawk compute nodes at HLRS. For this experiment, we utilised only one Hawk compute node with 128 cores, and its hardware architecture is described in the D4.2. Table 3 displays the speedup and energy gain of Savitzkey-Golay on the HPC system.

Table 3: Speedup and energy gain of Savitzky-Golay on HPC system

InBestMe Data	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
100 Asset Data	2X	2X	100%	0.217	48.01
500 Asset Data	17X	3X	100%	0.123	115.82
1000 Asset Data	29X	3X	100%	0.155	291.33
2000 Asset Data	20X	2X	100%	0.484	1813.51
4879 Asset Data	10X	2X	100%	2.035	3760.48

3.2.2 Kalman Filter

The Kalman filter [5], referred to as linear quadratic estimation, is a powerful technique used for estimating unknown variables and handles noise in measurements by taking into account statistical properties of the system dynamics and the measurement errors. The filter finds applications in diverse fields such as navigation, guidance, and finance. Details on Kalman filtering can be found in Deliverable D4.1.

3.2.2.1 Design Implementation

Kalman filtering has been implemented on all the different FPGA devices available on the SERRANO platform. It was also implemented using SERRANO's HPC infrastructure. The

accelerators for the cloud and edge devices were developed using the design methodology described in Deliverable D4.1. The provided input dataset contains 4890 stock price signals with 20000 data points and a total size of 4.5 GB.

3.2.2.1.1 Alveo FPGA Acceleration Cards

To provide an accurate FPGA-accelerated design, we do not use the Kalman filter implementation described in D4.1. Our acceleration strategy is based on the observation that the execution of the filtering algorithm is independent for different stock price time series and can therefore be easily parallelized. Task-level parallelism can be realised by instantiating multiple computational units on the FPGA, each of which performs Kalman filtering. We instantiate 8 compute units on the Alveo U50 FPGA, using the available High Bandwidth Memories (HBM). On the Alveo U200, 4 compute units are instantiated using the available Dynamic Random-Access Memories (DDR). Although the Alveo U200 contains more resources than the U50 FPGA, no more than 4 compute units can be instantiated during synthesis due to place-n-route issues.

In the Kalman filter kernel, the loops responsible for transferring data to/from the programmable logic and the loop that performs the actual computation are pipelined with an initiation interval of one. Besides the calculation loop, where the target initiation interval cannot be achieved due to data dependencies, the other loops are pipelined with the targeted initiation interval.

3.2.2.1.2 Xilinx MPSoC FPGAs

We use the same Kalman filtering acceleration strategy for the MPSoC ZCU104 and ZCU102 devices. The proposed implementation does not require many resources and can therefore be implemented on the resource-constrained edge FPGAs of the SERRANO platform. For both devices, we instantiate 2 compute units using the available High Performance (HP) memories. For the Kalman filter kernel, we use exactly the same approach as described for the cloud FPGAs.

The main difference between the accelerated versions targeting the edge FPGAs is based on the available RAM memory of the boards. In particular, the MPSoC ZCU104 has 2 GB RAM memory, while the MPSoC ZCU102 has 4 GB. The final version of the stock price dataset provided by INB has a size of about 4.5 GB, which makes clear that it cannot be processed at once. To overcome the memory limitation, we divide the dataset into smaller parts that fit into the memory and process them one by one. The official dataset is divided into 30 equally sized stock price datasets for the MPSoC ZCU104 device and 15 for the MPSoC ZCU102.

3.2.2.1.3 HPC

The parallelization strategy employed in the Kalman filter is similar to that of the Savitzky-Golay filter. In this case, the data preprocessing step involves converting the data into binary format, divided into disjoint segments with their respective indices. The signals are then distributed across multiple processing units, allowing for workload distribution.

Once the signals are distributed, the Kalman filter operates independently on each signal to filter out noise and produce the desired output result. This parallel processing approach enables efficient noise removal and estimation of the true state of each signal.

3.2.2.2 Evaluation Results

3.2.2.2.1 Alveo FPGA Acceleration Cards

Figure 17 shows the speedup and energy gains of the accurate version of the Kalman filter for the Alveo U50 and U200 FPGAs compared to the Python single-threaded execution on an Intel(R) Core (TM) i5-6500 CPU @3.2GHz. The speedups are 3074x and 1880x for the U50 and U200, while the energy gains are 5049x and 1884x, respectively. It is evident that our implementation outperforms the baseline. We can also see that the Alveo U50 FPGA is able to achieve a higher speedup compared to the Alveo U200, which is due to the fewer instantiated compute units.

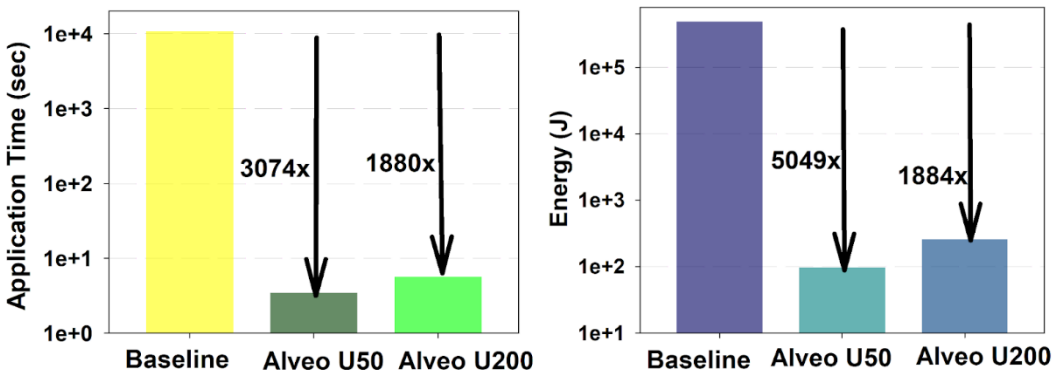


Figure 17: Latency and energy gains of Kalman on Alveo FPGAs

3.2.2.2.2 Xilinx MPSoC FPGAs

Figure 18 shows the speedup and energy gains of the accurate version of the Kalman filter for the MPSoC ZCU104 and ZCU102 FPGAs compared to the Python single-threaded execution on an ARM Cortex A53 @800MHz. The speedups are 1510x and 869x for the ZCU104 and ZCU102, while the energy gains are 2875x and 1806x, respectively. With respect to the Python baseline, the same observations can be made as for the cloud FPGAs. Another interesting observation is that the design for the MPSoC ZCU104 achieves a higher speedup compared to the corresponding design for the MPSoC ZCU102. This is due to the fact that it takes more time to allocate the buffers on ZCU102 compared to ZCU104. In particular, buffer allocation on ZCU102 takes about 30 seconds, whereas on ZCU104 it takes only 7 seconds (4.3x faster).

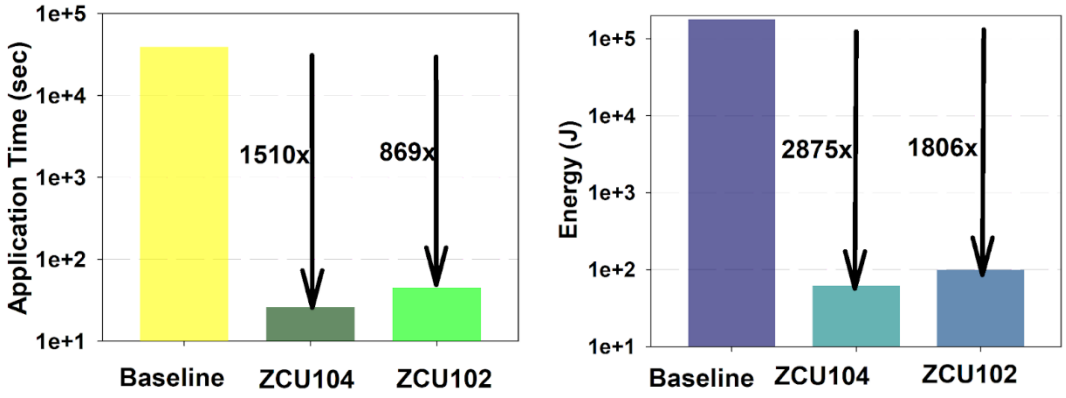


Figure 18: Latency and energy gains of Kalman on MPSoC FPGAs

Finally, the designs targeting the cloud FPGAs lead to higher speedups and energy gains due to the RAM limitation of the edge FPGAs. Processing the input dataset in batches introduces an overhead, making the cloud accelerators more suitable for Kalman filter processing.

3.2.2.2.3 HPC

Table 4 displays the speedup and energy gain of the parallel Kalman filter inside HPC service. HLRS conducted an experiment using multiple asset prices data batches from InBestMe. Similarly, the data used for the Savitzky-Golay filter.

Table 4: Speedup and energy gain of Kalman Filter on HPC system

InBestMe Data	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
100 Asset Data	4X	2X	100%	0.105	23.36
500 Asset Data	13X	3X	100%	0.178	166.7
1000 Asset Data	37X	4X	100%	0.105	198.1
2000 Asset Data	14X	2X	100%	0.530	1324.5
4879 Asset Data	10X	2X	100%	2.035	4476.98

3.2.3 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). Similar to the Savitzky Golay and Kalman filters, wavelet is used to smooth the UC time series. The UC2 uses the Daubechies 4 (db4) [6] wavelet for the smoothing procedure. In the Daubechies wavelets the

filter's length is typically equal to eight (8) and this is the length that was selected by the UC2. More information on the UC2 wavelet algorithm can be found in the deliverable D4.1.

3.2.3.1 Design Implementation

Accelerators for the computationally intensive DWT part of the wavelet filtering algorithm were developed for the cloud and edge FPGA and GPU devices as well as for the HPC platforms. Note, that the process of inverting the filtered signal to reconstruct the original (but smoothed prices) is not executed on the developed accelerators but on the general-purpose host devices.

3.2.3.1.1 Alveo FPGA Acceleration Cards

Similar to the Savitzky Golay filter, the Discrete Wavelet Transform (DWT) process is executed using a moving window mechanism. At each point, the coefficients of the db4 wavelet are convolved with the input time-series. To accelerate this process, a dataflow execution mechanism, similar to the one implemented for the Savitzky Golay filter on the Alveo U50 and U200 acceleration cards, was adopted in this case. First, the coefficients for the high-pass decomposition and the low-pass reconstruction are transferred into the FPGA's local memory. Next, two circuits that perform the convolutions on the first eight and the last eight input values are designed and executed in parallel. Last, the dataflow mechanism performs the DWT computations for the rest input points following a pipelined approach. To design this mechanism ping pong buffers [7] that enable out-of-order read and write operations are used for storing temporarily the input data. Figure 19 illustrates this design.

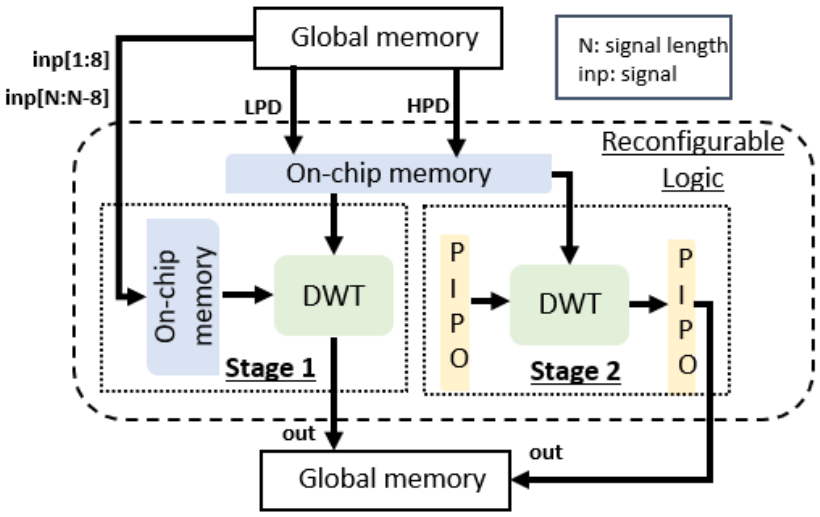


Figure 19: Wavelet acceleration mechanism

Finally, the outputs from the DWT algorithm are transferred back to the host device where the inverse discrete wavelet transform (IDWT) is executed and the smoothed prices are produced. To enable a task-level parallelism 6 compute units are instantiated on both the U50 and the U200 and are executed in parallel.

3.2.3.1.2 Xilinx MPSoC FPGAs

Similar design implementations to the ones that were developed for the Alveo cards were created for the two MPSoC FPGAs. However, due to the limited resources of those devices, two compute units instead of six are instantiated and are executed in parallel.

3.2.3.1.3 NVIDIA Tesla T4 GPU

In the case of Wavelet filter acceleration, the implemented acceleration was based on the methodology described in Deliverable D4.1 (M15). In advance, the classical CUDA acceleration flow was followed. This flow includes memory to device copy at the pageable GPU memory, kernel execution and device to host memory copy. Then we launched our kernel with N total threads, where N is the length of the input array of the input signal to be processed. Also, the block size was set at 32 threads per block and thus, the grid size was set at ceil (N/block size). It has to be mentioned that the *dwt_sym_stride()* function, which is the wavelet's most time consuming algorithmic part, was accelerated on the CUDA kernel.

3.2.3.1.4 NVIDIA Jetson AGX GPU

Similar to the T4 implementation, the acceleration for the Nvidia Jetson AGX is also based on the classical CUDA acceleration flow. And also in this case the block size was set at 32 threads per block and the grid size was set at ceil (N/block size).

3.2.3.2 Evaluation Results

3.2.3.2.1 Alveo FPGA Acceleration Cards

Figure 20 shows the execution time speedup and the energy gains when those accelerators are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 3.5x and 3.9x for the U50 and U200, while the energy gains are 3.8x and 2.4x respectively.

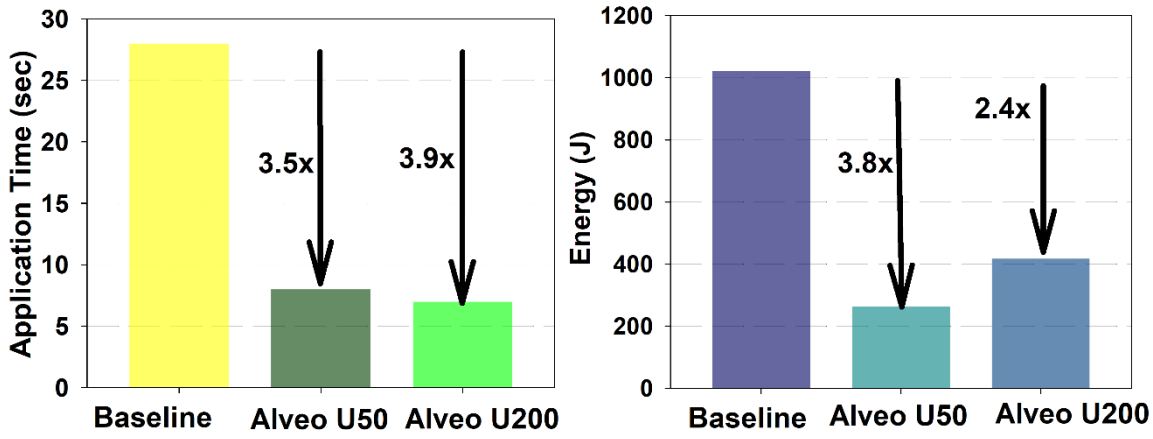


Figure 20: Latency and energy gains of Wavelet on Alveo FPGAs

3.2.3.2.2 Xilinx MPSoC FPGAs

Figure 21 shows the execution time speedup and the energy gains when those accelerators are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 1.9x and 2x for the ZCU102 and ZCU104, while the energy gains are 1.2x and 1.6x respectively.

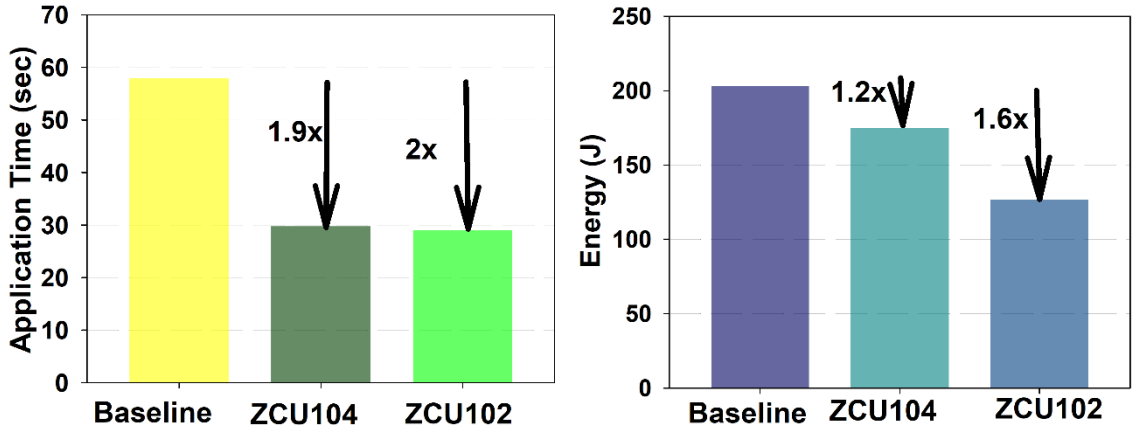


Figure 21: Latency and energy gains of Wavelet on MPSoC FPGAs

3.2.3.2.3 NVIDIA Tesla T4 GPU

Figure 22 shows the execution time speedup and the energy gains when this accelerator is executed on the NVIDIA Tesla T4 GPU. The execution time speedup is 5.84x, while the energy gain is 47.5x.

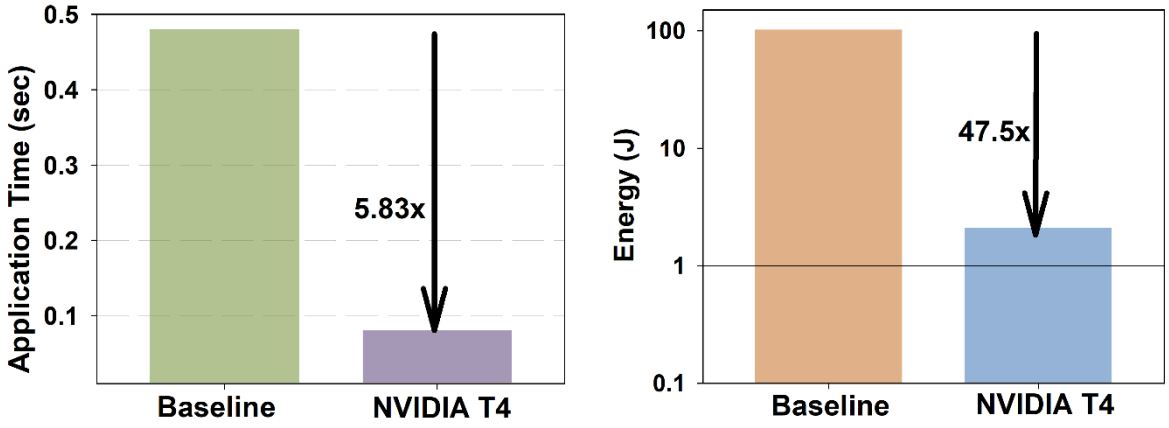


Figure 22: Latency and energy gain of Wavelet on T4 GPU

3.2.3.2.4 NVIDIA Jetson AGX GPU

Figure 23 shows the execution time speedup and the energy gains when this accelerator is executed on the NVIDIA Jetson AGX GPU. The execution time speedup is 2.87x, while the energy gain is 58.1x.

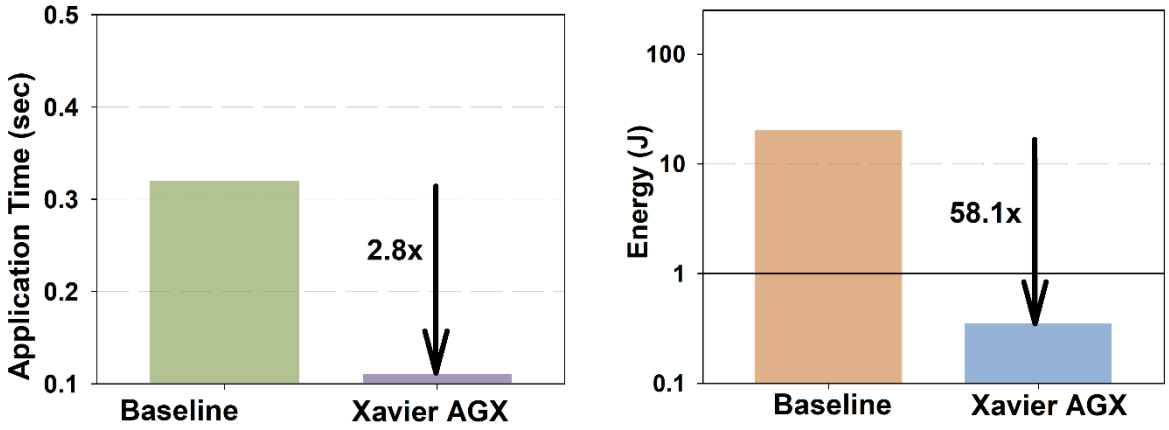


Figure 23: Latency and energy gains of Wavelet on Xavier AGX GPU

3.2.4 Black-Scholes Algorithm

The Black-Scholes [8] also known as the Black-Scholes-Merton model, is a mathematical formula used to calculate the theoretical price of options. The formula provides a way to estimate the fair value of European-style options, which are financial derivatives that give the holder the right to buy (call option) or sell (put option) an underlying asset at a predetermined price (strike price) within a specific time period. By considering various factors, such as the current price of the underlying asset, the strike price, the time to expiration, the risk-free interest rate, and the volatility of the asset's price, the Black-Scholes formula produces an option price that reflects the market's expectations.

3.2.4.1 Design Implementation

UC2 uses the Black-Scholes formula to calculate the call and put options for each price of every one of their assets. Therefore, the calculation of the following two formulas for all the prices is accelerated:

Table 5: Black-Scholes formula

```
Call option: C = S * N(d1) - X * e^(-r * T) * N(d2)
Put option: P = X * e^(-r * T) * N(-d2) - S * N(-d1),
Where:
d1 = (ln(S/X) + (r + (σ^2)/2) * T) / (σ * sqrt(T))
d2 = d1 - σ * sqrt(T)
```

In those formulas:

- *C* represents the price of the call option.
- *P* represents the price of the put option.
- *S* is the current price of the underlying asset.
- *X* is the strike price of the option.
- *T* is the time to expiration of the option, expressed in years.
- *r* is the risk-free interest rate.

- *N(x)* represents the cumulative standard normal distribution function.

Accelerators for the calculation of the Black-Scholes put and call option prices were developed on cloud and edge FPGA and GPU devices as well as on HPC resources.

3.2.4.1.1 Alveo FPGA Acceleration Cards

To accelerate the calculations of the put and call option prices, the designs that were developed on the U50 and U200 perform the computations using a dataflow mechanism. Specifically, in every clock cycle the accelerator reads a price and its expiry date from an asset and stores them in two First-In-First-Out (FIFO) buffers. The arithmetic circuits (multipliers and adders) that are required for the calculation of the Black-Scholes formulas are implemented to utilise the platform's DSP blocks for enhanced performance. Then, based on the user's request for the calculation of call or put options, multiplexers enable the arithmetic blocks that compute the call or put options respectively.

Finally, in order to parallelize the computations for the two option prices, two compute units are instantiated on both Alveo platforms. The first compute unit calculates only the put options for all prices while the second only the call, as a result the two options are computed in parallel for every price. Figure 24 below illustrates the design for the Black-Scholes acceleration.

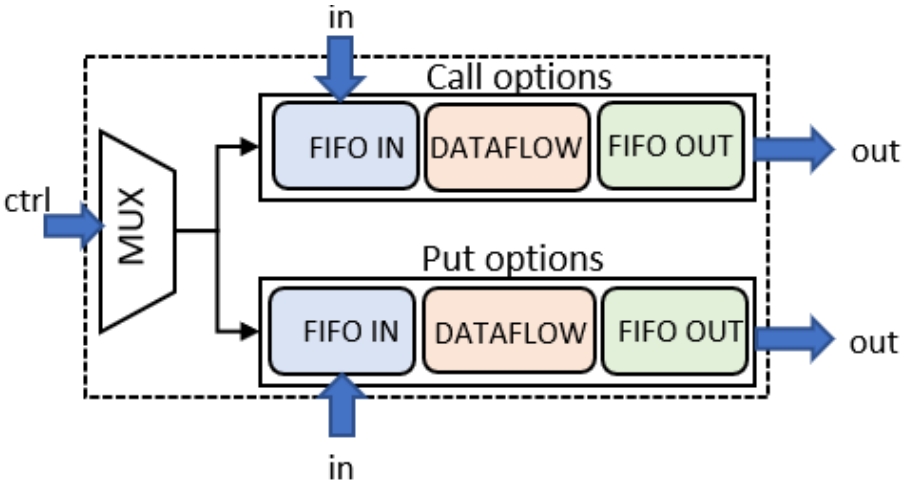


Figure 24: Acceleration approach

3.2.4.1.2 Xilinx MPSoC FPGAs

The designs that were developed for the Alveo cards can be also used on the MPSoC platforms without modifications. Due to the dataflow approach the number of the utilised computational resources is limited and doesn't exceed the MPSoCs available resources.

3.2.4.1.3 NVIDIA Tesla T4 GPU

For the acceleration of put and call options computation on the Nvidia Tesla T4, we took advantage of the unified memory. Unified memory is accessible from both CPU and GPU and

thus reduces the development complexity (simplifies the implementation) and also reduces the total execution time. The acceleration contains the unified memory allocation part, with the *cudaMallocManaged()* method and the kernel execution part. At the kernel for loops for the put and the call option prices computation were parallelized using CUDA. The adopted grid size for this implementation was ceil (N/block size), where N is the length of the processed signal and the adopted block size was 32 threads per block.

3.2.4.1.4 NVIDIA Jetson Orin and Nano GPUs

In the case of Jetson Orin, Nano and Xavier NX GPUs, the implemented acceleration was based on the classical CUDA acceleration flow which contains memory to device copy at the GPU memory, kernel execution and device to host memory copy. Then the kernel was launched with N total threads, where N is the length of the input array of the input signal to be processed. Also, the block size was set at 32 threads per block and the grid size was set at ceil (N/block size).

3.2.4.1.5 HPC

To accelerate the Black-Scholes kernel, a similar strategy as the Savitzky-Golay kernel is employed. In this case, the data will be initially converted into binary representation based on its indices. The signals will then be distributed among multiple processes. Each process will compute the output parameters such as put price and option price locally for the batch data prices. This approach helps to parallelize the computation and enhance the performance of the Black-Scholes kernel.

3.2.4.2 Evaluation Results

3.2.4.2.1 Alveo FPGA Acceleration Cards

Figure 25 shows the execution time speedup and the energy gains when those accelerators are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 36x and 52.5x for the U50 and U200, while the energy gains are 47.8x and 42x respectively.

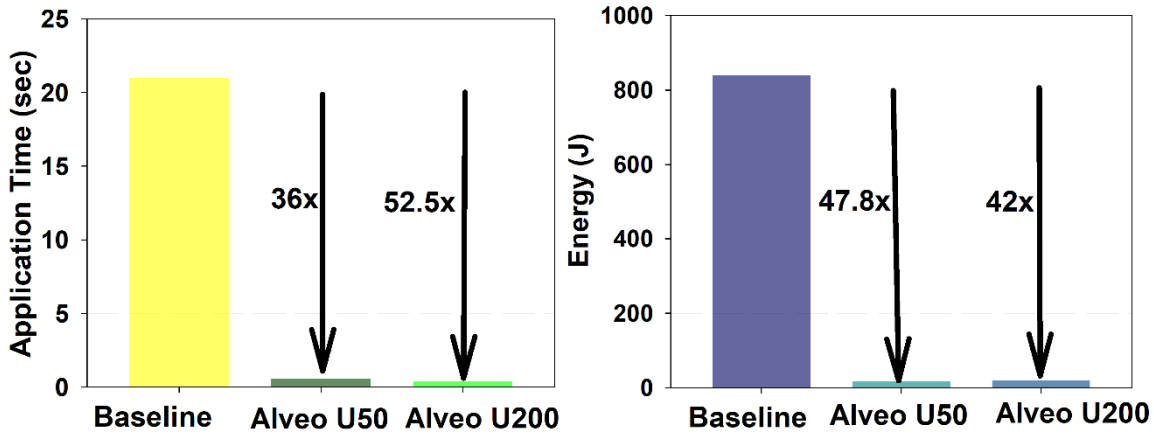


Figure 25: Latency and energy gains of Black-Scholes on Alveo FPGAs

3.2.4.2.2 Xilinx MPSoC FPGAs

Figure 26 shows the execution time speedup and the energy gains when those accelerators are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 130x and 117x for the ZCU102 and ZCU104, while the energy gains are 81x and 72x respectively.

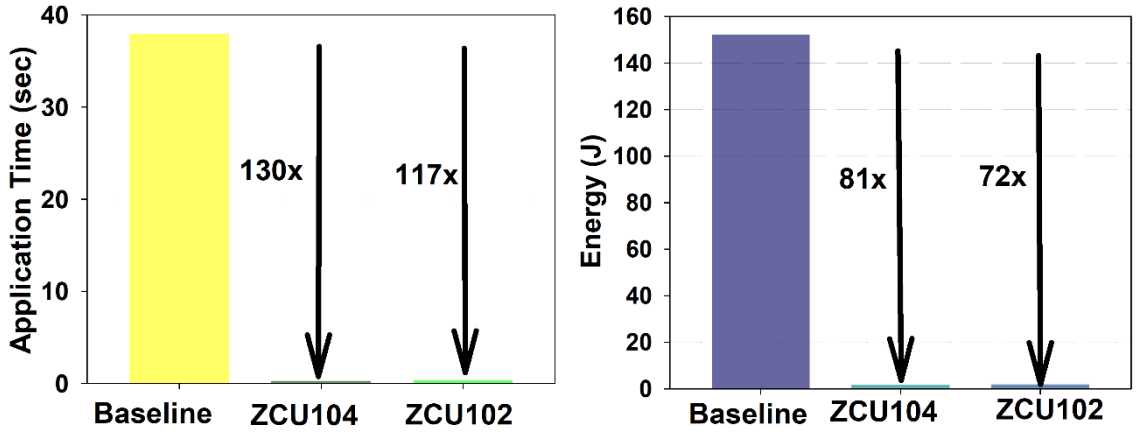


Figure 26: Latency and energy gains of Black-Scholes on MPSoC FPGAs

3.2.4.2.3 NVIDIA Tesla T4 GPU

Figure 27 shows the execution time speedup and the energy gain when this accelerator is executed on the NVIDIA Tesla T4 GPU. The execution time speedup is 39615.62x, while the energy gain is 60947.1x.

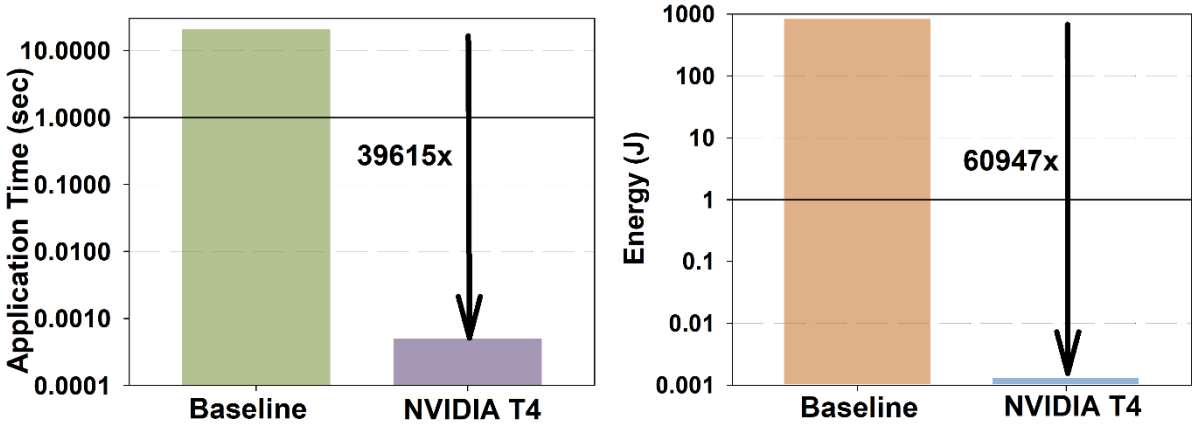


Figure 27: Latency and energy gains of Black-Scholes on T4 GPU

3.2.4.2.4 NVIDIA Jetson Orin and Nano GPUs

Figure 28 shows the execution time speedups and the energy gain when these accelerators are executed on the NVIDIA Jetson Orin and Nano GPUs. The execution time speedups are 12874x for the Jetson Orin and 1545.28x for the Jetson Nano, while the energy gains are 26626.69x and 4204.85x, respectively.

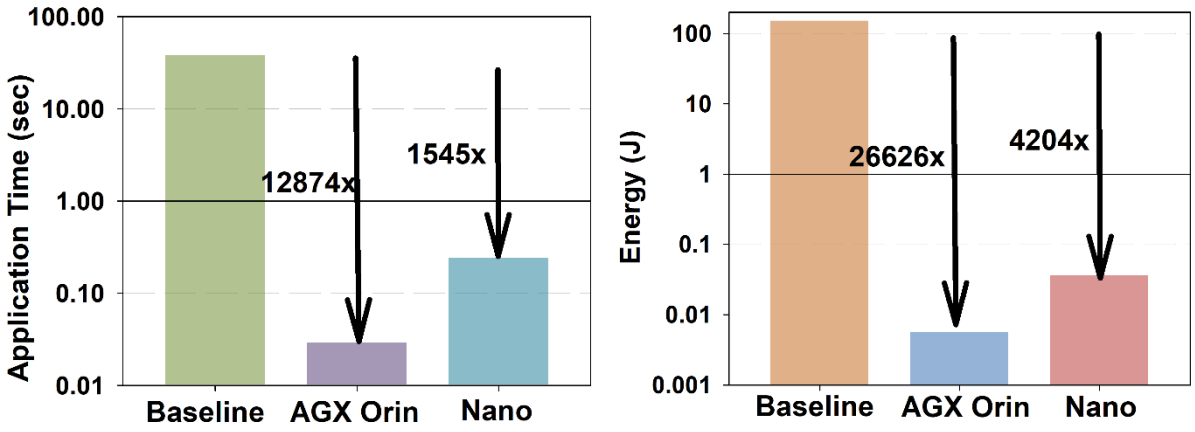


Figure 28: Latency and energy gains of Black-Scholes on Orin and Nano GPUs

3.2.4.2.5 HPC

Table 6 demonstrates the speedup and energy consumption of the Black-Scholes inside the HPC service, an experiment was conducted by HLRS using multiple asset price data batches from InBestMe, similar to the data used for the Savitzky-Golay filter.

Table 6: Speedup and energy gain of Blach-Scholes on HPC system

InBestMe Data	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
100 Asset Data	29.4X	21X	100%	0.0738	16.285
500 Asset Data	60X	11X	100%	0.1642	153.59
1000 Asset Data	136X	13X	100%	0.1551	290.11
2000 Asset Data	322X	14X	100%	0.1352	505.89
4879 Asset Data	572X	12X	100%	0.1856	4476.98

Figure 29 below summarises the results for all the FPGA and GPU designs for the UC1.

Application	UC	Platform	Device	Latency (ms)	BRAM	DSP	FF	LUT	Grid size	Block size	Power (W)	Energy gain	Speedup
Savitzky-Golay	INB	FPGA	U50	1654	0.8%	9.2%	12.2%	31%	–	–	28.2	2.1x	2x
Savitzky-Golay	INB	FPGA	U200	1343	0.7%	8%	10%	26%	–	–	54.1	1.3x	2.5x
Savitzky-Golay	INB	FPGA	ZCU104	3500	1.7%	15.9%	17.1%	30.6%	–	–	3.1	1.8x	1.9x
Savitzky-Golay	INB	FPGA	ZCU102	2940	0.5%	10.9%	14.3%	25.5%	–	–	3.1	2.2x	2.1x
Savitzky-Golay	INB	GPU	Tesla T4	808	–	–	–	–	625	32	27	4.4x	4.2x
Savitzky-Golay	INB	GPU	AGX Orin	2465	–	–	–	–	313	64	2	3.9x	2.5x
Savitzky-Golay	INB	GPU	Jetson Nano	5800	–	–	–	–	625	32	2.1	2.9x	1.1x
Kalman	INB	FPGA	U50	3512	40%	0.9%	0%	2.2%	–	–	27.5	5049x	3074x
Kalman	INB	FPGA	U200	5743	6.2%	0.4%	2.1%	1.6%	–	–	45	1884x	5064x
Kalman	INB	FPGA	ZCU104	25966	34%	2%	2%	6%	–	–	2.4	2875x	2875x
Kalman	INB	FPGA	ZCU102	45096	12%	2%	2%	4%	–	–	2.2	1806x	1806x
Wavelet	INB	FPGA	U50	8000	8.7%	35%	27.4%	49%	–	–	33	3.8x	3.5x
Wavelet	INB	FPGA	U200	7018	7.6%	30.5%	21.4%	38%	–	–	59.5	2.4x	3.9x
Wavelet	INB	FPGA	ZCU104	29800	15.6%	52.8%	34.8%	52.4%	–	–	5.9	1.2x	1.2x
Wavelet	INB	FPGA	ZCU102	29000	4.1%	36.2%	28%	42%	–	–	4.4	1.6x	1.6x
Wavelet	INB	GPU	Tesla T4	826	–	–	–	–	625	32	26	47.5x	5.8x
Wavelet	INB	GPU	Xavier AGX	1123	–	–	–	–	625	32	3.1	58.1x	2.8x
Black Scholes	INB	FPGA	U50	580	2%	46.4%	22.3%	56%	–	–	30.3	47.7x	36x
Black Scholes	INB	FPGA	U200	400	1.3%	40.7%	17.6%	47%	–	–	50	42x	39x
Black Scholes	INB	FPGA	ZCU104	291	8.8%	91%	40.3%	88.8%	–	–	6.4	81.6x	81.6x
Black Scholes	INB	FPGA	ZCU102	322	2.7%	62.4%	34.7%	82.4%	–	–	6.5	72.6	72.6x
Black Scholes	INB	GPU	Tesla T4	0.53	–	–	–	–	167	32	26	60947x	39615x
Black Scholes	INB	GPU	AGX Orin	2.95	–	–	–	–	167	32	1.9	26626x	12874x
Black Scholes	INB	GPU	Jetson Nano	24.5	–	–	–	–	167	32	1.4	4204x	1545x

Figure 29: UC2 FPGA and GPU designs

3.3 Acceleration of the Anomaly Detection in Manufacturing Settings (UC3, IDEKO) Algorithms

Table 7 summarises the algorithms used in the workflow of the anomaly detections in manufacturing environments UC3. As baseline for execution time and the energy consumption the metrics that are obtained by executing those algorithms on x86 and ARM based processor architectures are considered.

Table 7: Algorithms' acceleration for Anomaly Detection in Manufacturing Settings

Algorithm	Description
DBSCAN	Unsupervised learning clustering algorithm used for the anomaly detection
K-Means	Unsupervised learning clustering algorithm used for the anomaly detection
k-NN	Supervised learning clustering algorithm used for the anomaly detection
1D-FFT	1 dimensional Fast Fourier Transform (FFT) used for smoothing signals

3.3.1 DBSCAN Clustering Algorithm

Density-based spatial clustering of applications with noise [9] (DBSCAN) is a density-based non parametric clustering algorithm, i.e. given a set of data elements in a given space, it groups the elements that are close to each other and flags the data elements that are alone in low-density regions as outliers. In the IDEKO use case, DBSCAN is used to detect anomalies in a set of signals (i.e., to classify them as anomalous or non-anomalous), indicating a machine malfunction. To quantify the similarity between two signals, the Dynamic Time Warping [10] (DTW) algorithm is used.

After analysing the provided source code, we concluded that the computationally intensive part is the calculation of the DTW distances of all the different signals, so we only accelerated the DTW distance calculation. Details on DBSCAN can be found in Deliverable D4.1.

3.3.1.1 Design Implementation

DBSCAN has been implemented on all the different FPGA devices available on the SERRANO platform. The accelerators for the edge and cloud devices were developed using the design methodology described in Deliverable D4.1. The provided input dataset contains 110 signals with 3400 data points and a total size of 7.3MB

3.3.1.1.1 Alveo FPGA Acceleration Cards

Our acceleration approach is based, as in the Kalman filter, on parallelism at the task level. In particular, the calculation of the DTW distance of two signals does not depend on other DTW distance calculations and can therefore be calculated in parallel. We instantiate 8 compute units on the Alveo U50 FPGA, using the available HBMs. On the Alveo U200, 4 compute units are instantiated using the available DDRs. Although the Alveo U200 contains more resources than the U50 FPGA, no more than 4 compute units can be instantiated during synthesis due to place-n-route issues.

DTW is a dynamic programming algorithm, which means that the calculation of the current element depends on the calculation of previous ones. The dependencies in dynamic programming algorithms make parallel computing difficult. Our approach, therefore, was to pipeline the loops responsible for transferring data to/from the programmable logic and the loop that performs the actual computation. Apart from the calculation loop, where the target initiation interval cannot be achieved due to data dependencies caused by dynamic programming, the other loops are pipelined with an initiation interval of one.

3.3.1.1.2 Xilinx MPSoC FPGAs

We use the same acceleration strategy for the MPSoC ZCU104 and ZCU102 devices. The proposed implementation does not require many resources and can therefore be implemented on the resource-constrained edge FPGAs of the SERRANO platform. For both devices, we instantiate 2 compute units using the available High Performance (HP) memories. For the DTW distance computation kernel, we use exactly the same approach as described for the cloud FPGAs.

3.3.1.2 Evaluation Results

3.3.1.2.1 Alveo FPGA Acceleration Cards

Figure 30 shows the speedup and energy gains of the accurate version of the DTW distance computation for the Alveo U50 and U200 FPGAs compared to the Python execution. The speedups are 343x and 99x for the U50 and U200, while the energy gains are 152x and 26x,

respectively. It is obvious that our implementation performs better than the baseline solution in terms of execution time and energy consumption. We can also see that the Alveo U50 FPGA is able to achieve a higher speedup compared to the Alveo U200, which is due to the fewer instantiated compute units.

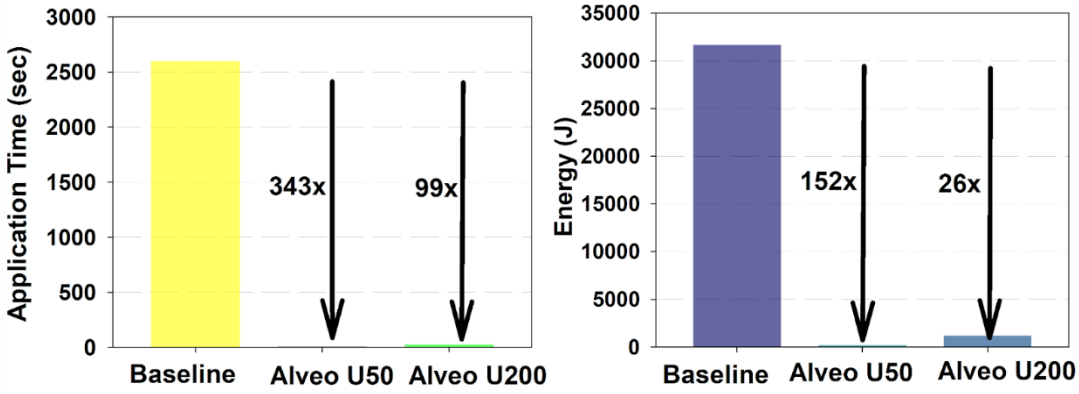


Figure 30: Latency and energy gains of DBSCAN on Alveo FPGAs

3.3.1.2.2 Xilinx MPSoC FPGAs

Figure 31 shows the speedup and energy gains of the accurate version of the DTW distance computation for the MPSoC ZCU104 and ZCU102 FPGAs compared to the Python single-threaded execution. The speedups are 17.8x and 13.6x for the ZCU104 and ZCU102, while the energy gains are 42.5x and 36x, respectively. As in the case of the Kalman filter, this is due to the fact that buffer allocation on ZCU102 takes more time compared to ZCU104. In particular, buffer allocation on ZCU102 takes about 54 seconds, while on ZCU104 it takes only 4.4 seconds (12.3x faster).

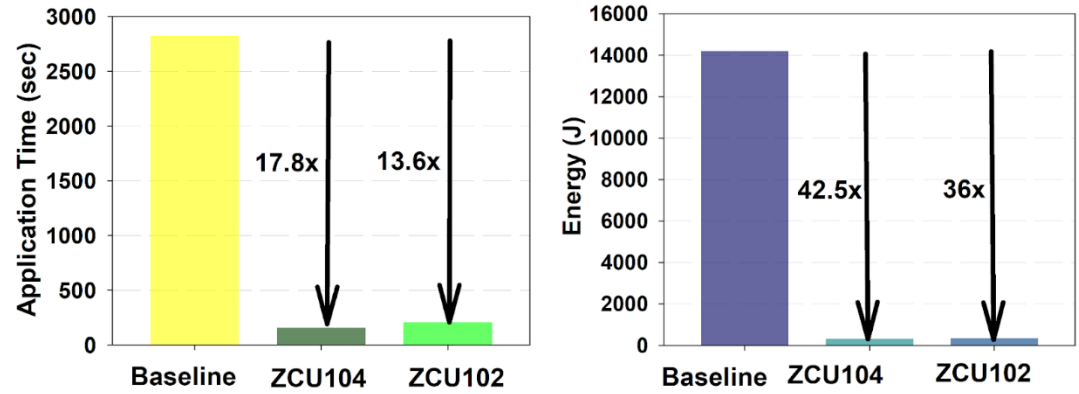


Figure 31: Latency and energy gains of DBSCAN on MPSoC FPGAs

3.3.2 1D-FFT Algorithm k-Means Clustering Algorithm

The Fast Fourier Transform [11] (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence or its inverse (IDFT). Fourier analysis converts a signal from its original domain to a frequency domain representation and vice versa. The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields, but computing directly from the definition is often too slow. An FFT quickly computes such transformations by decomposing the DFT into a product of sparse factors. This can reduce the complexity of computing DFT from O(N^2), which results from simply applying the definition of DFT, to O(NlogN), where N is the data size.

3.3.2.1 Design Implementation

FFT has been implemented on all the different FPGA devices available on the SERRANO platform. It was also implemented using SERRANO's HPC infrastructure. The accelerators for the edge and cloud devices were developed using the design methodology described in Deliverable D4.1. The provided input dataset contains 520 signals with 16384 data points and a total size of 81MB.

3.3.2.1.1 Alveo FPGA Acceleration Cards

Our approach to accelerating the FFT kernel is based on Xilinx's FFT IP library [12] and dataflow processing [13]. The dataflow mechanism consists of three subunits that perform reading, writing, and FFT calculation in a pipelined manner. In addition, to achieve task-level parallelization, multiple computational units are instantiated to process different signals in parallel. We instantiate 4 compute units on the Alveo U50 and U200 FPGAs and use the available HBMs and DDRs, respectively.

3.3.2.1.2 Xilinx MPSoC FPGAs

We use the same acceleration strategy for the MPSoC ZCU104 and ZCU102 devices. The proposed implementation does not require many resources and can therefore be implemented on the resource-constrained edge FPGAs of the SERRANO platform. For both devices, we instantiate 2 compute units using the available High Performance (HP) memories. For the FFT kernel, we use exactly the same approach as described for the cloud FPGAs.

3.3.2.1.3 HPC

IDEKO provides signal data in CSV format for processing, which needs to be preprocessed before it can be processed in the HPC environment. The data converter tool is used for this purpose. Firstly, it reads signals by their indices from the original CSV format, and secondly, it converts the signals into binary format using specific data precision, such as double or single precision (Figure 32).

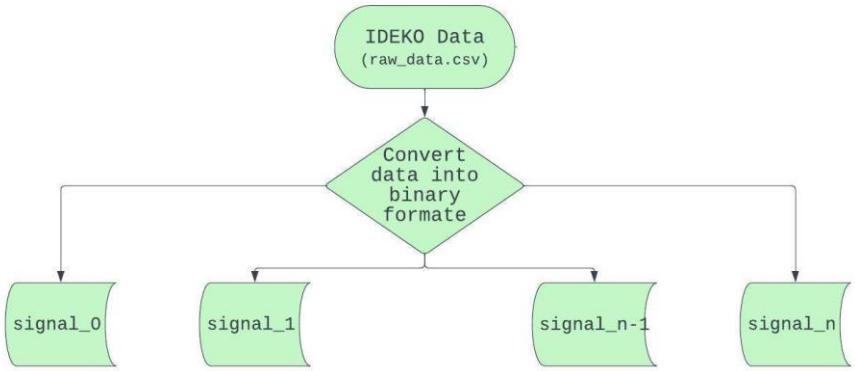


Figure 32: Converting the CSV data into binary format

Once the conversion process is complete, the disjoint signals are produced in binary format, as shown in Figure 32, and are ready for processing by kernels such as FFT. To achieve optimal performance, a parallelization strategy is employed, specifically task parallelization. This approach involves distributing the signals by their index to different processes (Figure 33), allowing for workload distribution among many processes. Each process can process a batch of signals, as there is no data dependency, and each signal can be processed independently.

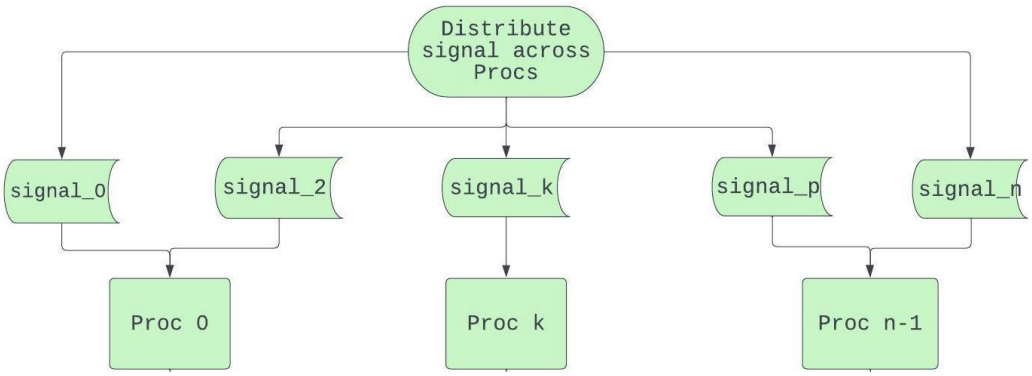


Figure 33: Uniform distribution of signals among processors

Figure 33 illustrates the uniform distribution of signals across different processes. Each process is assigned specific signals, with *process 0* accessing *signal_0*, *signal_1*, and *process n-1* assigned to process *signal_n-1* and *signal_n*. By distributing signals uniformly among processors, each processor has a roughly equal workload, ensuring optimal performance. This approach is based on task distribution and helps to maximise efficiency while reducing processing time.

Once the signals have been available locally by each processor, the FFT filter can act on each time series signal independently, as shown in Figure 34.

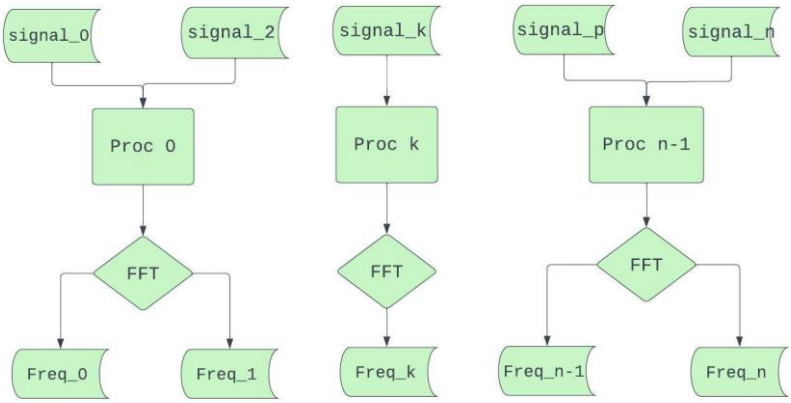


Figure 34: FFT performs on signals batches in processor

Taking the FFT of these signals, they are transformed into the frequency domain, and the amplitude of the spectrum is extracted as the output. In the final stage, the output data is aggregated, a CSV file is generated, and then returned to the providers of the use case. The data workflow of parallel FFT, which is the strategy we will apply in the remaining kernels, is presented in Figure 35.

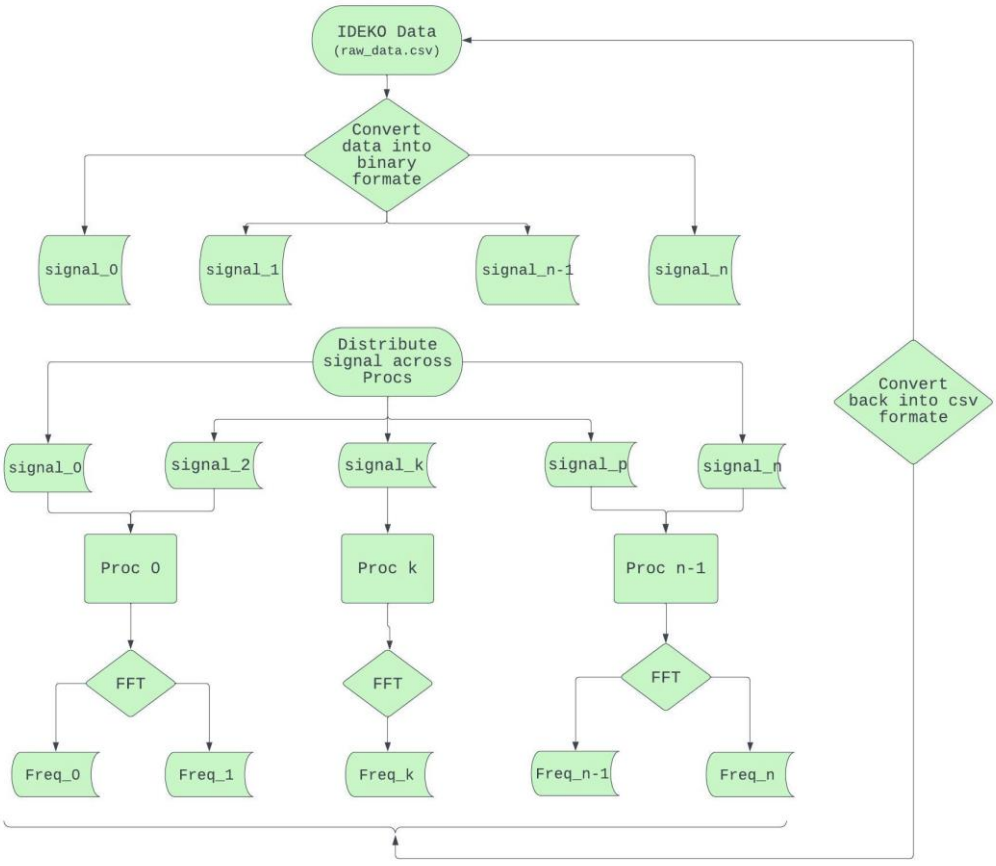


Figure 35: Data workflow of parallel FFT

3.3.2.2 Evaluation Results

3.3.2.2.1 Alveo FPGA Acceleration Cards

Figure 36 shows the speedup and energy gains of the accurate version of the FFT for the Alveo U50 and U200 FPGAs compared to the Python single-threaded execution. The speedups are 38.2x and 48.4x for the U50 and U200, while the energy gains are 50x and 34.8x, respectively. It is obvious that our implementation performs better than the baseline solution in terms of execution time and energy consumption. We can also see that the Alveo U200 FPGA is able to achieve a higher speedup compared to the Alveo U50, which is due to the more available resources of the device, allowing the creation of a more efficient design in terms of performance. However, the higher performance comes at the cost of higher power consumption. For example, the power consumption of the Alveo U200 is 50.6 watts, while that of the Alveo U50 is 27.8 watts (1.8x lower).

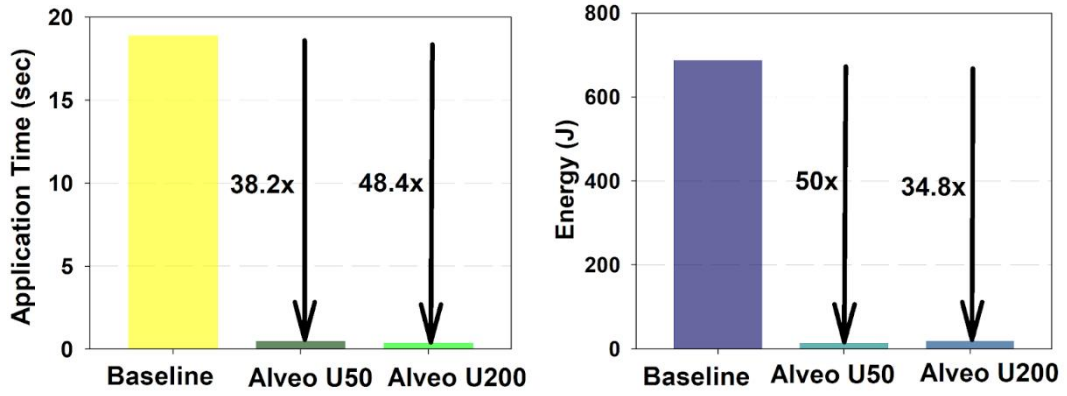


Figure 36: Latency and energy gains of 1D-FFT on Alveo FPGAs

3.3.2.2.2 Xilinx MPSoC FPGAs

Figure 37 shows the speedup and energy gains of the accurate version of the Kalman filter for the MPSoC ZCU104 and ZCU102 FPGAs compared to the Python single-threaded execution. The speedups are 18.9x and 7.2x for the ZCU104 and ZCU102, while the energy gains are 30x and 11.2x, respectively. This is due to (a) the fact that buffer allocation takes more time on the ZCU102 compared to the ZCU104, and (b) the timing issues encountered during the synthesis process when the target frequency for the ZCU102 was set at 300 MHz. To obtain the final design, we set a target frequency of 250 MHz, which results in higher execution time and thus higher energy consumption.

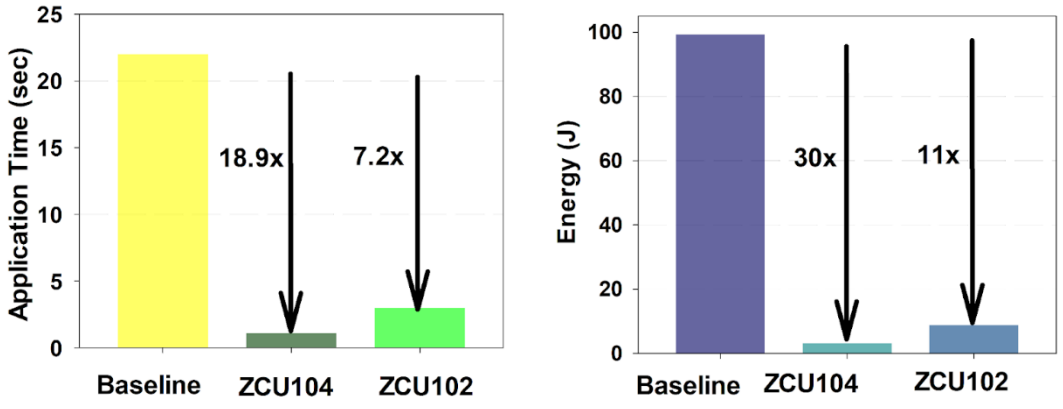


Figure 37: Latency and energy gains of 1D-FFT on MPSoC FPGAs

3.3.2.2.3 HPC

HLRS conducted tests on our parallelization framework using several batches of signals provided by IDEKO. Initially, we examined the effects of increasing the number of processes on the execution runtime and energy consumption time of 104 acceleration signals. Our findings showed in Figure 38 that increasing the number of processors resulted in a decrease in execution runtime and energy consumption. HLRS has presented in Table 8 the speedup and energy gain through the parallelization of the FFT kernel with various signal batches provided to us by IDEKO, and verifying the accuracy of the results, were done by them.

Table 8: Speedup and energy gain of FFT on HPC system

IDEKO Signal	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
26 cycle signal	5X	4X	100%	0.072	33.21
104 cycle signal	29X	10X	100%	0.053	23.76
156 cycle signal	26X	12X	100%	0.099	33.21
208 cycle signal	37X	7X	100%	0.079	79.82
234 cycle signal	85X	15X	100%	0.040	35.48
260 cycle signal	72X	11X	100%	0.057	64.33

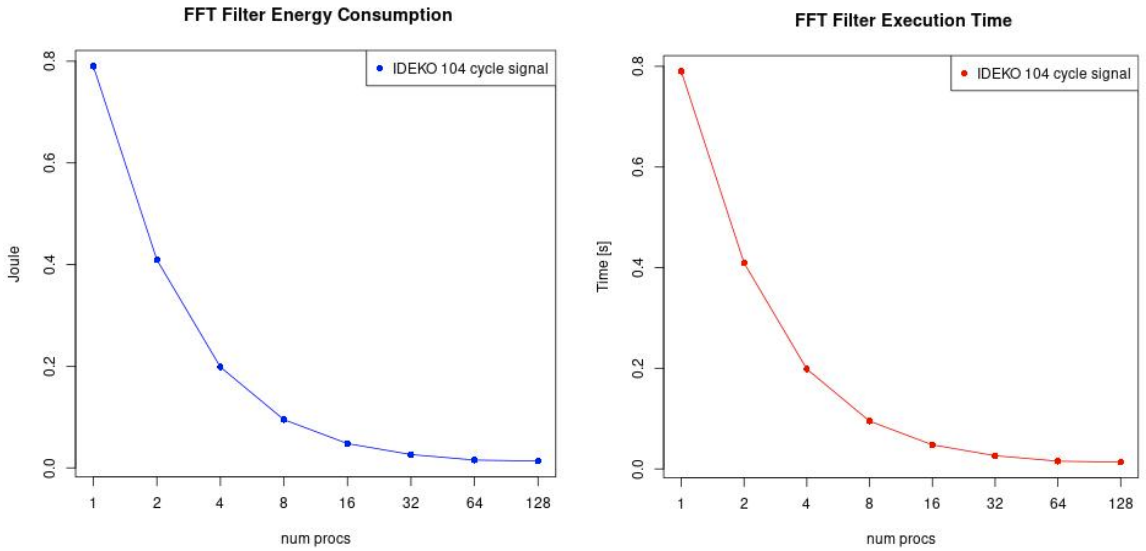


Figure 38: FFT Filter Energy Consumption and Execution across the number of processors

3.3.3 K-Means Clustering Algorithm

K-Means is a clustering algorithm that can be used for the classification of time series signals. The basic idea of K-Means is to group data points into a specified number of clusters, based on their similarity to one another. In this instance, we are grouping position time series signals into two clusters based on their similarity. The signals are classified into clusters that are in close proximity to each other. Typically, the Euclidean metric is used to measure distance in K-Means, but in this case, we use DTW [14] (Dynamic Time Warping) as it offers greater flexibility when matching time series signals with varying shapes, lengths, and alignments.

The left side of Figure 39 displays the time series position signals, which serve as the input data for K-Means classification. Upon application of K-Means classification with DTW metric, the position signals will be separated into two distinct clusters.

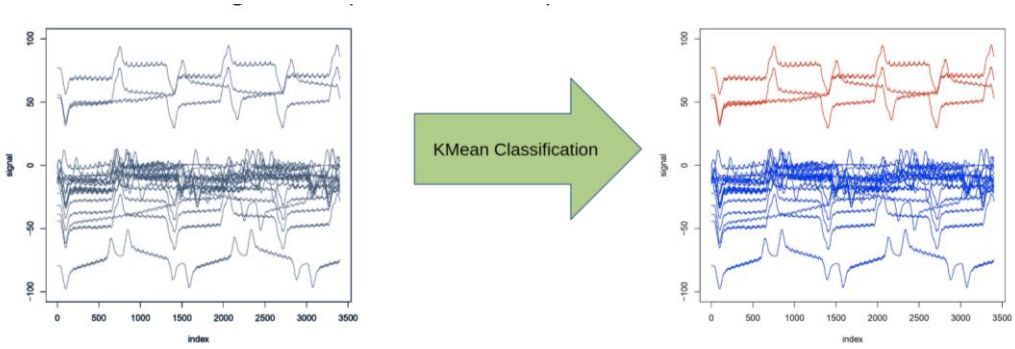


Figure 39: K-Mean classification method

3.3.3.1 Design Implementation

In the timeseries K-Means algorithm with DTW metric we also leverage LB_Keogh. LB_Keogh is a technique that provides an envelope (lower and upper bounds) around each time series sequence. It allows for a faster calculation of the distance between sequences by considering only the deviations that fall outside the envelope. This method is particularly useful when dealing with large datasets and provides us a lower computational complexity, when using the DTW algorithm, by leveraging a fast approximation approach in the algorithm level, which does not have a negative impact on the quality of results.

3.3.3.1.1 Alveo FPGA acceleration cards

Below is the general end-to-end algorithm implementation for the FPGAs.

Table 9: End-to-end K-Means implementation for the FPGAs

1. Initialization:
    a. Choose the parameters such as number of iterations or number of series.
    b. Assign the two centroids to the first two timeseries.
2. Assignment step:
    a. For each time series in the dataset, calculate the DTW distance to each cluster centroid. Accelerate this compute intensive kernel using the FPGA.
    b. Apply the LB_Keogh lower bounding technique to minimise the time of the distance calculations. LB_Keogh provides an approximate lower bound on the DTW distance, allowing for early pruning of dissimilar series.
    c. Assign each time series to the cluster with the closest centroid based on the DTW distance.
3. Update step:
    a. Recalculate the centroids of each cluster based on the assigned time series.
    b. The centroid of a cluster is the time series that minimises the sum of DTW distances to all the series assigned to that cluster.
    c. This step ensures that the cluster centroids are representative of the time series within their respective clusters.
4. Iteration:
    a. Repeat the assignment and update steps until the maximum number of iterations is reached.
5. Final Clustering:
    a. The algorithm outputs the final cluster assignments for anomaly (‘1’) or normal (‘0’) for each time series.

Below is the illustration of the algorithm. The DTW kernel is accelerated using the FPGA fabric taking advantage of the low latency on-chip storage (BRAMs) and transferring the result back to the host CPU. The algorithm continues to compute DTW distance with all the required

timeseries according to LB_Keogh values until the algorithm reaches the user-defined iteration limit.

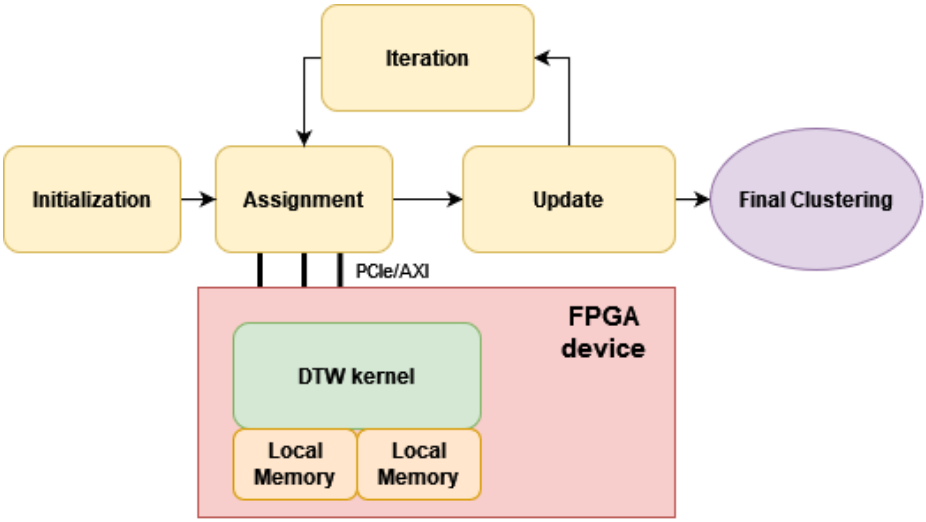


Figure 40: Illustration of timeseries K-Means for FPGAs

3.3.3.1.2 Xilinx MPSoC FPGAs

The designs created for Alveo cards can be seamlessly applied to MPSoC platforms without any modifications. The dataflow approach ensures that the utilisation of computational resources remains within the limits of the available resources in MPSoCs.

3.3.3.1.3 NVIDIA Tesla T4 GPU

For the GPU architecture we followed a different acceleration approach. In particular, we chose for acceleration the LB_Keogh kernel, as it has a high parallelization factor which is a good fit for GPUs. Taking advantage of the task level parallelization we computed the vector of all possible LB_Keogh values which are needed for the algorithm. This approach was very efficient for GPU devices and thus provided successful results in terms of throughput and latency

3.3.3.1.4 NVIDIA Jetson Orin and Nano GPUs

Similar to the server GPUs, Edge GPU devices such as Orin or Nano followed the same acceleration approach each with its own acceleration potential according to the device hardware capabilities.

3.3.3.1.5 HPC

Parallelizing K-Means Classification in an HPC environment involves applying the techniques discussed in the FFT kernel. Initially, we distribute the signals across processors and randomly assign two centroid signals in the first iteration (Figure 41). These centroids are then broadcasted to all processors, and each process computes the distance of each signal to the

random centroid locally. Subsequently, each signal is assigned to the centroid class with a closer distance, and we compute the new centroid by averaging the signals that belong to the same class. After computing the new centroid, we broadcast them to all processes in each loop iteration. We repeat this loop ten times, which gives us sufficient classification.

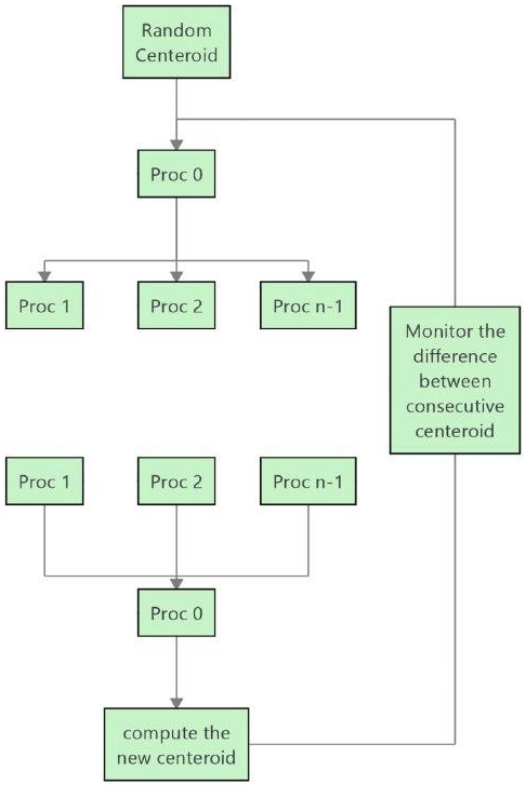


Figure 41: Parallelization of K-Means on the HPC system

3.3.3.2 Evaluation Results

3.3.3.2.1 Alveo FPGA Acceleration Cards

Below, we present the speed-up ratios for the performance and energy gains for each device for the accurate designs. All metrics were obtained using 110 timeseries as input dataset.

Figure 42 shows the execution time speed-up and the energy gains when we applied acceleration using the Alveo U200 and Alveo U50 FPGAs. Specifically, the performance speed-ups obtained are 120x and 123x and the energy gain speed-ups were 233x and 256x respectively.

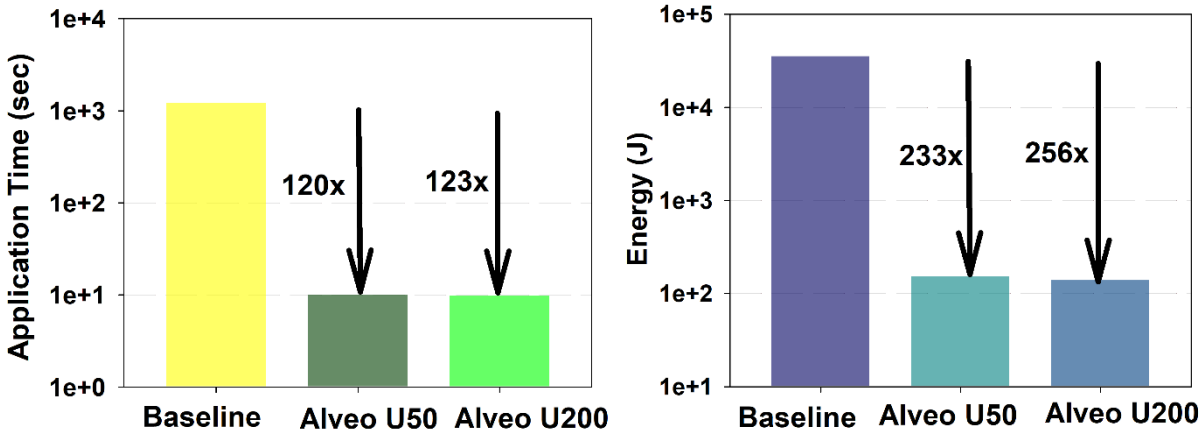


Figure 42: Latency and energy gains for K-Means on Alveo FPGAs

3.3.3.2.2 Xilinx MPSoC FPGAs

Figure 43 shows the execution time speed-up and the energy gains when we applied acceleration using the MPSoC FPGAs. Specifically, the performance speed-ups obtained is 97.4x and the energy gain speed-up is 117x.

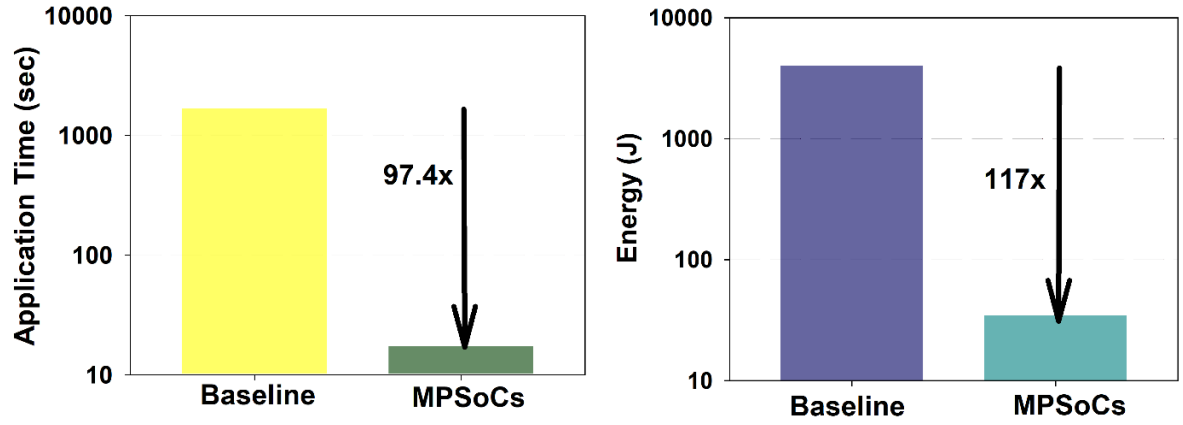


Figure 43: Latency and energy gains for K-means on MPSoC FPGAs

3.3.3.2.3 NVIDIA Tesla T4 GPU

Below, we present the results from the GPU acceleration card, specifically the Nvidia Tesla T4 GPU. Figure 44 shows the execution time speedup and the energy gains. The execution time speedup is 976.8x while the energy gain is 1061x.

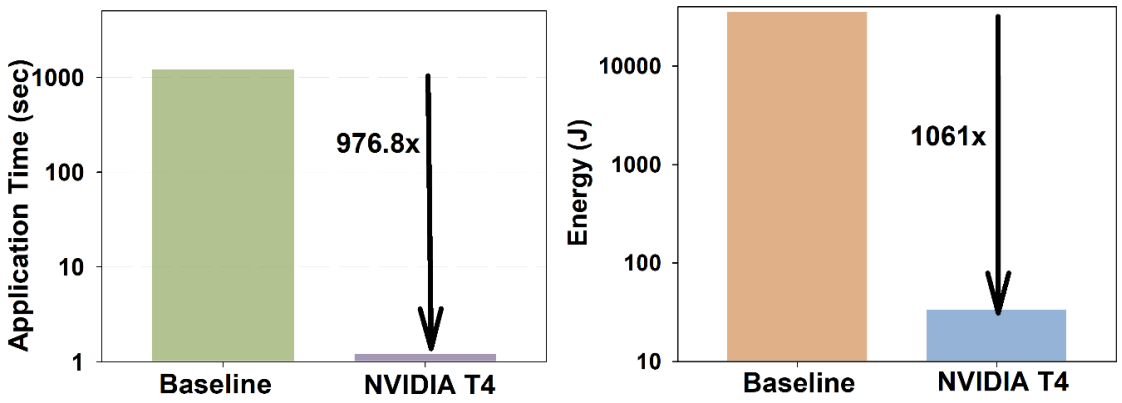


Figure 44: Latency and Energy gains for K-Means on Nvidia T4 GPU

3.3.3.2.4 NVIDIA Jetson Orin and Nano GPUs

Below, we present the results from the edge GPU devices, specifically the Nvidia Orin and Nano GPUs. Figure 45 shows the execution time speedups and the energy gains. The execution time speedups are 788x and 270x while the energy gains are 736x and 240x respectively.

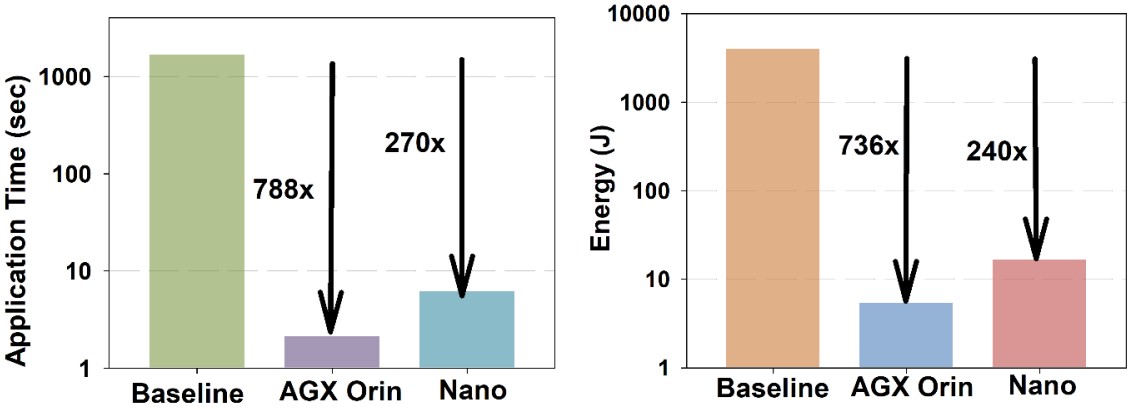


Figure 45: Latency and Energy gains for K-Means on Orin and Nano GPU devices.

3.3.3.2.5 HPC

HLRS tested parallelization techniques for K-Means using several position signal batches provided by IDEKO. Table 10 shows the achieved K-Means speedup, energy gain, and minimum execution time by running K-Means on compute nodes at Hawk supercomputer in HLRS for 110, 330, 550, ..., and 1100 position signals.

Table 10: Speedup and energy gain of K-means on HPC system

IDEKO Signal	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
110 position signal	71X	53X	100%	10.9545	4416.55

330 position signal	199X	51X	98%-100%	7.59292	5145.27
550 position signal	311X	48X	100%	8.21393	9276.83
770 position signal	449X	52X	100%	8.19523	12958
990 position signal	558X	55X	100%	8.05906	15031.3
1100 position signal	368X	32X	100%	13.1062	27405.4

IDEKO has generated acceleration data by using accelerometers at different parts on the screw balls. As a result, each signal has six coordinates, meaning that six signals must be considered as one signal. The K-means classification method also applied to these signal formats. Therefore, we have developed a new parallel implementation of K-means for this purpose, using the same parallelization strategy as applied for the K-means in position signals.

HLRS have performed parallel K-Means on the acceleration data using several data batches provided by IDEKO, that the speedup and minimum execution time are presented in Table 11.

Table 11: Speedup and energy gain of K-means for acceleration data on HPC system

IDEKO Signal	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
26 cycle signal	82X	34X	90%- 100%	24.007	10067.4
104 cycle signal	433X	74X	100%	25.217	29466.8
156 cycle signal	634X	69X	100%	25.743	47280.2
208 cycle signal	518X	44X	100%	42.08	98357.6
234 cycle signal	566X	44X	100%	43.354	112164
260 cycle signal	580X	38X	100%	47.035	141346

3.3.4 KNN Clustering Algorithm

The K-Nearest Neighbor [15] algorithm is a supervised machine learning technique used for classification, with applications in image processing and generative models. In the context of this study, KNN is employed for the classification of time series signals.

The fundamental idea behind KNN is to classify time series signals based on labelled training datasets. Figure 46 illustrates the training dataset, which consists of labelled signals represented by red and blue signals, as well as inference signals represented by green signals. The KNN method for time series signals calculates the distance between the inference signals and the training signals using the DTW metric and identifies the K-Nearest neighbors. The class label of the inference signal is determined by the majority label of these K-Nearest neighbors.

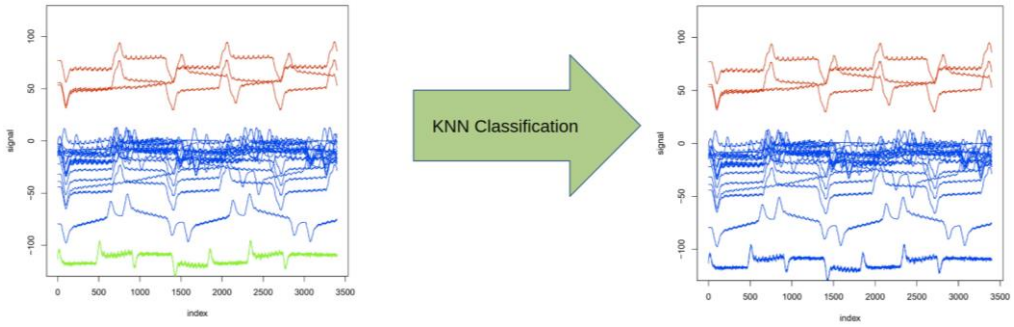


Figure 46: The inference signal, represented by green signals, is classified into the blue group using the KNN classification method, where the training signals are labelled as blue and red

3.3.4.1 Design Implementation

In the Timeseries KNN algorithm with DTW metric we also leverage LB_Keogh. It uses the concept of envelope to approximate the upper and lower bounds of the time series. The envelope is computed by taking a window around each point in the time series and constructing an envelope that encloses the potential range of values within the window. LB_Keogh helps reduce the number of distance calculations required by DTW and thus improves its efficiency.

3.3.4.1.1 Alveo FPGA Acceleration Cards

Our acceleration strategy for the FPGAs is as follows.

Table 12: KNN acceleration strategy for the FPGA

1. Compute LB_Keogh Envelopes: For each time series in the training set, compute the LB_Keogh envelope.
2. Query Processing: Given a query time series, compute its LB_Keogh envelope.
3. Distance Calculation: Calculate the DTW distance between the query time series and each time series in the training set, using the LB_Keogh envelope as an upper bound. If the DTW distance exceeds a threshold defined by the LB_Keogh envelope, the time series can be pruned from further consideration.

kNN Classification: Select the K-nearest neighbors based on the smallest DTW distances and use their labels to classify the query time series.

The illustration of the high-level algorithm is presented below:

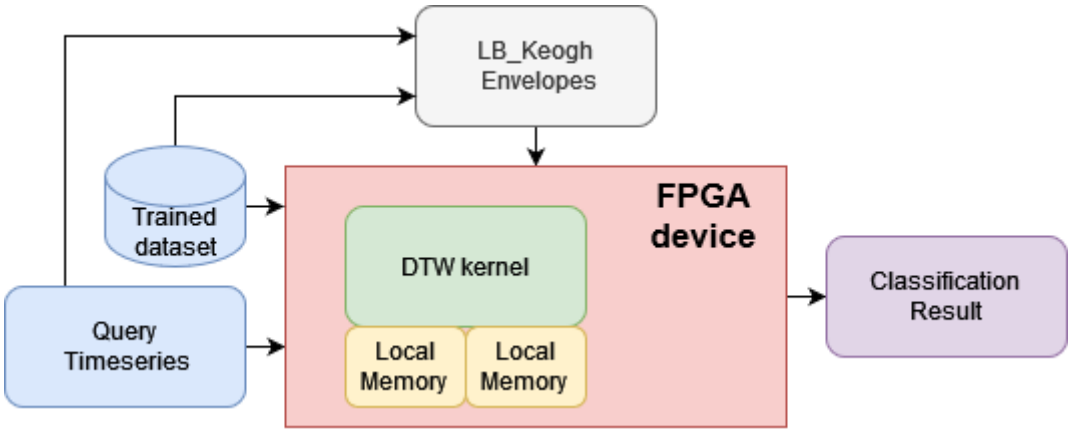


Figure 47: Illustration of TimeSeries KNN for FPGAs

3.3.4.1.2 Xilinx MPSoC FPGAs

The designs created for Alveo cards can be seamlessly applied to MPSoC platforms without any modifications. The dataflow approach ensures that the utilisation of computational resources remains within the limits of the available resources in MPSoCs.

3.3.4.1.3 NVIDIA Tesla T4 GPU

For the GPU architecture, we employed a different method to enhance its performance. Specifically, we opted to utilise the LB_Keogh kernel for acceleration due to its high degree of parallelization, which aligns well with GPUs. To leverage task-level parallelization, we calculated the vector of all possible LB_Keogh values required for the algorithm. This approach proved highly efficient on GPU devices, delivering successful outcomes in terms of throughput and latency.

More specifically, we employed GPU shared memory to establish a shared region of memory accessible to threads within a block. Within this memory block, we stored the input series used for computing LB_Keogh values. Additionally, we utilised shared memory to retain intermediate sum values of LB_Keogh. By employing a technique known as reduced sum, we synchronised and computed the sum across the threads in each block. This synchronisation was crucial to ensure proper data sharing and prevent race conditions. In essence, the GPU carried out simultaneous calculations on multiple data points, taking advantage of the extensive parallel architecture of the GPU. Consequently, this significantly accelerated the processing compared to sequential execution on a CPU.

The illustration of the high-level algorithm is presented below:

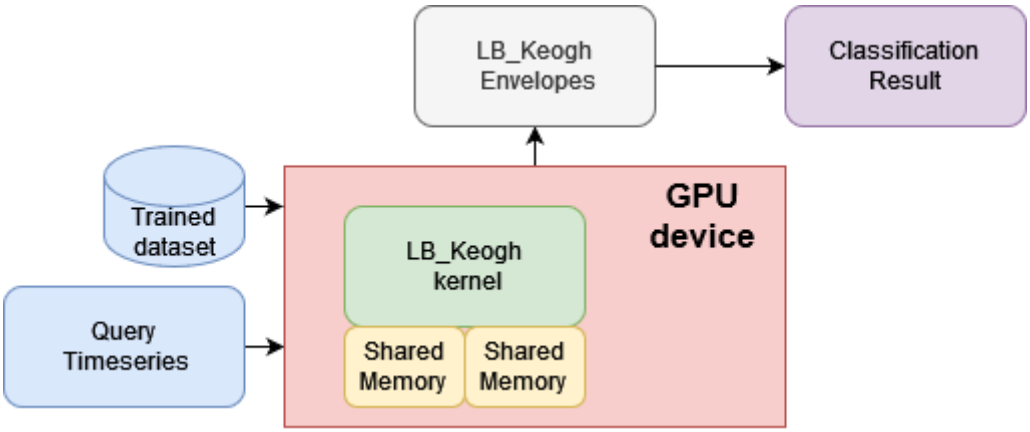


Figure 48: Illustration of TimeSeries KNN for GPUs

3.3.4.1.4 NVIDIA Jetson Orin and Nano GPUs

Similar to the server GPUs, Edge GPU devices such as Orin or Nano followed the same acceleration approach each with its own acceleration potential according to the device hardware capabilities.

3.3.4.1.5 HPC

To speed up the KNN algorithm, the training datasets were first distributed uniformly into processes. Next, the inference signal was broadcasted to all processes, as shown in Figure 49, and the distance between the inference signal and the training signals was computed locally within each process. These distances were then gathered into process 0. Finally, the class label of the inference signal was determined based on the majority label of the K-nearest neighbors.

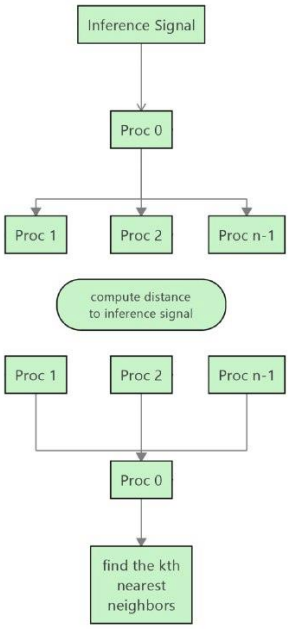


Figure 49: Parallelization of the KNN on the HPC system

3.3.4.2 Evaluation Results

3.3.4.2.1 Alveo FPGA Acceleration Cards

Below, we present the speed-up ratios for the performance and energy gains for each device for the accurate and approximate designs. All metrics were obtained using 110 timeseries as trained dataset and single timeseries for the query.

Figure 50 shows the execution time speed-up and the energy gains when we applied acceleration using the Alveo U200 and Alveo U50 FPGAs. Specifically, the performance speed-ups obtained are 52.8x and 65x and the energy gain speed-ups were 114.3x and 150.7x respectively.

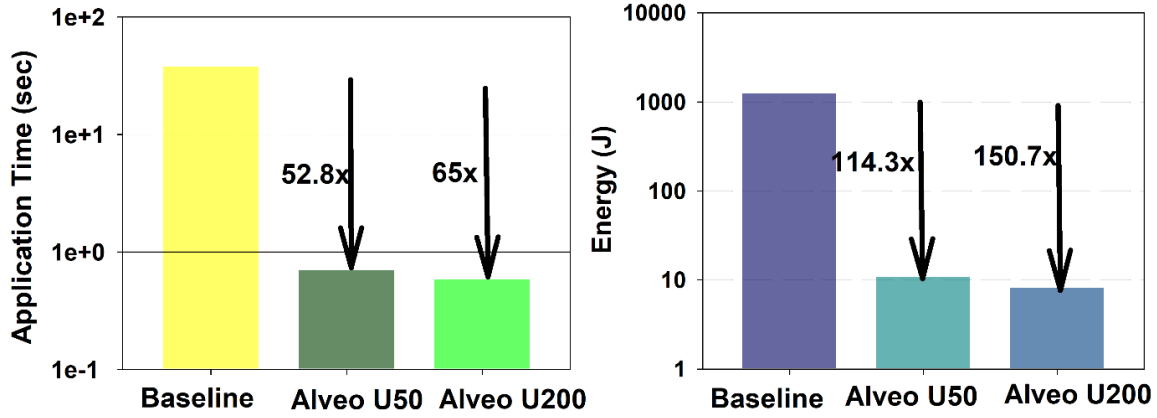


Figure 50: Latency and Energy gains for K-NN on Alveo FPGAs

3.3.4.2.2 Xilinx MPSoC FPGAs

Figure 51 shows the execution time speed-up and the energy gains when we applied acceleration using the MPSoC FPGAs. Specifically, the performance speed-up obtained is 123.6x and the energy gain speed-up is 61.1x.

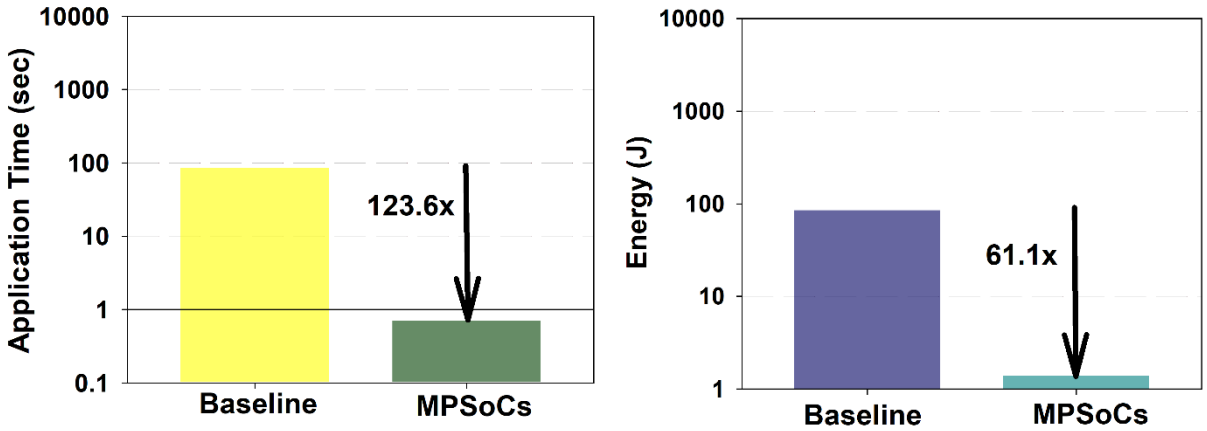


Figure 51: Latency and Energy gains for K-NN on MPSoC devices.

3.3.4.2.3 NVIDIA Tesla T4 GPU

Below, we present the results from the edge GPU devices, specifically the Nvidia T4 GPU. Figure 52 shows the execution time speedup and the energy gains. The execution time speedups are 164.3x and while the energy gains are 198x respectively.

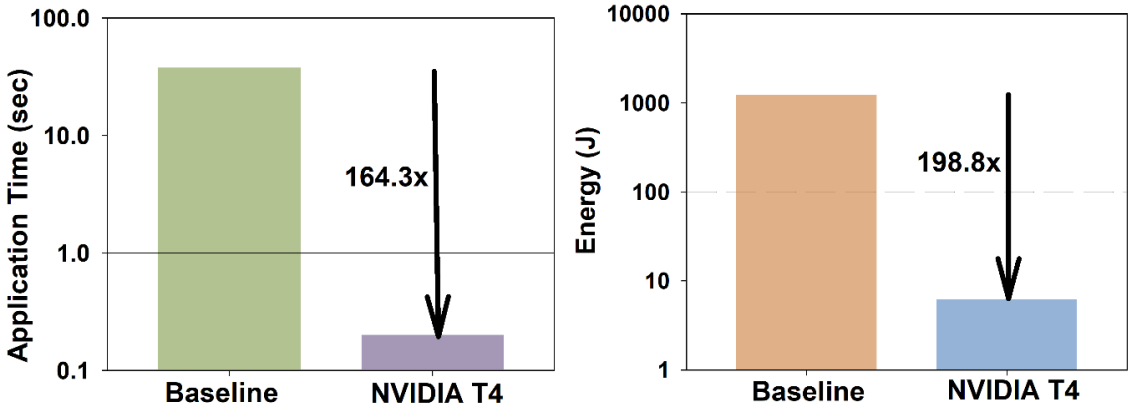


Figure 52: Latency and Energy gains for K-NN on T4 GPU device.

3.3.4.2.4 NVIDIA Jetson Orin and Nano GPUs

Below, we present the results from the edge GPU devices, specifically the Nvidia Orin and Nano GPUs. Figure 53 shows the execution time speedup and the energy gains. The execution time speedups are 417x and 383x while the energy gains are 189x and 145x respectively,

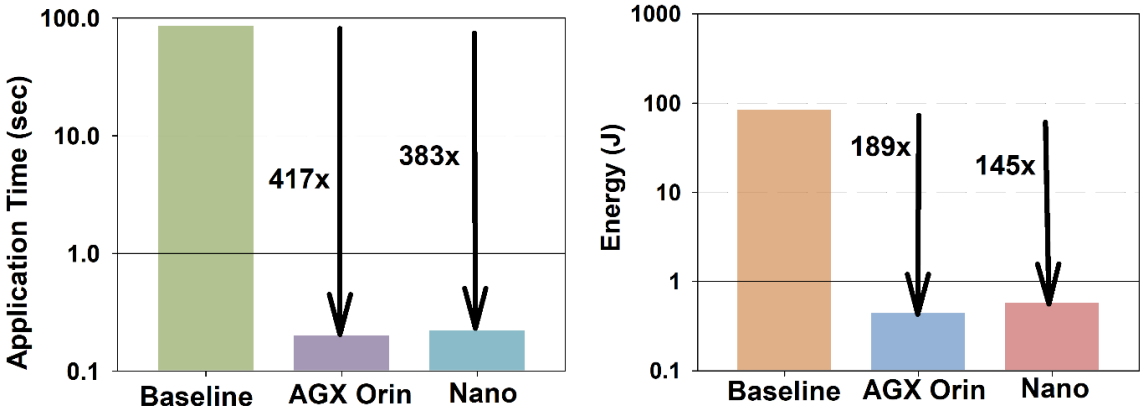


Figure 53: Latency and Energy gains for K-NN on Nvidia Orin and Nano GPU devices.

3.3.4.2.5 HPC

HLRS tested the parallelization of the KNN kernel using multiple training datasets and inference data provided by IDEKO. The training datasets consisted of time series signals with associated labels, and KNN was used to classify the inference signals. The results of our tests, including the speedup and minimum execution time, are presented in Table 13.

Table 13: Speedup and energy gain of KNN on the HPC system

IDEKO Signal	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
110 training signal	61X	46X	100%	0.42	96.4233
330 training signal	157X	39X	100%	0.49	334.166
550 training signal	257X	39X	100%	0.51	580.295
770 training signal	351X	38X	100%	0.52	830.691
990 training signal	418X	40X	100%	0.57	1079.75
1100 training signal	374X	33X	100%	0.71	1485.88

IDEKO supplied us with acceleration data consisting of six signals that were combined into one signal. We applied K-Means to this data and utilised their labels to categorise the acceleration inference signals. We also implemented a new KNN for acceleration data and tested parallelization with training signals in Table 14.

Table 14: Speedup and energy gain of KNN for acceleration data on the HPC system

IDEKO Signal	Speedup	Energy gain	Accuracy %	Minimum execution time (sec)	Minimum energy consumption (Joule)
26 cycle signal	87X	36X	100%	1.25	525.05
104 cycle signal	261X	45X	100%	1.87	2153.36
156 cycle signal	514X	57X	100%	1.37	2532.12
208 cycle signal	516X	44X	100%	1.86	4361.16
234 cycle signal	505X	41X	100%	2.03	5271.06
260 cycle signal	546X	38X	100%	2.09	6280.81

Figure 54 below summarises the results for all the FPGA and GPU designs for the UC3.

Application	UC	Platform	Device	Latency (ms)	BRAM	DSP	FF	LUT	Grid size	Block size	Power (W)	Energy gain	Speedup
DBSCAN	IDK	FPGA	U50	7581	7.2%	0.6%	0%	4.2%	–	–	27.3	152.9x	343x
DBSCAN	IDK	FPGA	U200	26214	2.2%	0.2%	1.6%	1.5%	–	–	46	26.2x	99.2x
DBSCAN	IDK	FPGA	ZCU104	158903	10%	2%	2%	6%	–	–	2.1	42.5x	17.8x
DBSCAN	IDK	FPGA	ZCU102	207430	2%	2%	2%	4%	–	–	1.9	36x	13.6x
KMEANS	IDK	FPGA	U50	10080	1%	0.08%	0.4%	0.6%	–	–	15.1	233x	120x
KMEANS	IDK	FPGA	U200	9841	0.6%	0.07%	0.3%	0.4%	–	–	14.1	256x	123x
KMEANS	IDK	FPGA	ZCU104	17144	4.1%	0.29%	1.4%	1.9%	–	–	2	117x	97x
KMEANS	IDK	FPGA	ZCU102	17144	2.6%	0.2%	1.1%	1.65%	–	–	2%	117x	97x
KMEANS	IDK	GPU	Tesla T4	1240	–	–	–	–	128	64	27	1061x	976x
KMEANS	IDK	GPU	AGX Orin	2117	–	–	–	–	128	64	2.6	736x	788x
KMEANS	IDK	GPU	Jetson Nano	6182	–	–	–	–	128	64	2.7	240X	270x
KNN	IDK	FPGA	U50	715	1%	0.08%	0.5%	0.6%	–	–	15.1	114x	52.8x
KNN	IDK	FPGA	U200	581	0.6%	0.07%	0.3%	0.5%	–	–	15	150x	65x
KNN	IDK	FPGA	ZCU104	695	4.1%	0.29%	1.4%	1.9%	–	–	2	61x	123x
KNN	IDK	FPGA	ZCU102	695	2.6%	0.2%	1.1%	1.6%	–	–	2	61x	123x
KNN	IDK	GPU	Tesla T4	230	–	–	–	–	128	64	27	198x	164x
KNN	IDK	GPU	AGX Orin	206	–	–	–	–	128	64	2.1	189x	417x
KNN	IDK	GPU	Jetson Nano	224	–	–	–	–	128	64	2.1	145X	383x
FFT	IDK	FPGA	U50	494	46%	3.5%	0%	6.5%	–	–	27.8	50x	38.2x
FFT	IDK	FPGA	U200	390	28.9%	3%	5.1%	4.8%	–	–	50.6	34.8x	34.8x
FFT	IDK	FPGA	ZCU104	1164	95%	2%	12%	20%	–	–	2.8	30.4x	18.9x
FFT	IDK	FPGA	ZCU102	3052	32%	2%	10%	18%	–	–	2.9	11.2x	7.2x

Figure 54: UC3 FPGA and GPU designs

4 Performance Maximization Under Maximum Affordable Error for HW and SW IPs

Section 2 outlined the acceleration of kernels in the FPGA, GPU and HPC system, which was required by use case providers. However, these computations demand vast compute and memory resources. In Task 4.2 of SERRANO, we were required to tackle these challenges by employing transprecision and approximation computing techniques [16] at the cost of deteriorating accuracy. In this section, we will explain how these techniques function and how we applied them in developing kernels for the FPGA, GPU and HPC system. By using these techniques, we will demonstrate the extent to which execution runtime and energy consumption have improved.

For the design of approximate accelerators on FPGAs, two versions are developed per accelerator and platform, each offering a different level of approximation. One version aims for a low approximation error, resulting in a smaller trade-off between error and energy gain, while the other version intentionally introduces a higher approximation error. It should be noted that the quantification of error metrics depends on the input dataset. The specific values for low and high approximation errors are calculated based on the provided by the use cases datasets for each algorithm.

There are numerous approximate computing methods applied to hardware platforms to meet the requirements of critical embedded applications with ultra-low power consumption and small footprint. AUTH has mainly used the techniques of **a) precision scaling**, which aims to reduce the precision of operands, **b) approximate memoization**, which is used to approximate operations that require many clock cycles on hardware, such as the logarithm function, and **c) loop perforation**, which skips loop iterations that incur significant overhead. For more information, refer to Deliverable D4.2. Finally, application-specific approximations were used based on the algorithmic characteristics of the application under test.

While using these techniques introduces several parameters and uncertainties, a Verification, Validation and Uncertainty Quantification (VVUQ) framework was developed to quantify these uncertainties. This framework helps to choose parameters, such as the number of processes, data precision and computation density to minimise execution time and energy consumption, while balancing the accuracy of the kernel and execution time trade-off. Moreover, the Gradient Descent method is used to develop a non-linear formula estimating the execution time and energy consumption of the HPC service for different data batches.

4.1 Approximation of the Fintech Analysis (UC2, InBestMe) Algorithms

4.1.1 Savitzky-Golay (SAVGOL) Filter

4.1.1.1 Approximate and Transpresicion Techniques

Approximate versions of the Savitzky-Golay filter FPGA and HPC accelerators have been developed.

4.1.1.1.1 Alveo FPGA Acceleration Cards

The Alveo U50 and U200 FPGA acceleration cards utilise approximate designs that leverage HLS arbitrary precision data types to quantize the inputs and intermediate operands. In the low approximation error variant, the floating-point values are converted to the ***ap_fixed<15, 12, AP_RND_CONV>*** data type. This fixed-point representation allocates 15 bits for the integral part and 3 bits for the fractional part. Additionally, a rounding circuit is implemented. Similarly, the high approximation error variant employs the ***ap_fixed<14, 12, AP_RND_CONV>*** data type, reducing the fractional part to 2 bits.

4.1.1.1.2 Xilinx MPSoC FPGAs

The quantization schemes that were used in the Alveo designs were also employed for the design of the approximate accelerators in the MPSoC platforms.

4.1.1.1.3 HPC

Transprecision techniques are an approach used in computation that aims to optimise the use of memory resources. By applying lower data precision in the computation, this approach reduces the memory footprint and minimises the execution run time and energy consumption. Despite the use of lower precision data, the accuracy of the final result does not necessarily have to be compromised.

There are several transprecision computing techniques that have been developed. We apply the Mixed-precision computing [17]: This technique involves using different precision levels within a single computation. To implement this technique, we use templated data type, allowing us to template all the input and output data involved in the implementation. As illustrated in Figure 55, we use two distinct data types for input and output data, with 'I' representing input data precision and 'O' representing output data precision. As a result, we are able to change the data type inside the implementation dynamically.

```
template<typename I, typename O>
void
```

Figure 55: Template data type for transprecision techniques

Approximation computing techniques are used to improve the performance and reduce the execution time of computations. One well-established approach is loop perforation, which involves skipping some iterations in repetitive loops. This reduces the workload and results in a significant reduction in execution time. However, this approach can also lead to changes in the quality and accuracy of the output compared to the original code. It is a straightforward method used when the computation can be reduced, and some degree of error can be tolerated.

There are several ways to implement loop perforation. One common approach is to use a technique called "step skipping." Step skipping involves skipping some of the loop iterations by incrementing the loop counter by a larger amount than one. For example, if a loop iterates over an array of values, step skipping may involve only processing every third or fourth element in the array instead of processing every single element. Figure 56 demonstrates the application of loop perforation, which transforms the canonical loop into a tuned loop. The perforation stride 's' is used to indicate how many iterations of the original loop are skipped.

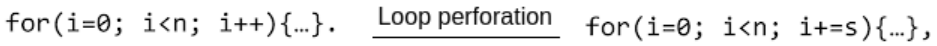


Figure 56: Loop perforation in approximation computing techniques

4.1.1.2 Evaluation Results

4.1.1.2.1 Alveo FPGA Acceleration Cards

Figure 57 shows the execution time speedup and the energy gains when the accelerators with the low approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 2.1x and 2.6x for the U50 and U200, while the energy gains are 2.4x and 1.3x respectively. The Mean Absolute Error (MAE) is 0.17.

Figure 58 shows the execution time speedup and the energy gains when the accelerators with the high approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 2.1x and 2.7x for the U50 and U200, while the energy gains are 2.6x and 1.6x respectively. The Mean Absolute Error (MAE) is 6.4.

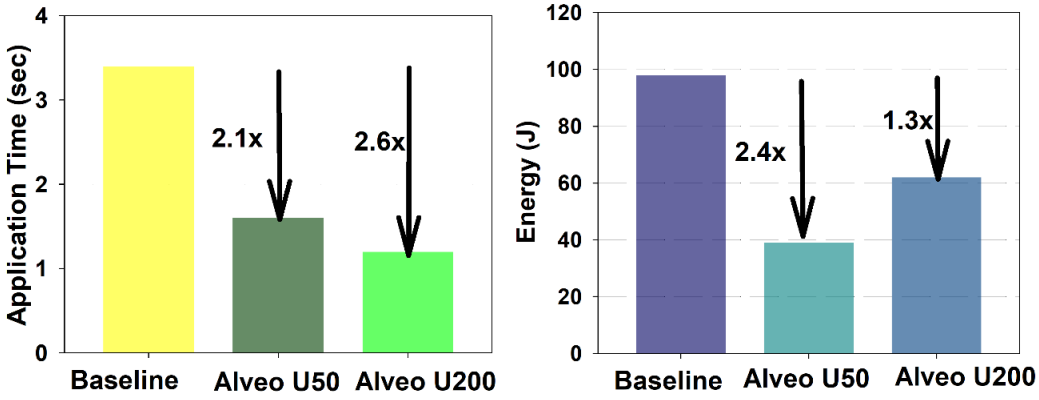


Figure 57: Latency and energy gains of the low approximate SAVGOL on the Alveo FPGAs

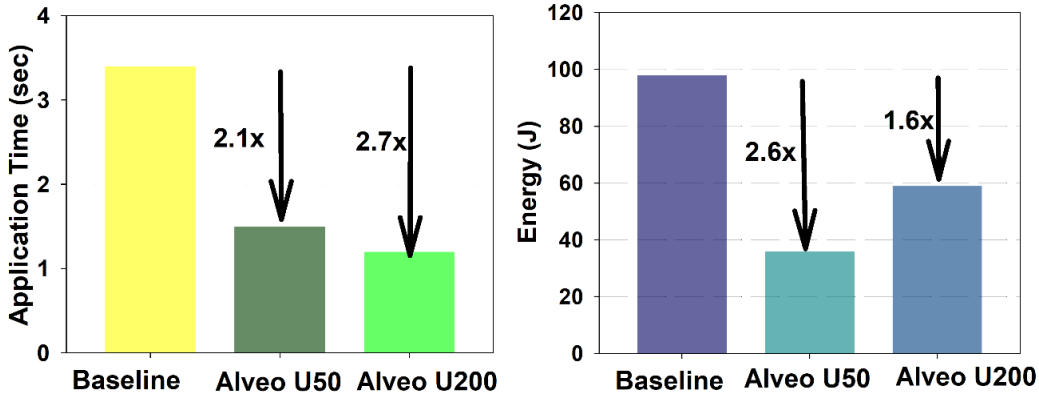


Figure 58: Latency and energy gains of the high approximate SAVGOL on the Alveo FPGAs

4.1.1.2.2 Xilinx MPSoC FPGAs

Figure 59 shows the execution time speedup and the energy gains when the accelerators with the low-approximation error are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 2x and 2.2x for the ZCU104 and ZCU102, while the energy gains are 2.7x and 3x respectively. The Mean Absolute Error (MAE) is 0.17.

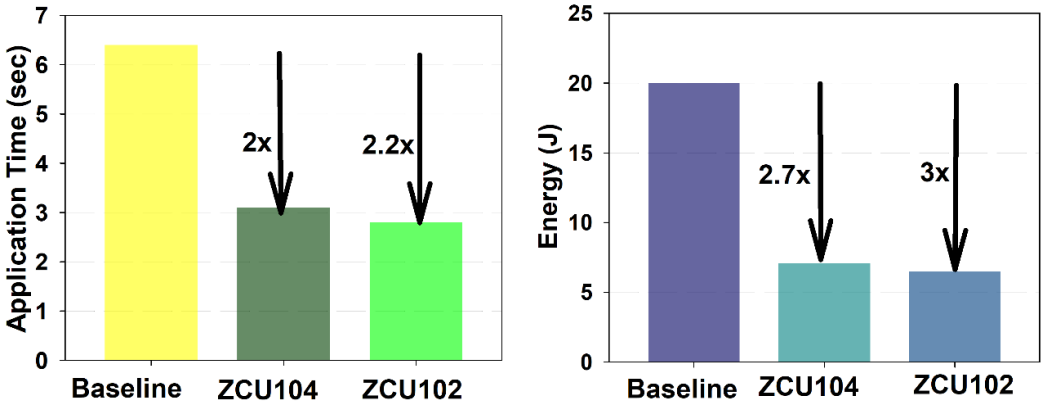


Figure 59: Latency and energy gains of the low approximate SAVGOL on the MPSoC FPGAs

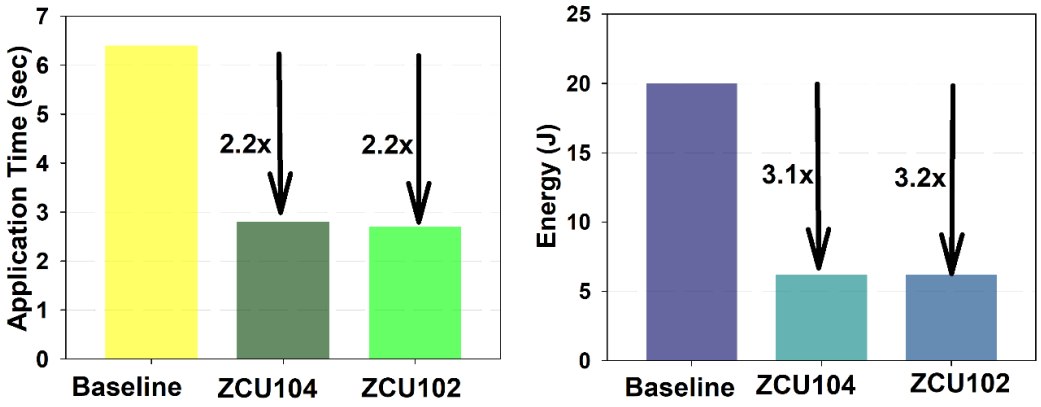


Figure 60: Latency and energy gains of the high approximate SAVGOL on the MPSoC FPGAs

Figure 60 shows the execution time speedup and the energy gains when the accelerators with the high-approximation error are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 2.2x for the ZCU104 and ZCU102, while the energy gains are 3.1x and 3.2x respectively. The Mean Absolute Error (MAE) is 6.4.

4.1.1.2.3 HPC

Transprecision techniques will be applied into the Savitzkey-Golay kernel. By employing two different data types: double precision and single precision (float), for both input and output data types, denoted as I and O in Figure 55, respectively. Table 15 presents how combining these different data types can potentially improve the execution runtime and reduce energy consumption.

Table 15: Transprecision techniques in the Savitzkey-Golay

Input data precision	Output data precision	Execution time (sec)	Energy consumption (Joule)	Execution time improvement	Energy consumption improvement
double	double	0.1169	109.358	-	-
double	float	0.0742	69.401	36%	36%
float	double	0.0999	93.499	14%	15%
float	float	0.0429	40.153	63%	64%

Loop perforation as approximation computing techniques has been utilised in Savitzky-Golay implementation. It is important to note that loop perforation cannot be applied to critical loops in kernels, as it would lead to execution errors. Table 16 showcases the results of loop perforation in the Savitzky-Golay with input data from InBestMe, where different perforation strides {1, 2, 4, 8} were used.

Table 16: Approximation techniques in the Savitzkey-Golay

Perforation strides	Execution time	Energy consumption (Joule)	L2 error norm	Execution time improvement	Energy consumption improvement	Error increment
1	0.0025	0.0427	91.605	-	-	-
2	0.0021	0.0367	113.68	16%	14%	24%
4	0.0020	0.0338	123.25	20%	21%	34%
8	0.0018	0.0328	127.87	28%	23%	38%

4.1.2 Kalman Filter

4.1.2.1 Approximate and Transpresicion Techniques

We provide two approximate versions for each FPGA-accelerated version of the Kalman filter, targeting the different devices available on the SERRANO platform.

4.1.2.1.1 Alveo FPGA Acceleration Cards

Our approximation strategy is based on the implementation of the batched Kalman filter described in Deliverable D4.1. In particular, we divide each signal into k equal parts and perform Kalman filtering for each of these parts. This source transformation makes it possible to break the dependency and to calculate the Kalman filter for each partial signal in parallel. As we show in D4.1, the error caused by this transformation is negligible due to the error tolerance of the Kalman filter. The number of batches is set at 10 for the low approximation variant and 50 for the high approximation variant for the Alveo U50. For the Alveo U200 FPGA, the batch size for both variants is set to 50, since timing problems occurred when synthesising the low-approximate version with a batch size of 10.

We further approximate the batched Kalman filter using the precision scaling approximation technique described in Deliverable D4.2. In the variant with small approximation error, the floating-point values are converted to the data type [18] `ap_fixed<19, 13, AP_RND>`. This fixed-point representation provides 13 bits for the integer part and 6 bits for the decimal part. For the variant with high approximation error the data type `ap_fixed<14, 13, AP_RND>` is used, which reduces the decimal part to 1 bit. These data types are used for the approximation variants of the two devices.

4.1.2.1.2 Xilinx MPSoC FPGAs

The approximation schemes used in the Alveo designs were also used to design the approximate accelerators in the MPSoC platforms.

4.1.2.1.3 HPC

The transprecision techniques applied in the Kalman filter are similar to the approach used in the Saitzkey-Golay filter. Different precisions, particularly lower precision, can be employed by utilising templated data types for input and output data. This leads to a reduction in execution time while maintaining the accuracy of the final output.

Also, approximation techniques such as loop perforation, as explained in the Saitzkey-Golay filter section, can be utilised in the Kalman filter. Some of the loop iterations within the Kalman filter kernel can be skipped to reduce the workload and decrease execution time, however, at the expense of altering the accuracy of the final results.

4.1.2.2 Evaluation Results

4.1.2.2.1 Alveo FPGA Acceleration Cards

Figure 61 shows the speedup and energy gains of the low approximation error version of the Kalman filter for the Alveo U50 and U200 FPGAs compared to the Python single-threaded execution. The speedups are 5064x and 4352x for the U50 and U200, while the energy gains are 7719x and 3899x, respectively. The low approximation error version can achieve 1.65x and 2.3x higher speedups and 1.5x and 2.1x higher energy gains compared to the accurate Alveo U50 and U200 FPGAs versions. These benefits are associated with negligible errors, as the average Mean Absolute Error (MAE) for the 4890 stock price signals is about 0.05.

Figure 62 shows the corresponding results for the version of the Kalman filter with high approximation error. The speedups are 5586x and 4468x for the U50 and U200, while the energy gains are 7801x and 4100x, respectively. The high approximation error version can achieve 1.82x and 2.4x higher speedups and 1.55x and 2.2x higher energy gains compared to the accurate Alveo U50 and U200 FPGAs versions. These benefits are associated with an average MAE of 1.15.

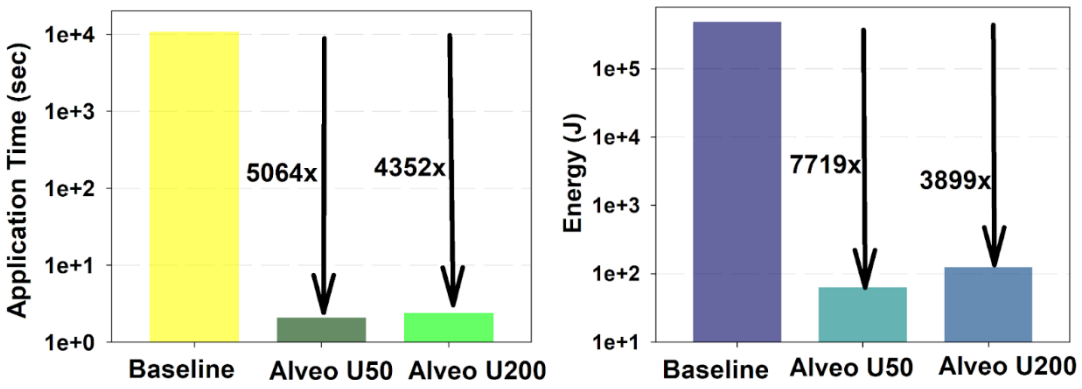


Figure 61: Latency and energy gains of the low approximate Kalman on the Alveo FPGAs

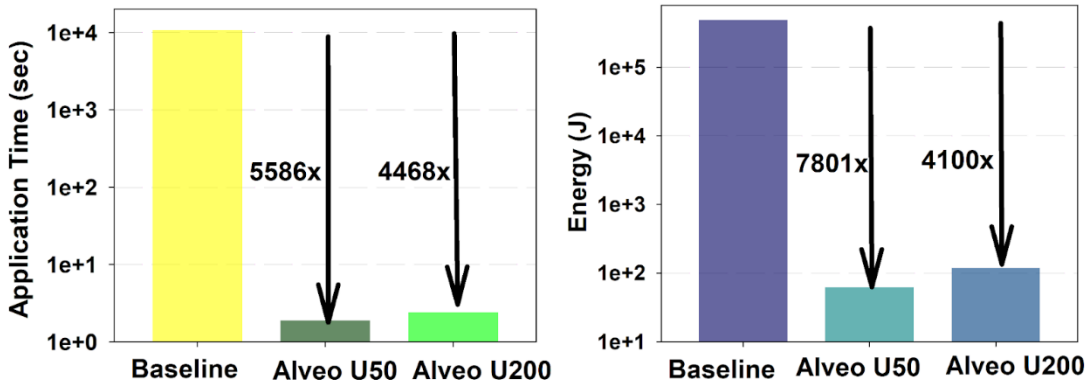


Figure 62: Latency and energy gains of the high approximate Kalman on the Alveo FPGAs

4.1.2.2.2 Xilinx MPSoC FPGAs

Figure 63 shows the speedup and energy gains of the low approximation error version of the Kalman filter for the MPSoC ZCU104 and ZCU102 FPGAs compared to the Python single-threaded execution. The speedups are 2707x and 1170x for the ZCU104 and ZCU102, while the energy gains are 4948x and 2139x, respectively. The low approximation error version can achieve 1.79x and 1.35x higher speedups and 1.7x and 1.2x higher energy gains compared to the accurate versions for the MPSoC ZCU104 and ZCU102 FPGAs. As in the case of cloud devices, the average MAE is 0.05.

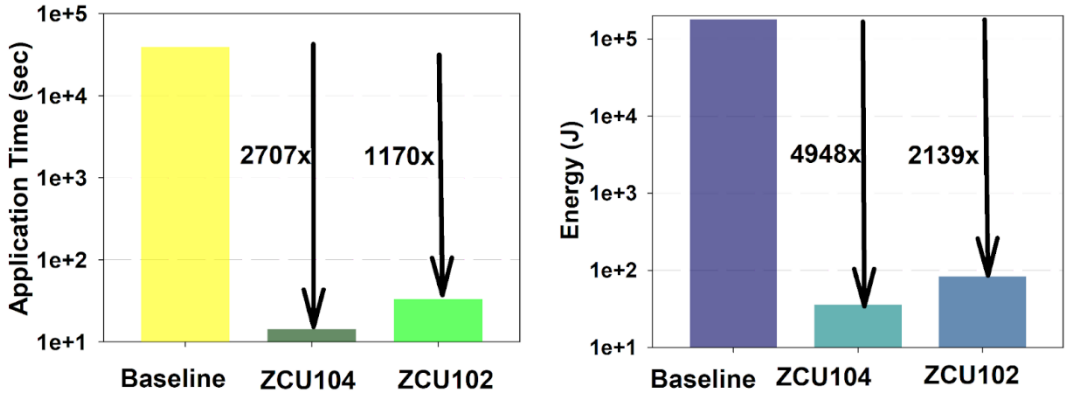


Figure 63: Latency and energy gains of the low approximate Kalman on the MPSoC FPGAs

Figure 64 shows the corresponding results for the version of the Kalman filter with a high approximation error. The speedups are 2905x and 1510x for the ZCU104 and ZCU102, while the energy gains are 4023x and 2380x, respectively. The high approximation error version can achieve 1.9x and 1.75x higher speedups and 1.4x and 1.3x higher energy gains compared to the accurate versions for the MPSoC ZCU104 and ZCU102 FPGAs. An interesting observation is that for the MPSoC ZCU104, the energy gains are higher for the low-approximation version, which is counterintuitive. This is due to the increase in batch size and thus parallelism, which translates into higher DSP and LUT utilisation. The higher use of DSPs and LUTs is responsible

for the 1.3x increase in power consumption that accounts for the lower energy gain in the high approximation error design. Similar to cloud devices, the average MAE is 1.15.

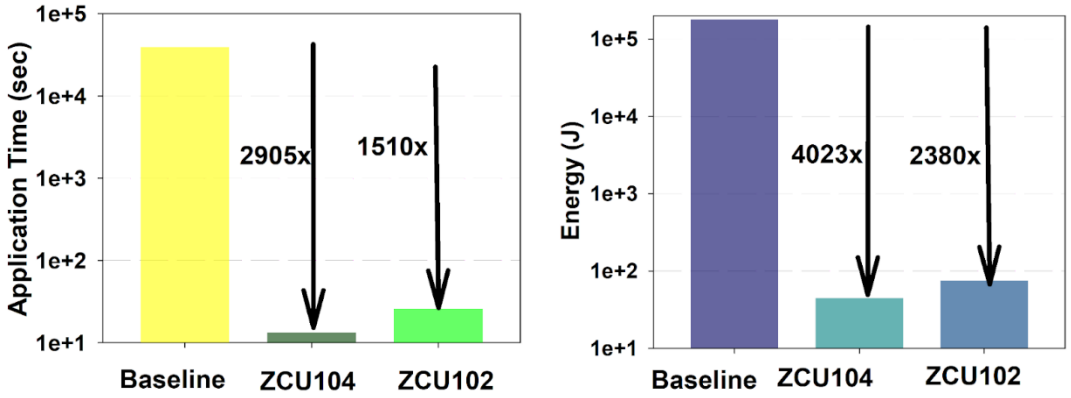


Figure 64: Latency and energy gains of the high approximate Kalman on the MPSoC FPGAs

4.1.2.2.3 HPC

To assess the impact of transprecision techniques on the Kalman filter, two different data types were used: double precision and single precision for both input and output data. Table 17 shows the execution time and energy improvements achieved by implementing mixed data precision in the Kalman filter.

Table 17: Transprecision techniques in Kalman filter

Input data precision	Output data precision	Execution time (sec)	Energy Consumption (Joule)	Execution time improvement	Energy consumption improvement
double	double	0.1782	166.7	-	-
double	float	0.1007	94.23	43%	44 %
float	double	0.0761	71.25	57%	58%
float	float	0.0135	12.63	92%	94%

Table 18 illustrates the impact of loop perforation on the Kalman filter. Different perforation strides, namely {1, 2, 4, 8}, were applied. As a result, the execution time and energy consumption of the kernel were reduced. However, it should be noted that the error norm increased, indicating a loss of accuracy in the final results.

Table 18: Approximation techniques in Kalman Filter

Perforation strides	Execution time	Energy Consumption (Joule)	L2 error norm	Execution time improvement	Energy consumption improvement	Error increment
1	0.00043	0.0733	97.097	-	-	-
2	0.00031	0.0531	115.987	28%	27%	18%
4	0.00024	0.0419	124.344	44%	41%	27%
8	0.00021	0.0366	128.432	50%	51%	31%

4.1.3 Wavelet Filter

4.1.3.1 Approximate and Transpresicion Techniques

Approximate versions of the wavelet filter accelerators have been developed for the FPGA platforms and the HPC resources.

4.1.3.1.1 Alveo FPGA Acceleration Cards

The approximate versions of the wavelet kernel implemented on the Alveo U50 and U200 platforms utilise different data types, resulting in variations in error, resource utilisation, power consumption, and performance metrics.

In the accurate version of the wavelet filter, the double data type is employed for both the filter's input signal and the wavelet coefficients. In contrast, the low-approximate version uses the float data type, which leads to the utilisation of fewer resources.

For the high-approximate version, the accelerator's inputs undergo quantization, similar to the quantization method used for the approximate Savitzky-Golay accelerators. In this quantization process, the HLS ***ap_fixed<19,16>*** data type is utilised, allocating 19 bits for number representation. Out of the 19 bits, only 3 are dedicated to the fractional part of the numbers.

4.1.3.1.2 Xilinx MPSoC FPGAs

The approximation approach that is described in the sub-section above was followed for the design of the MPSoC approximate accelerators. The same data types with the ones used on the Alveo designs were used.

4.1.3.2 Evaluation Results

4.1.3.2.1 Alveo FPGA Acceleration Cards

Figure 65 shows the execution time speedup and the energy gains when the accelerators with the low approximation error are executed on the Alveo Xilinx acceleration cards. The

execution time speedups are 3.8x and 4.3x for the U50 and U200, while the energy gains are 4.8x and 2.9x respectively. The Mean Absolute Error (MAE) is 0.01.

Figure 66 shows the execution time speedup and the energy gains when the accelerators with the high approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 3.8x and 4x for the U50 and U200, while the energy gains are 4.9x and 2.9x, respectively. The MAE is 22.2.

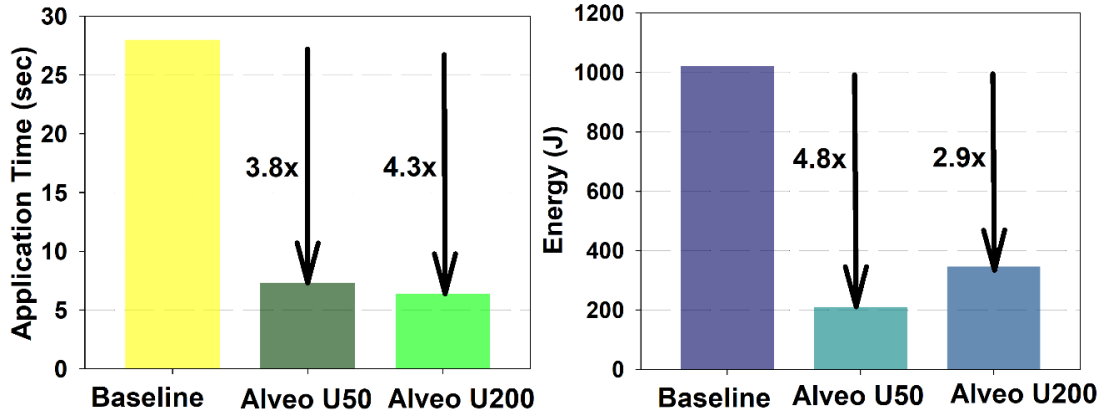


Figure 65: Latency and energy gains of the low approximate Wavelet on the Alveo FPGAs

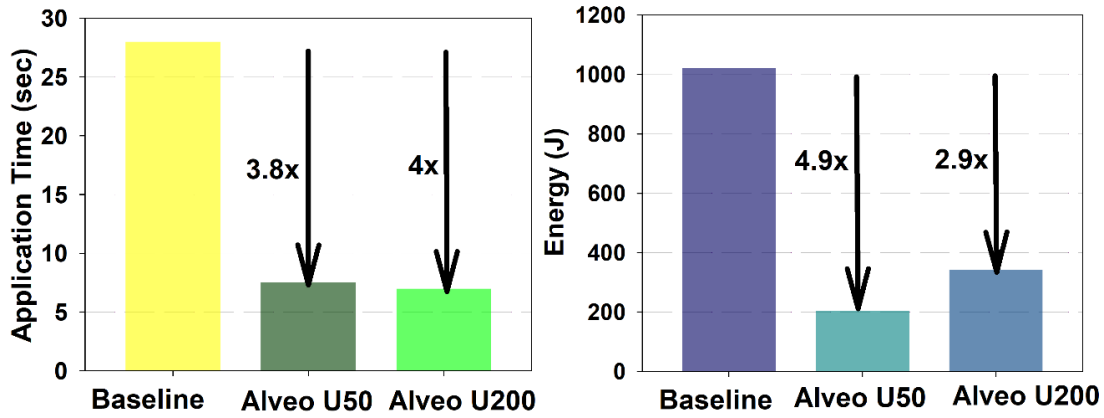


Figure 66: Latency and energy gains of the high approximate Wavelet on the Alveo FPGAs

4.1.3.2.2 Xilinx MPSoC FPGAs

Figure 67 shows the execution time speedup and the energy gains when the accelerators with the low-approximation error are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 2.1x and 2x for the ZCU104 and ZCU102, while the energy gains are 2x and 2.4x respectively. The Mean Absolute Error (MAE) is 0.01.

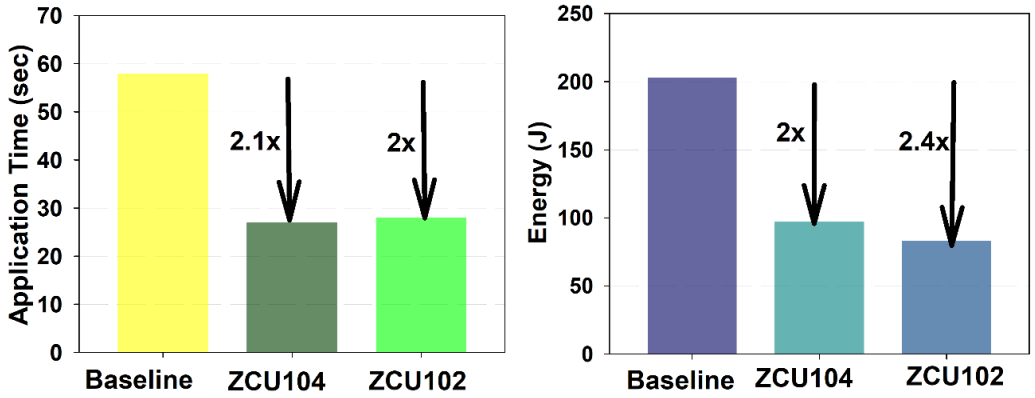


Figure 67: Latency and energy gains of the low approximate Wavelet on the MPSoC FPGAs

Figure 68 shows the execution time speedup and the energy gains when the accelerators with the high-approximation error are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 2.1x for the ZCU104 and ZCU102, while the energy gains are 2.8x and 3x respectively. The Mean Absolute Error (MAE) is 22.2.

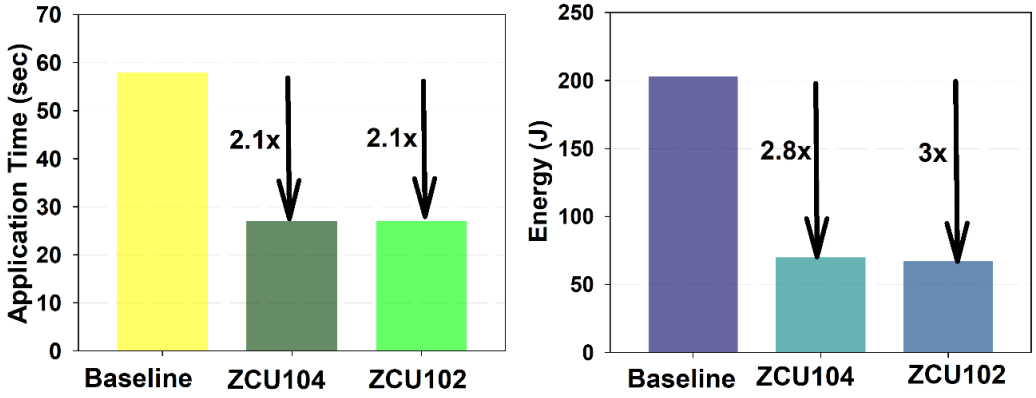


Figure 68: Latency and energy gains of the high approximate Wavelet on the MPSoC FPGAs

4.1.4 Black-Scholes Algorithm

4.1.4.1 Approximate and Transpresicion Techniques

Approximate versions for the calculation of the Black-Scholes formula on FPGA and HPC platforms have been developed.

4.1.4.1.1 Alveo FPGA Acceleration Cards

To design the two approximate accelerators for the Alveo U50 and U200 platforms, two approximation schemes are employed. Firstly, similar to the Savitzky-Golay and Wavelet filter approximate kernels, quantization is applied to the inputs and operands of the kernels.

Specifically, for the low-approximate version, the HLS ***ap_fixed<23, 13, AP_RND_CONV>*** data type is utilised. This means that a 23-bit arithmetic representation is employed, with 10 bits allocated for the fractional part. Conversely, the high-approximate version uses the ***ap_fixed<18, 13, AP_RND_CONV>*** data type. In this case, an 18-bit representation is used, with 5 bits dedicated to the fractional part.

Furthermore, as outlined in Section 2.2.4, computations involving exponential functions are necessary for calculating put and call options. However, performing such calculations on hardware presents challenges and necessitates specialised RTL blocks that heavily utilise the platform's DSP resources. To approximate these functions, Taylor series expansions are employed, enabling the use of polynomials that leverage multipliers and adder trees. The choice of the polynomial order determines the trade-off between performance, power consumption, resource utilisation, and approximation error.

For the low-approximate version, a polynomial order of 15 is employed, while the high-approximate version uses an order of 12.

4.1.4.1.2 Xilinx MPSoC FPGAs

The approximation techniques that are described in the section above were also employed for designing the two MPSoC approximate accelerators. The same quantization factor and the polynomial orders that were described above were used.

4.1.4.2 Evaluation Results

4.1.4.2.1 Alveo FPGA Acceleration Cards

Figure 69 shows the execution time speedup and the energy gains when the accelerators with the low approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 39x and 56x for the U50 and U200, while the energy gains are 75x and 58x respectively. The MAE for the put options is 4.3 and for the call options is 7.5.

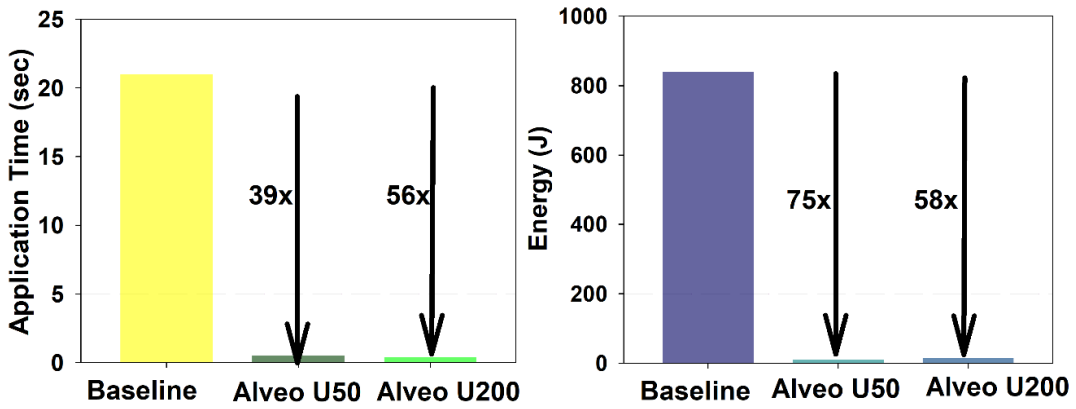


Figure 69: Latency and energy gains of the low approximate Black-Scholes on the Alveo FPGAs

Figure 70 shows the execution time speedup and the energy gains when the accelerators with the high approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups are 76x and 64x for the U50 and U200, while the energy gains are 148x and 71x respectively. The MAE for the put options is 180 and for the call options is 181.

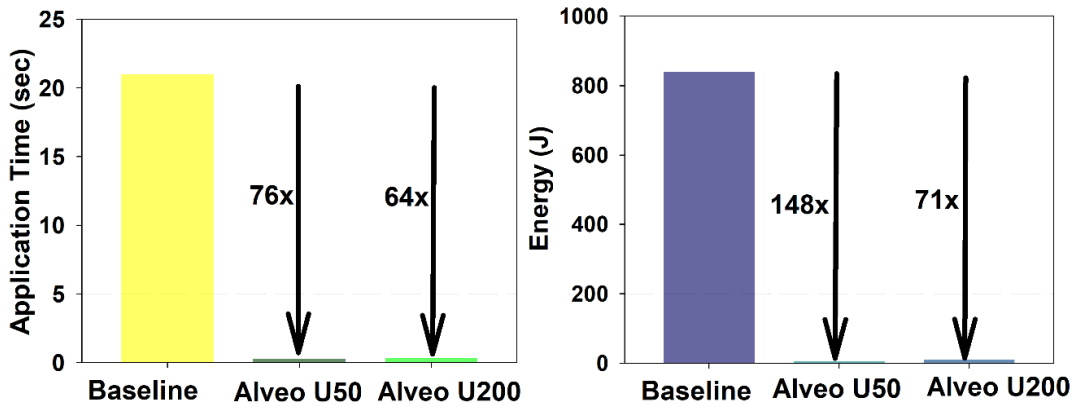


Figure 70: Latency and energy gains of the high approximate Black-Scholes on the Alveo FPGAs

4.1.4.2.2 Xilinx MPSoC FPGAs

Figure 71 shows the execution time speedup and the energy gains when the accelerators with the high-approximation error are executed on the Xilinx MPSoC FPGAs. The execution time speedups are 140x and 118x for the ZCU104 and ZCU102, while the energy gains are 180x and 148x respectively. The MAE for the put options is 180 and for the call options is 181.

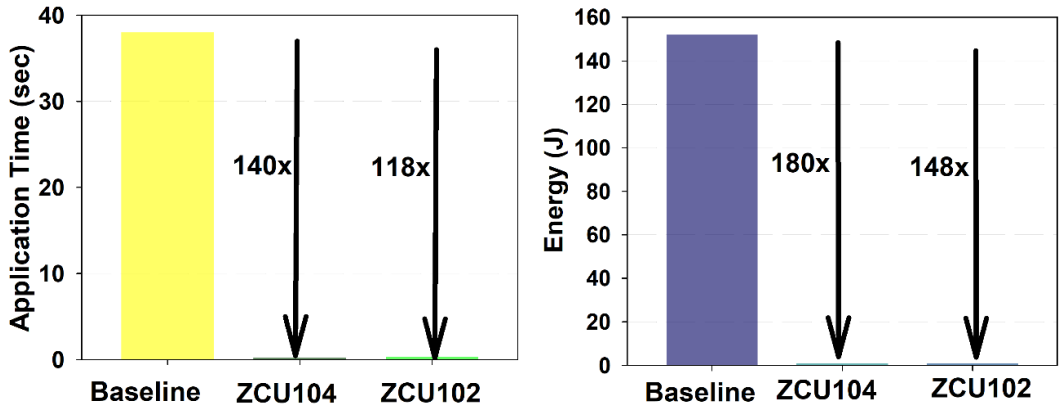


Figure 71: Latency and energy gains of the high approximate Black-Scholes on the MPSoCs

Figure 72 summarises the results for all the low approximate FPGA designs for the UC2, while Figure 73 summarises the results for all the high approximate FPGA designs.

Application	UC	Device	Latency (ms)	BRAM	DSP	FF	LUT	MAE	Power (W)	Energy gain	Speedup
Savitzky-Golay	INB	U50	1603	0.8%	0%	4.5%	12%	0.17	24.8	2.4x	2.1x
Savitzky-Golay	INB	U200	1278	0.8%	0%	3.8%	9.3%	0.17	48.6	1.5x	2.6x
Savitzky-Golay	INB	ZCU104	3120	1.7%	0%	8.8%	19.9%	0.17	2.3	2.7x	2x
Savitzky-Golay	INB	ZCU102	2860	0.5%	0%	7.4%	18.6%	0.17	2.3	3x	2.2x
Kalman	INB	U50	2132	25.6%	2.7%	0%	14.4%	0.05	29.6	7719x	5064x
Kalman	INB	U200	2481	9.1%	5.8%	16.2%	20.2%	0.05	50.4	4352x	3899x
Kalman	INB	ZCU104	14483	20%	2%	10%	20%	0.05	2.5	4948x	2707x
Kalman	INB	ZCU102	33501	6%	2%	8%	16%	0.05	2.5	2139x	1170x
Wavelet	INB	U50	7306	4.6%	16.8%	13.5%	24%	0.01	29	4.8x	3.8x
Wavelet	INB	U200	6408	4%	14%	10.9%	19.5%	0.01	54.2	2.9x	4.3x
Wavelet	INB	ZCU104	27050	9.2%	23%	19%	33%	0.01	3.6	2x	2.1x
Wavelet	INB	ZCU102	28740	2%	17%	11.9%	20.3%	0.01	2.9	2.4x	2x
Black Scholes	INB	U50	533	2%	11.6%	4.7%	9.8%	~5	20.8	75x	39x
Black Scholes	INB	U200	372	1.3%	10.1%	3.8%	7.4%	~5	38.8	58x	56x
Black Scholes	INB	ZCU104	289	8.8%	40.7%	13.5%	26.7%	~5	3.1	169x	131x
Black Scholes	INB	ZCU102	323	2.7%	22.9%	11.2%	22.3%	~5	3.3	142x	117x

Figure 72: Summary of all the low approximate FPGA designs for the UC2

Application	UC	Device	Latency (ms)	BRAM	DSP	FF	LUT	MAE	Power (W)	Energy gain	Speedup
Savitzky-Golay	INB	U50	1517	0.8%	0%	3.8%	9.6%	6.4	24.2	2.6x	2.2x
Savitzky-Golay	INB	U200	1236	0.8%	0%	3.1%	8.3%	6.4	48.5	1.6x	2.7x
Savitzky-Golay	INB	ZCU104	2850	1.7%	0%	7.6%	15.2%	6.4	2.2	3.1x	2.2x
Savitzky-Golay	INB	ZCU102	2840	0.5%	0%	6.4%	12.7%	6.4	2.2	3.2x	2.2x
Kalman	INB	U50	1933	25.6%	13.4%	0%	25.6%	1.15	32.3	7801x	5586x
Kalman	INB	U200	2417	8%	6%	8%	9.4%	1.15	49.2	4468x	4100x
Kalman	INB	ZCU104	13495	16%	10%	20%	34%	1.15	3.3	4023x	2905x
Kalman	INB	ZCU102	25950	6%	6%	16%	28%	1.15	2.9	2380x	1510x
Wavelet	INB	U50	7580	4.6%	6.5%	6%	12%	22.2	27	4.9x	3.6x
Wavelet	INB	U200	7000	4%	5.7%	5.1%	9.1%	22.2	49	3x	4x
Wavelet	INB	ZCU104	27000	9.1%	7.5%	7.2%	13%	22.2	2.6	2.8x	2.1x
Wavelet	INB	ZCU102	27050	3.1%	7.8%	9.4%	17.2%	22.2	2.5	3x	2.1x
Black Scholes	INB	U50	275	2%	7.7%	3%	6.4%	~180	20.6	148x	76x
Black Scholes	INB	U200	326	1.3%	6.7%	2.4%	5%	~180	36.2	71x	64x
Black Scholes	INB	ZCU104	271	8.8%	42%	11.7%	23%	~180	3.1	180x	140x
Black Scholes	INB	ZCU102	320	2.7%	28.9%	9.8%	19%	~180	3.2	148x	118x

Figure 73: Summary of all the high approximate FPGA designs for the UC2

4.2 Approximation of Anomaly Detection in Manufacturing Settings (UC3, IDEKO) Algorithms

4.2.1 DBSCAN Clustering Algorithm

4.2.1.1 Design Implementation

We provide two approximate versions for each FPGA-accelerated version of the Time Series DBSCAN, targeting the different devices available on the SERRANO platform. Due to the complexity of the DBSCAN algorithm, specifically communication infrastructure complexity, an HPC acceleration is still under implementation.

4.2.1.1.1 Alveo FPGA acceleration cards

Our approximation strategy is based on the loop perforation approximation technique described in Deliverable D4.2. Considering that N are the DTW distance calculations that need to be computed and M is the number of available compute units on the FPGA, a total of N/M batches are computed. As with loop perforation, where loop iterations are omitted, we omit all DTW distance computations every K batches. The number of K is fixed at 75 for the low approximation variant and 25 for the high approximation variant in the Alveo U50 and U200 FPGAs.

4.2.1.1.2 Xilinx MPSoC FPGAs

The approximation schemes used in the Alveo designs were also used to design the approximate accelerators in the MPSoC platforms.

4.2.1.2 Evaluation Results

4.2.1.2.1 Alveo FPGA Acceleration Cards

Figure 74 shows the speedup and energy gains of the low approximation error version of the Time Series DBSCAN for the Alveo U50 and U200 FPGAs compared to the Python single-threaded execution. The speedups are 399x and 118x for the U50 and U200, while the energy gains are 178x and 31.3x, respectively. The low approximation error version can achieve 1.2x higher speedups and 1.2x higher energy gains compared to the accurate versions for the Alveo U50 and U200 FPGAs. These advantages are associated with an accuracy of 90%, which means that 90% of the examined signals were correctly classified compared to the exact version of the algorithm.

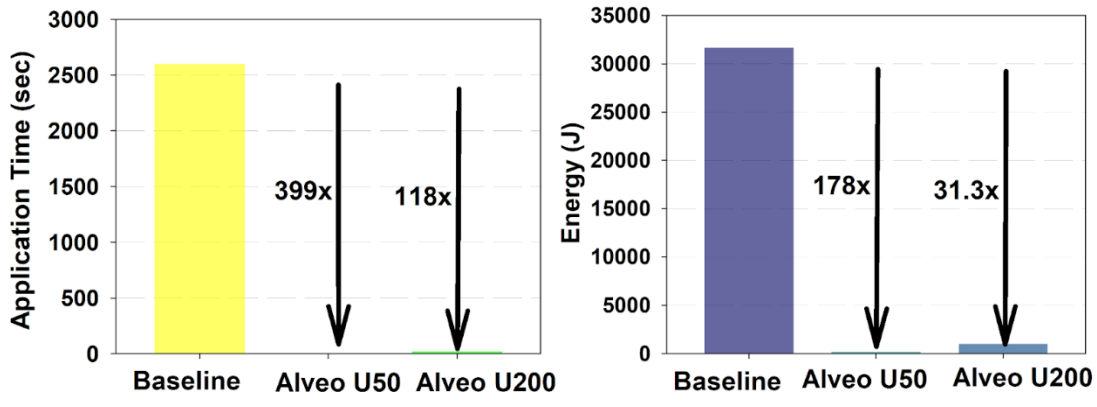


Figure 74: Latency and energy gains of the low approximate DBSCAN for the Alveo FPGAs

Figure 75 shows the corresponding results for the version of the Time Series DBSCAN with high approximation error. The speedups are 449x and 131x for the U50 and U200, while the energy gains are 200x and 34.7x, respectively. The high approximation error version can achieve 1.3x higher speedups and 1.3x higher energy gains compared to the accurate versions

for the Alveo U50 and U200 FPGAs. These benefits are associated with an accuracy of 82%. It should be noted that we did not use higher values for K to account for the IDEKO requirement that accuracy should not be less than 80%.

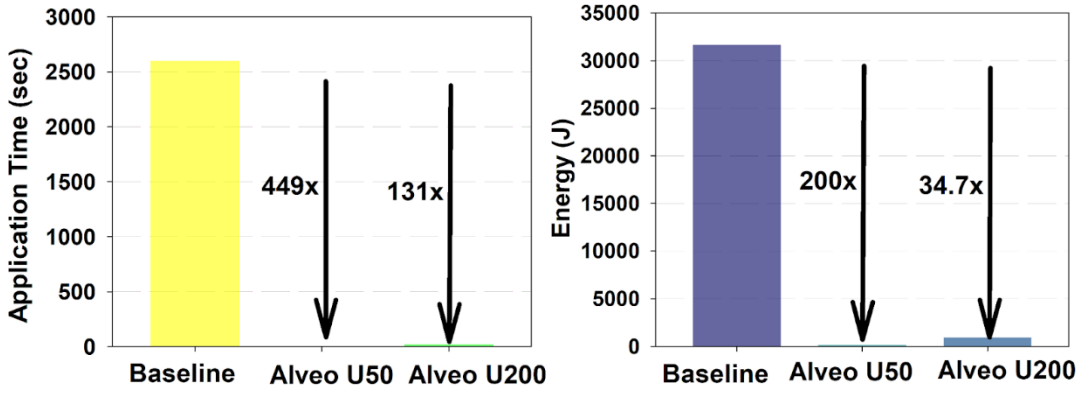


Figure 75: Latency and energy gains of the high approximate DBSCAN for the Alveo FPGAs

4.2.1.2.2 Xilinx MPSoC FPGAs

Figure 76 shows the speedup and energy gains of the low approximation error version of the Time Series DBSCAN for the MPSoC ZCU104 and ZCU102 FPGAs compared to the Python single-threaded execution. The speedups are 20x and 15.3x for the ZCU104 and ZCU102, while the energy gains are 47.8x and 40.5x, respectively. The low approximation error version can achieve 1.12x and 1.13x higher speedups and 1.12x and 1.13x higher energy gains compared to the accurate versions for the MPSoC ZCU104 and ZCU102 FPGAs. Similar to cloud devices, the accuracy is 90%.

Figure 77 shows the corresponding results for the version of the Time Series DBSCAN with high approximation error. The speedups are 22.4x and 16.5x for the ZCU104 and ZCU102, while the energy gains are 53.5x and 43.5x, respectively. The high approximation error version can achieve 1.3x and 1.2x higher speedups and 1.3x and 1.2x higher energy gains compared to the accurate versions for the MPSoC ZCU104 and ZCU102 FPGAs. Similar to cloud devices, the accuracy is 82%.

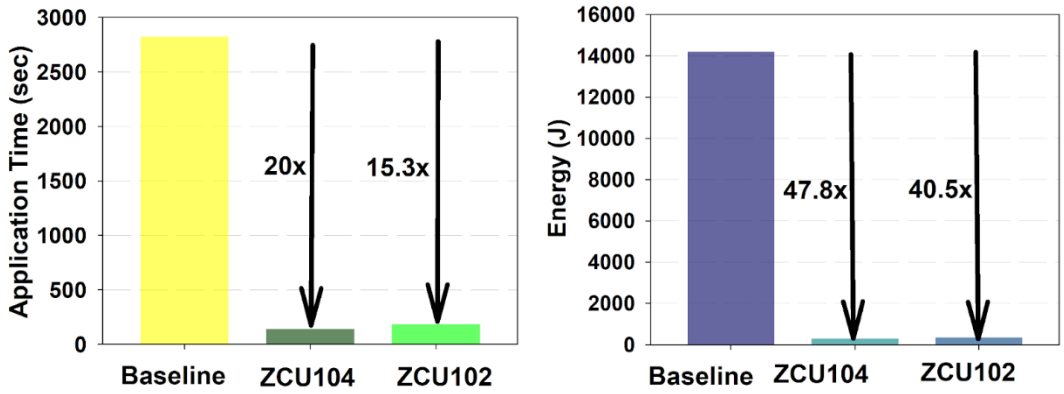


Figure 76: Latency and energy gains of the low approximate DBSCAN for the MPSoC FPGAs

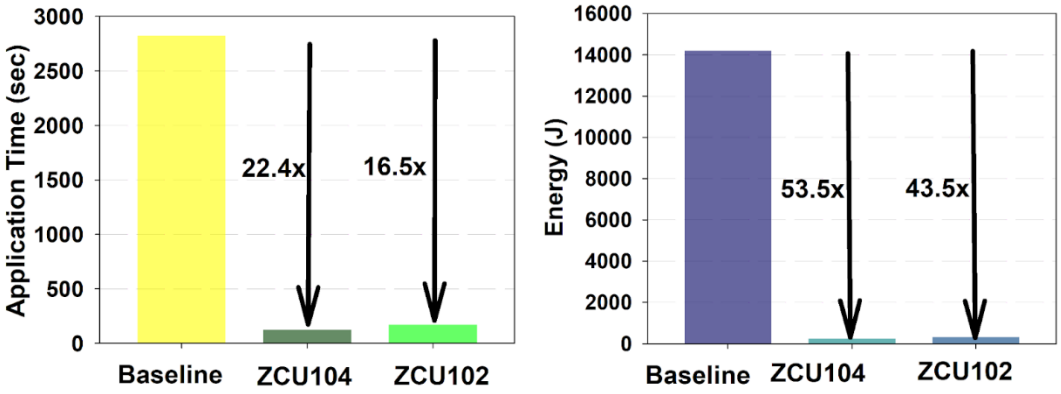


Figure 77: Latency and energy gains of the high approximate DBSCAN for the MPSoC FPGAs

4.2.2 1D-FFT Algorithm

4.2.2.1 Approximate and Transpresicion Techniques

Approximate versions of the 1D-FFT HPC accelerators have been developed, whereas the approximate versions for FPGA, specifically with the precision scaling, were not performed due to the usage of a Xilinx IP that does not allow the usage of custom data types (from ap_int and ap_fixed libraries).

4.2.2.1.1 HPC

The approximation computing techniques, discussed in Section 3.1.2 and Section 3.1.1, for the Kalman filter and the Savitzky-Golay filter cannot be directly applied to the FFT (Fast Fourier Transform) filter. In the FFT filter, time series signals are transformed into the frequency domain. If some of the input data is skipped to reduce the workload, the FFT algorithm will generate completely different results. Therefore, approximation computing techniques cannot be employed in the FFT filter. However, the transprecision computing techniques described in the section can still be applied to the FFT filter. Using lower precision data for both input and output, reduce execution time and energy consumption, while maintaining the accuracy of the results.

4.2.2.2 Evaluation Results

4.2.2.2.1 HPC

Transprecision techniques have been applied in the FFT kernel. Table 19 demonstrates the application of mixed data precision for both input and output data. As a result, the execution time and energy consumption have been improved, and the accuracy of the output will not be changed.

Table 19: Transprecision techniques in FFT with 104 acceleration data

Input data precision	Output data precision	Execution time (sec)	Energy Consumption (Joule)	Execution time improvement	Energy consumption improvement
double	double	0.2914	58.75	-	-
double	float	0.2051	41.34	30%	30%
float	double	0.2681	54.05	10%	11%
float	float	0.1888	38.07	35%	64%

4.2.3 K-Means Clustering Algorithm

4.2.3.1 Approximate and transpresicion techniques

Approximate versions of the Timeseries K-Means clustering for FPGA and HPC accelerators have been developed.

4.2.3.1.1 Alveo FPGA Acceleration Cards

The Alveo U50 and U200 FPGA acceleration cards make use of approximate designs that take advantage of HLS (High-Level Synthesis) arbitrary precision data types to quantize the inputs and intermediate operands. In the variant with low approximation only 1 K-Means iteration is used which trade-offs speed in the algorithm execution but with lower accuracy. In the design with the high approximation error, the floating-point values are converted to the ap_fixed<25,24,AP_RND, AP_SAT> data type. This particular fixed-point representation assigns 24 bits to the integer part and 1 bit to the fractional part. This approximate design can be seamlessly compiled using the 'AXX=1' flag during compilation.

4.2.3.1.2 Xilinx MPSoC FPGAs

The quantization schemes that were used in the Alveo designs were also employed for the design of the approximate accelerators in the MPSoC platforms.

4.2.3.1.3 HPC

Approximation computing techniques will be implemented in the K-means kernel, and it has a significant improvement in the execution time of K-means. This technique is known as quality-based control loop. These loops determine how many times certain parts of an algorithm are executed until the desired level of convergence is achieved. Quality-based control loops continuously monitor an internal metric and halt the iterations when a condition based on both internal state and user-specified parameters is met (Figure 78). Typically, users rely on default values for these parameters or choose conservative values, which can lead to suboptimal performance.

To address this issue, approximation computing techniques allow for the relaxation of the control parameters. Consequently, the loop may not iterate as much, and the output may differ from the default implementation. It is important to mention that control parameters must be carefully chosen to ensure that the desired level of quality is maintained.

```
while(error < epsilon){…}.
```

Figure 78: Quality based control loop

Quality-based control loop techniques were incorporated into the K-means kernel implementation, as explained in Section 2.3.3 Figure 41. By introducing a condition within the loop and monitoring the distance between two centroids during each iteration, we can stop the iteration and break the while loop if the difference between these centroids is less than a control parameter (epsilon). While the default implementation has the loop iterate 10 times, we can apply different values such as e-4, e-2, and e-1 for epsilon. The goal is to reduce the number of iterations and execution time by using a larger epsilon value, but we also anticipate that classification accuracy may suffer due to this approximation technique.

4.2.3.2 Evaluation Results

4.2.3.2.1 Alveo FPGA Acceleration Cards

Figure 79 shows the execution time speedup and the energy gains when the accelerators with the low approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups now are 183x and 192x for the U50 and U200, while the energy gains are 359x and 402x respectively. The accuracy of this version is 95.7%.

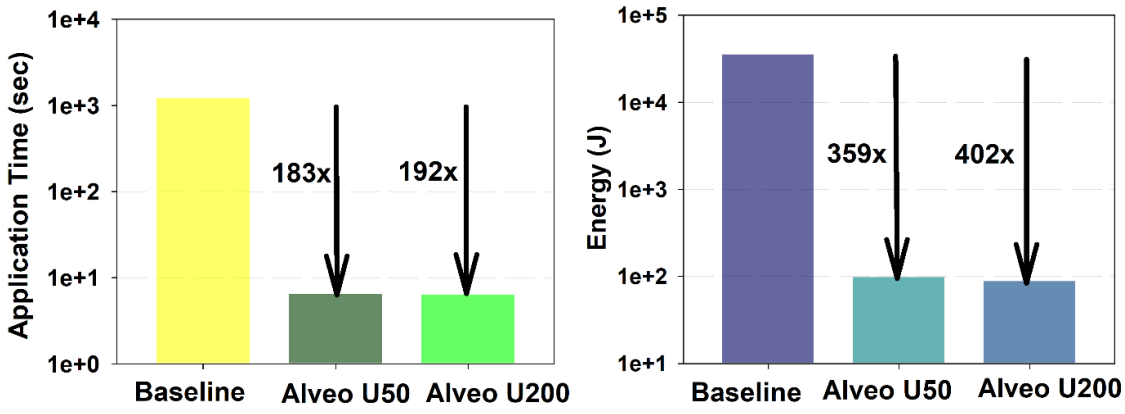


Figure 79: Latency and Energy gains for K-Means with low approximation error on Alveo FPGAs

Figure 80 shows the execution time speedup and the energy gains when the accelerators with the high approximation error are executed on the Alveo Xilinx acceleration cards. The execution time speedups now are 725x and 760x for the U50 and U200, while the energy gains are 1410x and 1583x respectively. The accuracy of this version is 75%.

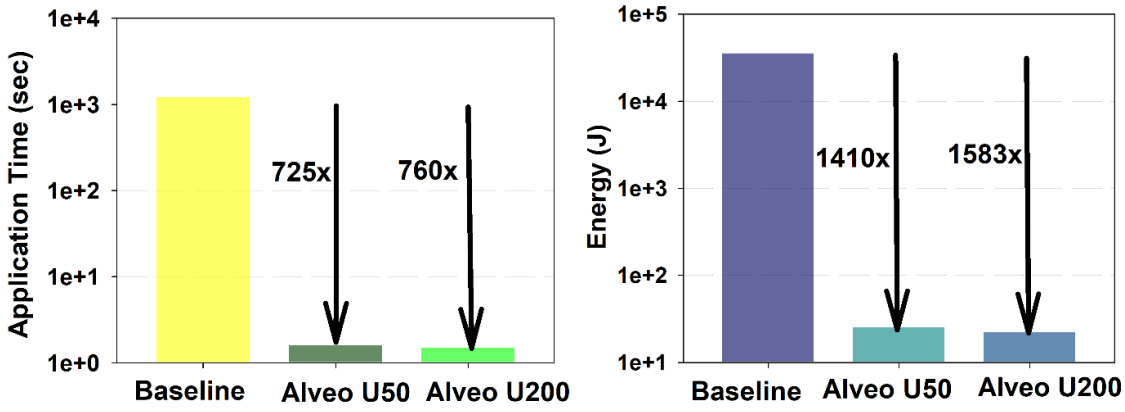


Figure 80: Latency and Energy gains for K-Means with high approximation error on Alveo FPGAs

Figure 81 shows the execution time speedup and the energy gains when the accelerators with the low approximation error are executed on the MPSoC FPGAs. The execution time speedup now is 248.7x while the energy gain is 332x respectively. The accuracy of this version is 95.7%.

4.2.3.2.2 Xilinx MPSoC FPGAs

Figure 82 shows the execution time speedup and the energy gains when the accelerators with the high approximation error are executed on the MPSoC FPGAs. The execution time speedup now is 762.5x while the energy gain is 916.2x respectively. The accuracy of this version is 75%.

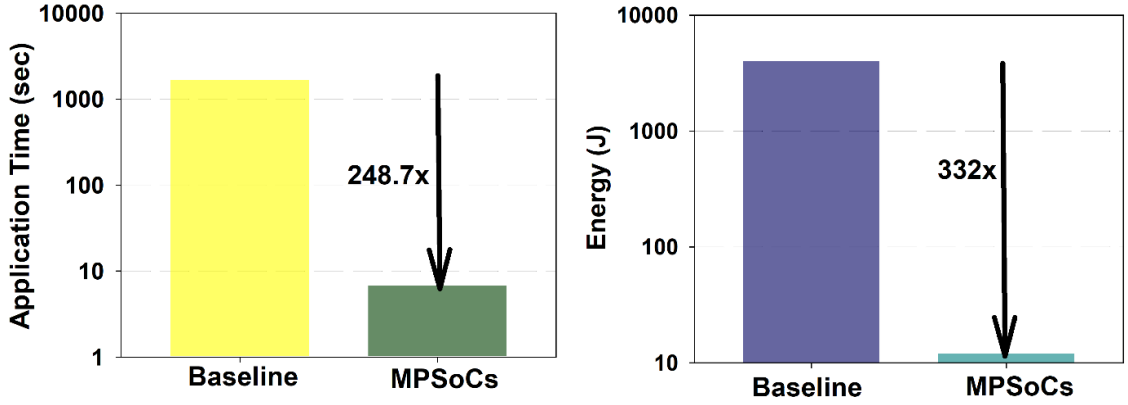


Figure 81: Latency and Energy gains for K-Means with low approximation error on MPSoC FPGAs

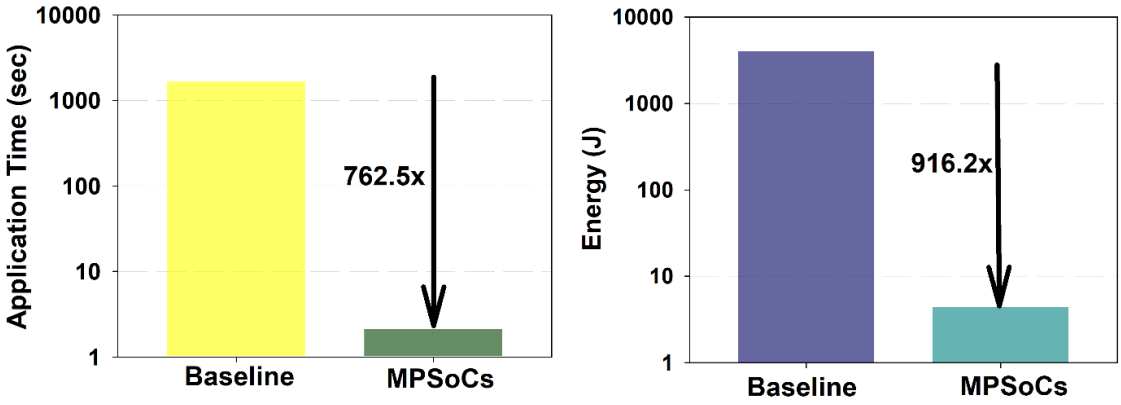


Figure 82: Latency and Energy gains for K-Means with high approximation error on MPSoC FPGAs

4.2.3.2.3 HPC

Table 20 presents the implementation of the quality-based control approximation technique on the K-means kernel, which was utilised for the IDEKO position signal. To carry out this implementation, we set the epsilon value to e-4 and re-executed the K-means classification on the datasets. As a result, we observed a reduction in the number of iterations within the loop, which in turn led to improvements in the execution time and energy consumption. Notably, the accuracy of classification remained unchanged, and in all datasets, the K-means kernel, when implemented with the quality-based control loop approximation technique, produced identical classification results.

Table 20: Approximation computing techniques in the K-Means on the HPC system

IDEKO signal	Epsilon	Execution time (sec)	Execution time improvement	Energy consumption improvement	Accuracy
110 position signal	e-4	2.181	5.0X	8.9X	100%
330 position signal	e-4	2.445	3.1X	3.1X	100%
550 position signal	e-4	3.339	2.4X	2.4X	100%
770 position signal	e-4	3.357	2.4X	2.4X	100%
990 position signal	e-4	3.403	2.3X	2.3X	100%
1100 position signal	e-4	5.726	2.2X	2.2X	100%

Transprecision techniques will be applied into the K-means Kernel. By employing Two different data types: double precision and single precision (float), for both input and output data types, denoted as I and K, respectively. Table 21 presents how combining these different data types can potentially improve the execution runtime and reduce energy consumption.

Table 21: Transprecision computing techniques in the K-Means on the HPC system

Input data precision	Output data precision	Execution time (sec)	Energy Consumption (Joule)	Execution time improvement	Energy consumption improvement
double	double	2.251	907.548	-	-
double	float	2.136	861.448	5%	5%
float	double	2.146	865.445	5%	5%
float	float	1.965	792.618	13%	12%

4.2.4 KNN Clustering Algorithm

4.2.4.1 Approximate and transpresicion techniques

Approximate versions of the Timeseries KNN classification algorithm for FPGA and HPC accelerators have been developed.

4.2.4.1.1 Alveo FPGA Acceleration Cards

The Alveo U50 and U200 FPGA acceleration cards make use of approximate designs that take advantage of HLS (High-Level Synthesis) arbitrary precision data types to quantize the inputs and intermediate operands. In the design with the high approximation error, the floating-point values are converted to the ap_fixed<25,24,AP_RND, AP_SAT> data type. This particular fixed-point representation assigns 24 bits to the integer part and 1 bit to the fractional part. This approximate design can be seamlessly compiled using the 'AXX=1' flag during compilation.

4.2.4.1.2 Xilinx MPSoC FPGAs

The quantization schemes that were used in the Alveo designs were also employed for the design of the approximate accelerators in the MPSoC platforms.

4.2.4.1.3 HPC

In the KNN kernel, approximation computing techniques are not typically introduced, as they can compromise the accuracy of the results. In this specific use case, the KNN kernel is responsible for classifying inference signals between two classes, and introducing uncertainty through approximation techniques may not be desirable.

However, transprecision computing techniques can still be applied in the KNN kernel. By utilising different and lower precision data types for the input data, execution time and energy consumption can be reduced without significantly impacting the accuracy of the classification results.

4.2.4.2 Evaluation Results

4.2.4.2.1 Alveo FPGA Acceleration Cards

Figure 83 shows the execution time speedup and the energy gains when the accelerators with approximation are executed on the Alveo Xilinx acceleration cards. The execution time speedups now are 61.1x and 85.1x for the U50 and U200, while the energy gains are 133x and 198.6x respectively. The accuracy of this version is 95.4%.

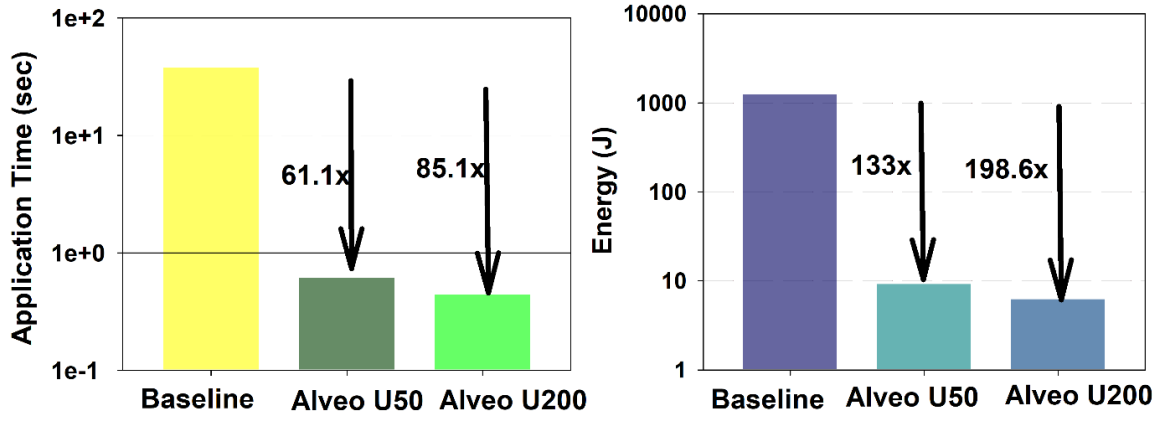


Figure 83: Latency and Energy gains for approximate K-NN on Alveo FPGAs

4.2.4.2.2 Xilinx MPSoC FPGAs

Figure 84 shows the execution time speedup and the energy gains when the accelerators with approximation are executed on the MPSoC FPGAs. The execution time speedup is now 211x, while the energy gain is 116x.

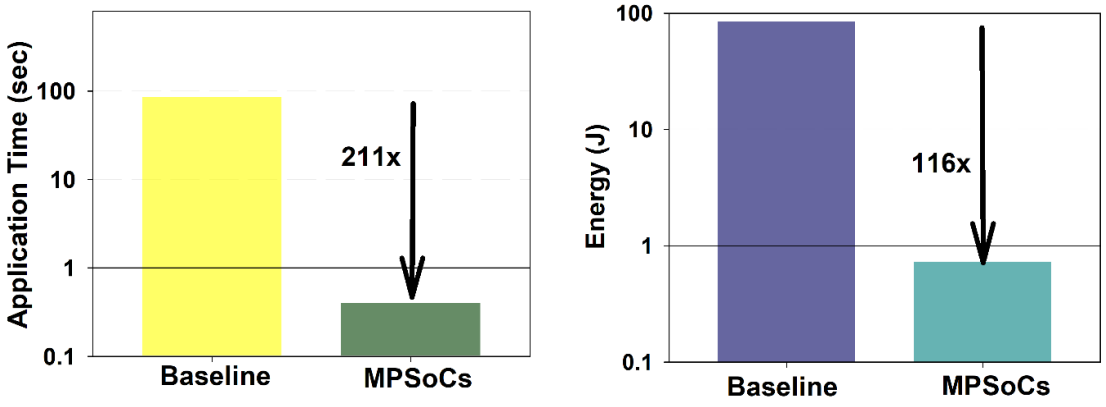


Figure 84: Latency and Energy gains for approximate K-NN on MPSoC FPGAs

4.2.4.2.3 HPC

To assess the impact of transprecision techniques on the KNN kernel, two different data types were used: double precision and single precision for both input data. Table 22 shows the execution time and energy improvements achieved by implementing mixed data precision in the KNN Kernel.

Table 22: Transprecision computing techniques in the KNN on the HPC system

Input data precision	Execution time (sec)	Energy Consumption (Joule)	Execution time improvement	Energy consumption improvement
double	3.930	1584.61	-	-
float	3.488	1406.66	11.5%	11.6%

Figure 85 below summarises the results for all the low approximate FPGA designs for the UC3. Similarly, Figure 86 summarises the results for all the high approximate FPGA designs.

Application	UC	Device	Latency (ms)	BRAM	DSP	FF	LUT	Accuracy	Power (W)	Energy gain	Speedup
DBSCAN	IDK	U50	6531	7.2%	0.6%	0%	4.2%	90%	27.3	178x	399x
DBSCAN	IDK	U200	21959	2.2%	0.2%	1.6%	1.5%	90%	46	31.2x	118x
DBSCAN	IDK	ZCU104	141559	10%	2%	2%	6%	90%	2.1	47.8x	20x
DBSCAN	IDK	ZCU102	184513	2%	2%	2%	4%	90%	1.9	40.5x	15.3x
KMEANS	IDK	U50	6594	1.7%	0%	0.5%	0.8%	95.4%	15	359x	183.7x
KMEANS	IDK	U200	6304	1%	0%	0.3%	0.6%	95.4%	14	192.1x	402x
KMEANS	IDK	ZCU104	6714	6.8%	0.1%	1.5%	2.5%	95.4%	1.8	332x	248x
KMEANS	IDK	ZCU102	6714	2.2%	0%	1.2%	2.1%	95.4%	1.8	332x	248x
KNN	IDK	U50	618	1.7%	0%	0.5%	0.8%	95.4%	15	133x	61x
KNN	IDK	U200	444	1%	0%	0.4%	0.6%	95.4%	15.1	198x	85x
KNN	IDK	ZCU104	407	6.8%	0.1%	1.5%	2.5%	95.4%	1.8	211x	116x
KNN	IDK	ZCU102	407	2.7%	0%	1.2%	2.1%	95.4%	1.8	211x	116x

Figure 85: Summary of all the low approximate FPGA designs for the UC3

Application	UC	Device	Latency (ms)	BRAM	DSP	FF	LUT	Accuracy	Power (W)	Energy gain	Speedup
DBSCAN	IDK	U50	5782	7.2%	0.6%	0%	4.2%	82%	27.3	200x	449x
DBSCAN	IDK	U200	19810	2.2%	0.2%	1.6%	1.5%	82%	46	34x	131x
DBSCAN	IDK	ZCU104	126368	10%	2%	2%	6%	82%	2.1	53.5x	22.4x
DBSCAN	IDK	ZCU102	171533	2%	2%	2%	4%	82%	1.9	43.5x	16.5x
KMEANS	IDK	U50	1669	1%	0%	0.4%	0.6%	75%	15.1	1410x	725x
KMEANS	IDK	U200	1592	0.6%	0%	0.3%	0.4%	75%	14.1	760x	1583x
KMEANS	IDK	ZCU104	2190	4.1%	0.3%	1.4%	1.9%	75%	2	916x	762x
KMEANS	IDK	ZCU102	2190	2.6%	0.2%	1.2%	1.6%	75%	2	916x	762x

Figure 86: Summary of all the high approximate FPGA designs for the UC3

4.3 Verification, Validation, and Uncertainty Quantification (VVUQ)

In Sections 3.1 and 3.2, we discussed transprecision and approximation computing techniques and how they have been integrated into kernel implementation. Transprecision techniques involve utilising limited data precision during kernel execution, and we have observed that

using low precision data types can reduce memory footprint, execution run time, and energy consumption without compromising the accuracy of the output. We have also applied approximation techniques, which consist of loop perforation and quality-based control loops. Both of these techniques reduce computation run time and energy consumption at the cost of changing the accuracy of the output results.

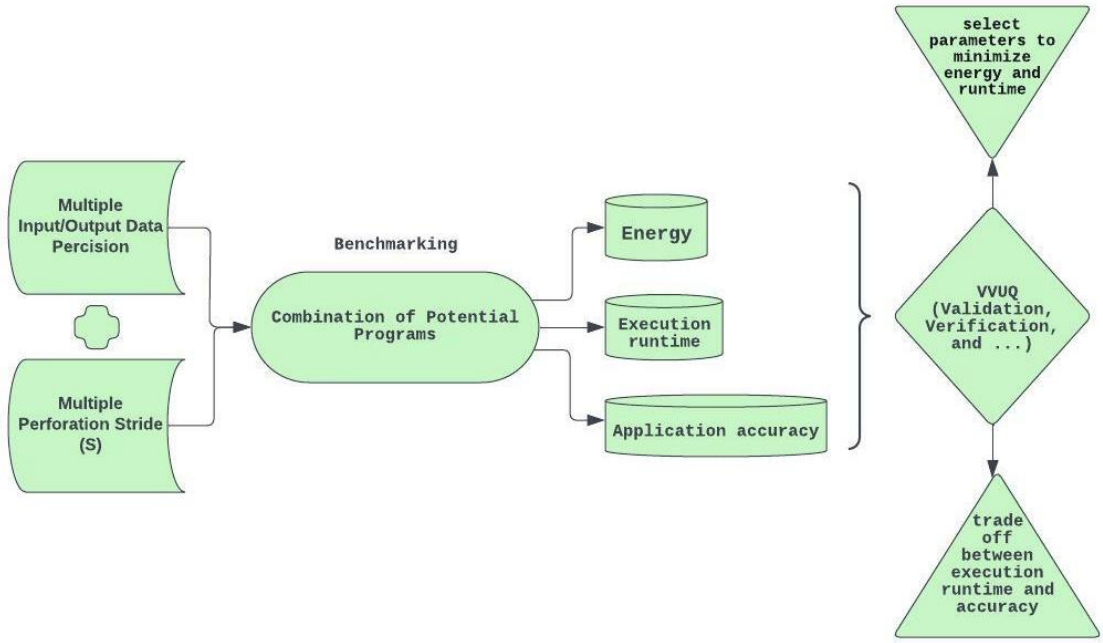


Figure 87: Verification Validation, and Uncertainty Quantification (VVUQ)

However, a question arises about how we can manage all these parameters and uncertainties in the kernel. The SERRANO platform, we have developed Verification, Validation, and Uncertainty Quantification [19] (VVUQ) to choose the best combination of input/output data precision and loop perforation to minimise computation run time and energy consumption. Additionally, VVUQ addresses the trade-off between the accuracy of the results and execution time, and helps use case providers choose the best parallel parameter number of processes to achieve these optimizations (Figure 87).

4.3.1 Automated Benchmarking

Various parameters come into play during kernel execution. Regarding transprecision techniques, we consider the mixed data precision scenario for input and output, where {double-double, double-float, float-double, float-float} are the available options, as illustrated in Figure 88. We use different perforation stride values for approximation techniques, such as s={1,2,4,8,16}. Additionally, for parallel kernel execution, we use num_MPI_Procs to denote the number of processes, which can take values of {1, 2, 4, 16, 32, 64, 128, 256}. To automate benchmarking, we utilise a nested loop structure similar to Figure 88, where we execute the kernel for each combination of these parameters and monitor execution time, energy

consumption, and application accuracy. Therefore, automated Benchmarking leads to the automatic generation of files containing profiling measurements of the kernel in all possible combinations.

```
EXE=build/SERRANO

for precision_senario in {1,2,3,4}
  do echo precision_senario $precision_senario
   for perforation_stride in {1,2,4,8,16}
     do echo perforation_stride $perforation_stride
            for num_MPI_Procs in {1,2,4,8,16,32,64,128}
               do echo num_MPI_Procs $num_MPI_Procs
  #           sleep 1
mpirun --oversubscribe -np $num_MPI_Procs $EXE $icase $
```

Figure 88: Automated Benchmarking with regard to approximation and transprecision techniques

4.3.2 VVUQ User Interface

We will describe how users can interact with the VVUQ interface, using the Kalman Filter kernel as an example. After conducting Automated Benchmarking on the input data, several profiling files are automatically generated, as previously explained. With the VVUQ framework, users can explore all of these profiling files and quantify the uncertainty to obtain optimal parameters such as the number of processes, input and output data precision, and performance stride for executing the Kalman Filter that minimises execution time and energy consumption. In the bash script, users can specify the kernel name and input data size while leaving the remaining parameters at their default values.

```
    #Choosing Filter
kernel="KalmanFilter"

    #Input data size
Data_size="110"

     #Data percision for Input and Output data (transprecision techniques)
Input_data_precision="NULL"
Output_data_precision="NULL"


    #Applying the approximate techniques
Apply_approximation_techniques=-1


     #Applying the affordable error
Apply_error_range=1
error_offset="50"
error_endset="150"
```

Figure 89: VVUQ configuration Interface

By executing the bash script, the VVUQ framework provides the optimal parameters for executing the Kalman filter (Figure 90). To achieve minimum execution runtime and energy consumption, the kernel needs to be executed with 128 MPI cores and with input and output data precision set as float, while using a perforation stride of 4, and the L2 absolute error norm would be 48.58.

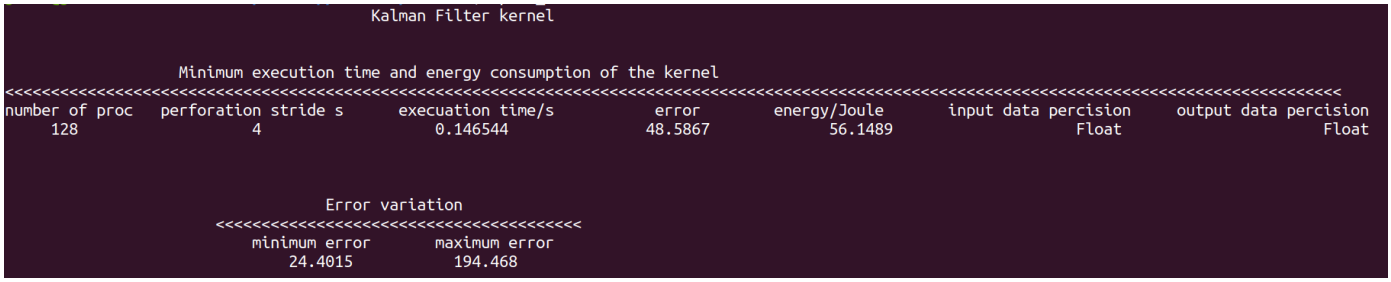


Figure 90: Optimal parameter received by VVUQ framework to execute Kalman filter in parallel

We also received information about how the error varies due to loop perforation in approximation techniques. Increasing the perforation stride reduces the execution time but increases the error. We can adjust the affordable error range for our case by setting the parameters for error offset and error end set. Therefore, in task 4.2, we need to balance application accuracy and execution runtime. By applying the error_offset of 50 and error_endset of 150, we received a new parallel parameter that minimises execution time and energy consumption while considering the affordable error we just set.

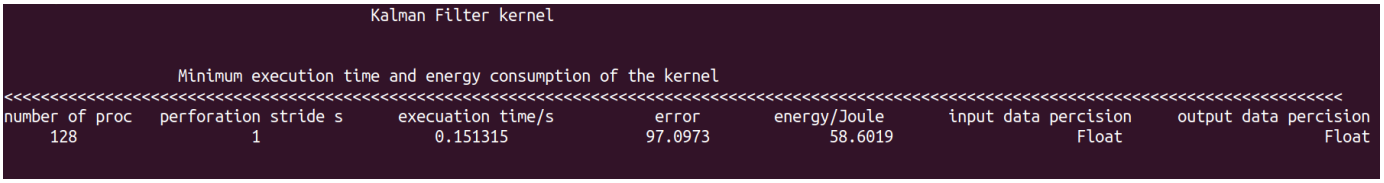


Figure 91: VVUQ addresses the trade-off between accuracy and execution time

4.3.3 Kernel Performance Approximation

In section 3.3.2, we have introduced the VVUQ framework and its ability, in conjunction with Automated Benchmarking, to evaluate the accuracy and reliability of results. However, one potential drawback of utilising this framework is that it may lead to increased costs, as it requires more computational time to consider all combinations of data precision, perforation stride, and multiple numbers of cores, and then identify uncertainties and validate the results. To tackle these difficulties, we plan to employ the VVUQ framework on a small number of use cases. We will then combine these outcomes with AI/ML techniques to determine the best parameter values for new data batches. By doing so, we can overcome the computational expenses associated with using the VVUQ framework while enhancing the efficiency and dependability of the SERRANO digital platform.

In Table 4, we have computed the minimum execution time for different data batches in each kernel. Consequently, the Gradient Descent method was applied to achieve a nonlinear formula that approximates the minimum execution time. This formula is subsequently used to estimate the minimum execution time of the random signals we obtained. In this approach, the gradient descent method was applied to compute the optimal parameters {a, b, c} in the following equation.

$$Y = a * exp(-b * x) + c$$

Table 13 displays the benchmark results for the parallel KNN kernel using several acceleration data batches from IDEKO, indicating that we achieved the minimum execution time. With this information, we can utilise the Gradient Descent method to determine the most accurate nonlinear formula (Table 23) that approximates the minimum execution time of the KNN kernel across various data sizes.

Table 23: Nonlinear formula achieved by gradient descent method

KNN minimum Execution time approximation	Formula
Nonlinear function (Gradient Decent)	1.03247*exp(0.0294826*x) - 0.0302112
Linear Function (Linear Regression)	0.0382379*x + 1.11035

Nonlinear formula and linear approximation in addition to the original minimum execution time measurement displayed in Figure 92. By applying the nonlinear formula, we can optimally approximate the minimum execution time of the KNN kernel for various data sizes.

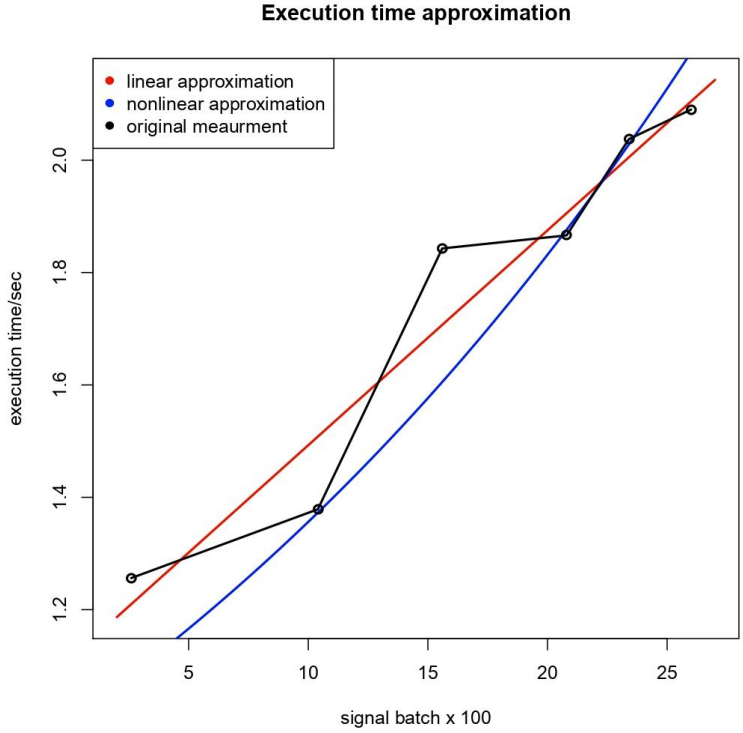


Figure 92: Nonlinear and Linear approximation of minimum execution time

4.4 Detection Methods and Energy Consumption

This section describes the experiments carried out regarding algorithmic transprecise adaptation of ML analysis and detection methods. We aim to investigate models and techniques that enable distributed streaming applications to be deployed and redistributed across edge/cloud computing systems. By utilising devices that are dispersed through a network that is in the proximity of end users, it is possible to reduce network latency and increase the available bandwidth. It is important to mention that these types of user networks where the target edge devices are located are intrinsically dynamic. In the case of mobile devices, it is easy to see that these connect to different end points (base stations) as they roam. Thus, these devices might be available in an intermittent manner for computation tasks. Our anomaly detection solution described in WP5 deliverable, Event Detection Engine (EDE), proposes leveraging these resources from heterogeneous compute and network resources utilising algorithmic transprecise adaptation mechanisms.

Transprecise computing states that computation need not always be exact and proposes a disciplined trade-off of precision against accuracy, which impacts computational effort, energy efficiency, memory usage, and communication bandwidth and latency. This approach allows for dynamic adaptation of precision during computation depending on the underlying system context and available resources. In the case of distributed streaming, this adds a new dimension to the problem of scheduling streaming applications and will ultimately lead to superior performance, energy efficiency, and user experience. The experiments described in this section demonstrate the feasibility of this unique approach by developing a transprecise stream processing application framework and transprecision-aware middleware. The use cases for performance anomaly detection, network anomaly detection, and graph processing will guide the research and underpin technology demonstrators as relevant for the EDE platform (part of Service Assurance and Remediation) developed in WP5.

The experiments aim to address three fundamental scientific questions. Firstly, our objective is to establish the scope of transprecision in stream processing applications by developing algorithms capable of trading-off result accuracy with non-functional properties. This task is challenging as not all algorithms, including those based on machine learning, can easily adapt to transprecision methodologies. Secondly, we aim to define appropriate programming abstractions for transprecise streaming applications. These well-defined abstractions will enhance end-user productivity by providing greater control over computation and scheduling policies on edge/cloud systems. Lastly, our goal is to achieve dynamic transprecise-aware mapping of streaming applications on edge/cloud resources. This involves accurately modelling the potential tradeoffs between resources and precision for each application and the operators they utilize.

4.4.1 Detection and Analysis of Energy Consumption

In the following subsection we will describe the overall design and goal of our experiments regarding algorithmic transprecise adaptation. We selected a wide range of ML methods suitable for anomaly detection tasks. Next, we executed several experiments whose main goal was to find, using guided Hyper-Parameter Optimization (HPO), the best hyper-parameters for each model in order to maximise predictive performance. These experiments were carried out during the development of the Service Assurance and Remediation (SAR) component in WP5.

Once we had the best performing hyper-parameters for each ML method we designed a set of experiments to basically benchmark what the energy impact of different training and inference scenarios are for each model. For the measurement of power utilisation we used the Intel developed Running Average Power Limit (RAPL) interface [20]. It is used for reporting accumulated energy consumption for various power domains. Server grade CPUs, largely from the Intel Xeon family (post 2010 Sandy Bridge architecture) are supported. These domains are largely dependent on the CPU being utilised. Figure 93 shows how each monitorable RAPL domains are organised. Package domain contains information regarding CPU cores, Cache memory and integrated GPU (if available). The DRAM domain gathers information regarding the working memory. CPU and Integrated GPU measurements can be fetched using the Core and Uncore domains respectably. An additional domain regarding Nvidia GPUs can be obtained (Starting from Nvidia Volta 2018) using the NVIDIA Management Library (NVML) [21].

Because most of our available HPC cluster is based around AMD processors (mostly AMD EPYC 7702 [22]) not all RAPL domains are supported. In fact in order to utilise RAPL measurements some modifications to the Linux kernel had to be made. However, even so we were able to gather energy consumption data only for the package domain. If there are two CPU sockets on the physical hardware then there are two package domains which need to be monitored (PKG 0, PKG 1). In our case, we have a single socket for our testing server. The complete specifications of the testing infrastructure can be found bellow (Table 24).

Table 24: Experiment infrastructure 16 x HPE Proliant DL385 Gen10

Specification	Description
CPU	128x EPYC 7702 2,0 Ghz/core
RAM	1024 GB
Storage	2x 480 GB SSD
Inter-connect (storage and communications)	2x25GbE adaptors
Operating system	Ubuntu 22.04 LTS (custom kernel)

Initial experiments were carried out using the *perf* [23] Linux tool used for performance analysis. It supports hardware performance counters, tracepoints, software performance counters, and dynamic probes. An example usage scenario can be found bellow:

Table 25: Perf usage scenario

```
perf stat -e ,power/energy-gpu/,power/energy-pkg/ ede.py -f config.yaml
```

The previous example collects energy consumptions (in Joules) for the GPU and Package domains. While utilising *perf* for energy measurements we have found that it provides limited measurements capabilities for our use-case. For example, when we train/validate a predictive model we usually execute some form of cross-validation, while using perf we would require to split up each split into separate perf calls, there is no reliable method to split measurements otherwise.

This is the main reason we decided to instrument our code to provide as precise measurement as possible. For this we used pyJoules [24]. This library allows us to measure specific code fragments. In our case we wished to measure only model training and prediction, ignoring any preprocessing operators. For each cross-validation fold we collect training, inference energy consumption and duration. We still measure the overall power consumption using perf as a form of sanity check.

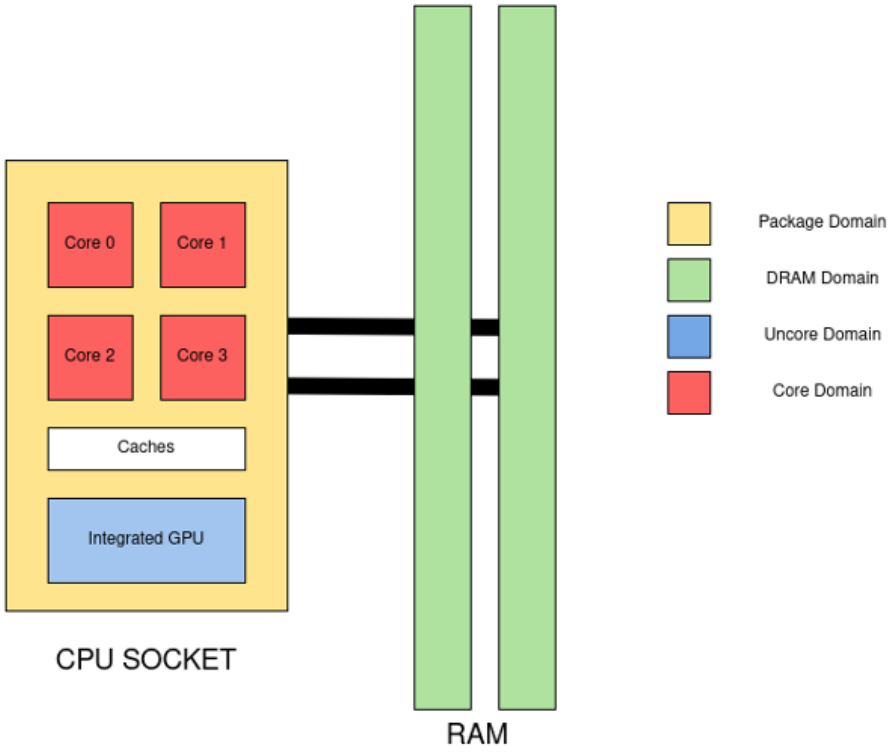


Figure 93: RAPL Domains (according to pyJoules)

4.4.2 Data Set

The dataset used during the testing phase was described in WP5 deliverables. For the sake of completeness we will describe it here as well. The dataset was generated by monitoring a 4 node deployment of Apache Spark on which a data and compute intensive distributed application was being executed. System level metrics were collected with a polling period of 1 second. For each node we collect 89 features. For the experiments detailed in this deliverable we used a single node run, with 4 hardware anomaly classes. Each hardware anomaly was induced using a distributed anomaly induction tool developed by the UVT team:

- CPU Load - Simulates an abnormally high CPU utilisation
- Memory - Simulates both memory allocation issues as well as memory leaks
- DDOT - Simulates CPU ALU and Cache memory issues
- Copy - Simulates persistent storage I/O issues

Figure 94 shows the class distribution of the dataset used for the dataset used. We can see that the anomalous events are vastly underrepresented in the dataset. Furthermore some anomalies have overlapping effects/symptoms DDOT and CPU anomalies have a large CPU component while COPY has a large Memory component.

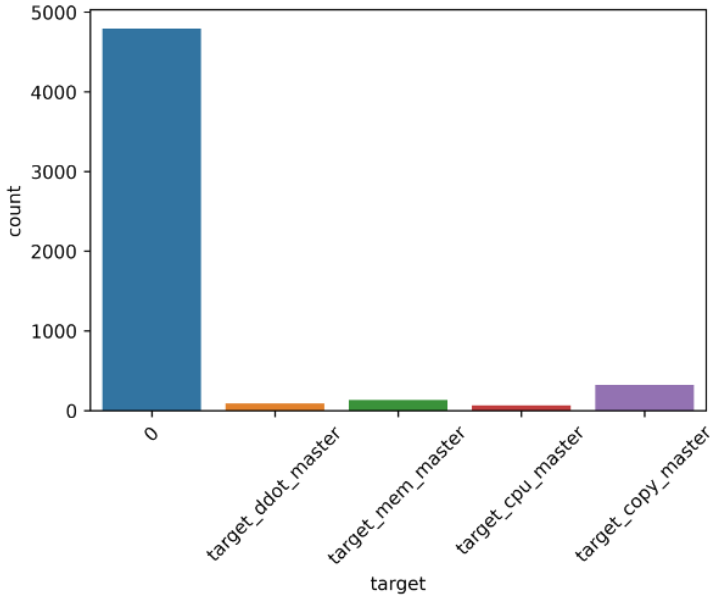


Figure 94: Class distribution

4.4.3 Experiments and Results

We based our experiments around the results obtained during WP5 related experiments. Specifically, we used the best method parameters obtained after an extensive, guided, hyper-parameter optimization of each method detection method. The methods chosen for this experiment are all decision tree based supervised classification methods. These were chosen because they represent on the one hand the most common method types used in these

scenarios and on the other hand are very well suited detection tasks in unbalanced datasets. The methods used are; AdaBoost, RandomForest, XGBoost, LightGBM, CatBoost.

The current experiments can be split up into 3 distinct phases. Each of these phases are designed to fully describe the tradeoff between energy consumption and predictive performance.

The first phase we execute for each algorithm a 10 fold cross-validation utilising stratified method for the splitting of training and validation sets. This enables us to maintain the same class distribution for these sets as the original set. If we would split the data randomly some classes are likely to be underrepresented or even missing from the training or validation sets. This will lead to skewed performance evaluation. The main goal of this phase is to obtain a baseline of performance, seeing that the best hyper-parameters are used during this step.

During the second phase we execute a recursive feature elimination for each method. Training predictive models initially on one feature then adding at each iteration an additional feature until we obtain the complete featurespace. This allows us to check how the input data size affects both power consumption and predictive performance.

Figure 95 and Figure 96 show the results regarding power consumption during training and inference respectively. We can see AdaBoost has an almost linear increase in power consumption while RandomForest and to some extent CatBoost, showcases almost the same data usage, independent of input feature space. XGBoost and LightGBM have relatively noisy power consumption. Out of all of the methods the highest single energy consumption was obtained by XGBoost while the lowest is CatBoost. We should mention that for each iteration a the same 10 fold cross-validation methodology is used, meaning that in total 10 * 89 models were trained during this process. Figure 97 shows the scores obtained for each iteration. We can see that both methods perform well with a relatively small feature space, XGBoost actually performing better.

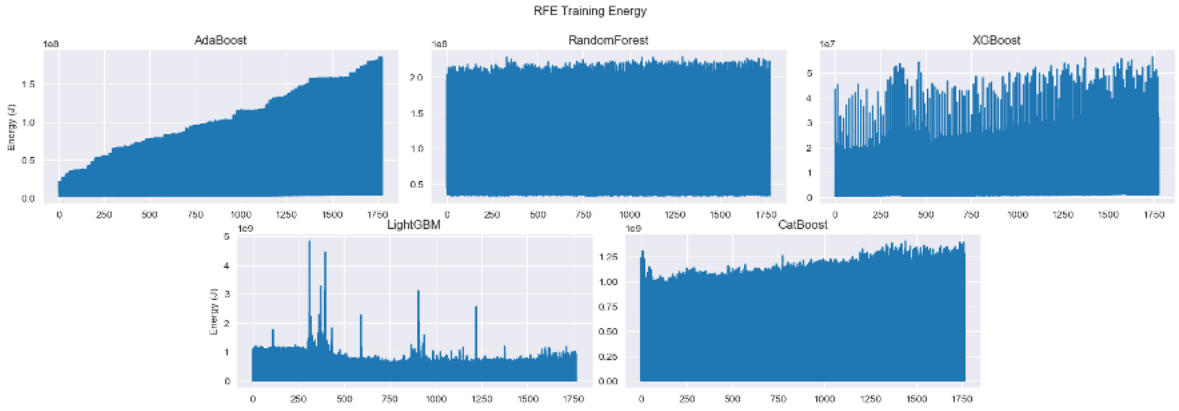


Figure 95: Recursive Feature Elimination - Training

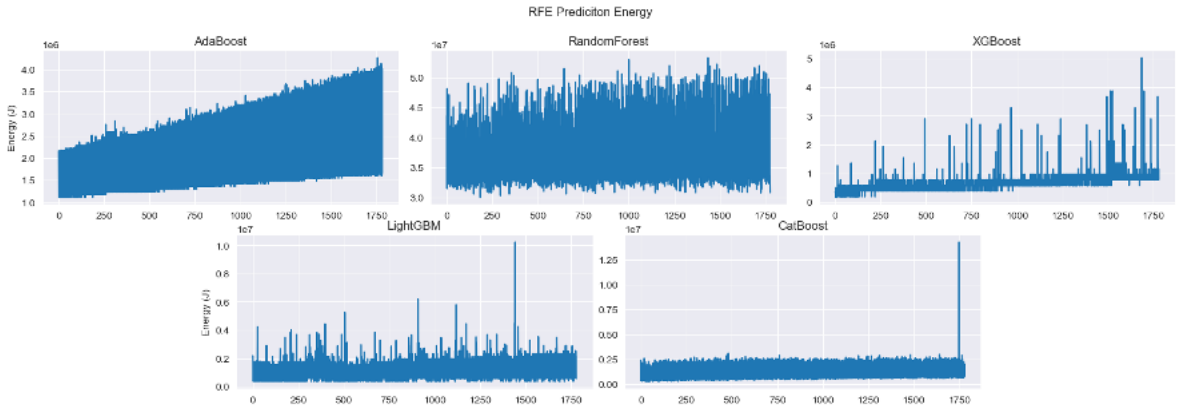


Figure 96: Recursive Feature Elimination - Inference

In phase 3 we take the feature space defined during HPO and see what the impact of each parameter value has on predictive performance and energy consumption. Figure 98 shows the impact of parameter values on energy consumption. We can clearly see that some parameters have a much larger impact than others, a clear pattern can be observed for all 3 methods.

For the sake of brevity and simplicity we will not list the results for each method and its parameters. Instead we will focus on AdaBoost for the simple reason it has the lowest energy consumption out of all methods currently tested. Figure 99 shows energy consumption (left-hand side) and the predictive performance using the F1 score (right-hand side).

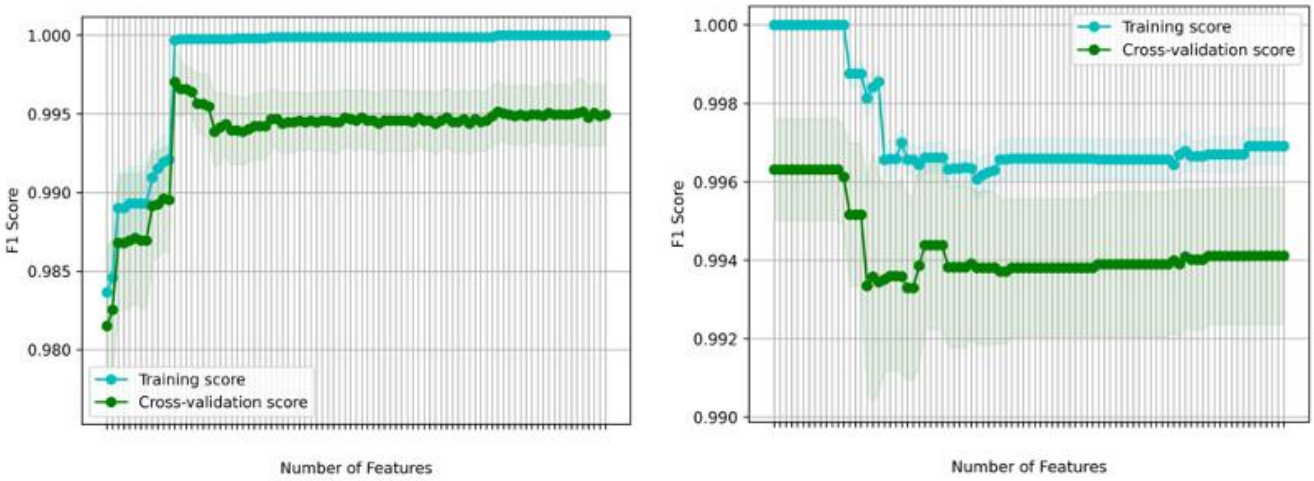


Figure 97: RFE scores for CatBoost and XGBoost

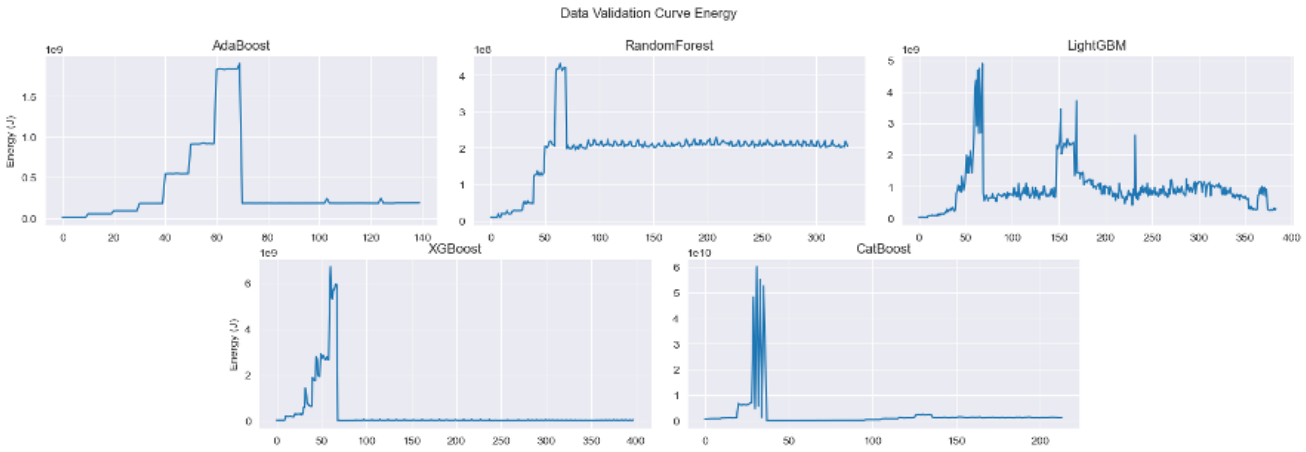
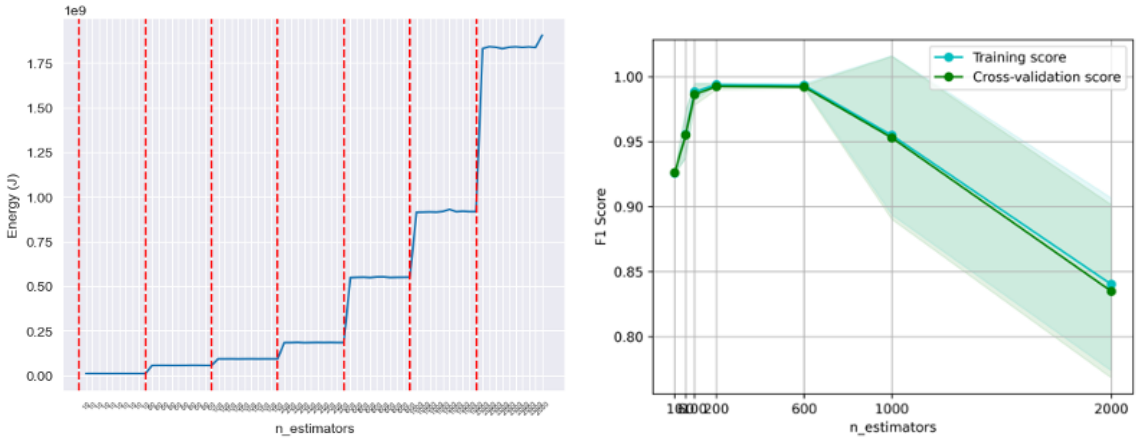


Figure 98: Validation curves for all hyper-parameter values.

We should mention at this point that all experiments were carried out using a fixed random seed to aid in experiment reproducibility. This fixed seed was used for the generation of training and validation splits for cross-validation and for method initialization. One interesting side effect is how some training and validation dataset pairs during cross-validation cause spikes in energy consumption. Specifically, this occurs during training and is linked to some underlying characteristics of the data. The exact cause is not yet completely understood and requires further research. At this point the most probable cause of this might be related to specific events (i.e. rows) and the data distribution and/or entropy.



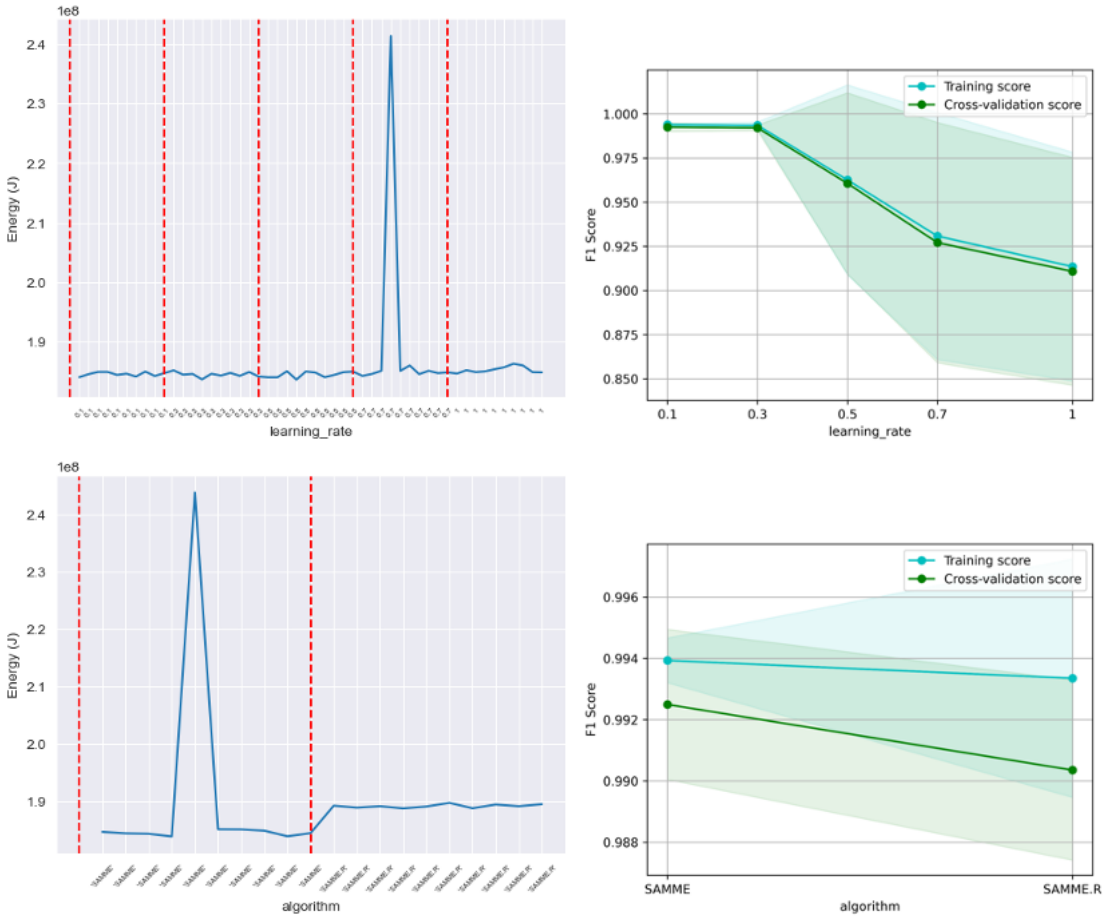


Figure 99: Energy Consumption AdaBoost per parameter value and F1 Score per parameter

Table 26: Classification report for AdaBoost (Fold 8)

classes	pre	rec	spe	f1	geo	iba	sup
normal	0.998	0.998	0.991	0.998	0.995	0.991	959.0
ddot	1.0	1.0	1.0	1.0	1.0	1.0	18.0
mem	1.0	0.923	1.0	0.960	0.960	0.915	26.0

cpu	0.928	1.0	0.999	0.962	0.999	0.99	13.0
copy	0.984	1.0	0.999	0.992	0.999	0.99	64.0
avg/total	0.997	0.997	0.997	0.997	0.997	0.997	0.997

Table 26 provides a classification report in case of AdaBoost, it represents a detailed breakdown using several scoring measures including F1 score separated for each class in part. The “support” column also shows the number of occurrences of each class in the validation data. These kinds of reports are generated for each training fold during cross-validation.

4.4.4 Conclusions and Discussion

The results presented in this section let us create a repository of both pre-trained predictive models and a set of viable hyper-parameters which yield a workable tradeoff between predictive performance and energy consumption. We have also observed a strong correlation between both training and inference times and energy consumption, thus we can conclude that, at least for the methods listed here, a lower energy consumption leads to faster inference times.

We have HPO related results for several deep learning methods however, seeing that they require specialised hardware to execute the experimental results are not yet complete as of writing this deliverable. Similarly, we are currently working on extending the experimental work done with unsupervised detection methods to include energy consumption metrics. For both scenarios we further aim to include additional transprecision optimizations, especially in the case of deep neural network-based models where mixed precision and post training optimizations such as weight quantization and clustering are known adaptations.

An extended version of the results summarised here are to be published in a journal article in the upcoming months by the UVT team.

5 Seamlessly Integration of Heterogeneous Architectures for Improving Developers' Productivity in HW/SW Co-design of Data-intensive Applications

The SERRANO platform aims to provide a set of tools and design methodologies that will ease the designers to develop performance and power optimised solutions for heterogeneous computing platforms deployed at the edge and the cloud. In today's era, the demands for high-performance computations and low-power designs are continuously escalating. To meet the user requirements for energy efficiency, high throughput, and security in this new computing paradigm, the concept of adaptive computing has emerged. Acceleration platforms such as GPUs and FPGAs offer higher performance than most conventional processing systems, while still fulfilling the user requirements for energy efficiency and security. However, the efficient design and deployment of computationally intensive applications on these specialised compute units can often be unclear, even for the most experienced developers. This challenge is further compounded by the need to expedite the development cycle and design solutions within limited timeframes. In the context of the SERRANO project those issues are addressed by designing FPGA and GPU accelerators through the Plug&Chip framework.

The components that realise the Plug&Chip framework that facilitates the FPGA and GPU developers to speed-up the development cycle by utilising a set of tools is described in this section. Namely those tools are:

1. A tool for the automatic optimization of FPGA accelerated kernels.
2. A design methodology for the design of memory efficient FPGA accelerators.
3. A tool for the automatic optimization of CUDA kernels.

5.1 Automatic Optimization for FPGA Accelerated Kernels

High-Level Synthesis simplified the hardware development process by allowing developers to instruct the compiler how to perform synthesis by adding directives to a C/C++ or OpenCL source code. However, manually selecting the appropriate directives is an extremely difficult task even for experienced designers, mainly because of **a)** the huge decision space and **b)** the inherent interdependence with the underlying FPGA architecture. The lack of end-to-end tools that provide optimised HLS configurations in an automated manner is therefore one of the major obstacles to realising the FPGA Automatic Code Deployment vision [25]. SERRANO fills the gap with *GenHLSOptimizer*, an end-to-end tool for automatically optimising C/C++ kernels with respect to the underlying architecture of the target FPGA.

5.1.1 GenHLSOptimizer: A Genetic Algorithm-based Optimizer for High-Level Synthesis

SERRANO offers a tool that automatically optimises synthesizable C/C++ kernels for Xilinx FPGAs through High-Level Synthesis. This optimization scheme identifies points of interest, i.e., loops and arrays, applies directives (e.g., loop unrolling, array partition), and performs synthesis to get the latency and resource utilisation. By applying different combinations of directives, the optimizer proposes an approximation to the Pareto-optimal designs with respect to the underlying architecture of the target device.

Due to the large design space, the DSE is performed using the algorithm NSGA-II [26]. which is implemented in the Python Multi Objective Optimization (PyMOO) library [27]. The goal of the Genetic Algorithm (GA) is to minimise the latency and area of the design, taking into account the resource constraints of the target FPGA. The population size and offspring of the NSGA-II algorithm are set at *40*. We also configure the Genetic Algorithm with the random operator for sampling and selection, the simulated binary operator for crossover, and the polynomial operator for mutation. All operators are configured with the default parameters of the library. The termination criterion was set at *24* generations.

The exploration phase consists of the following steps: **a)** the configuration population is initialised, **b)** each configuration of the current population is applied to the source code using a source-to-source compiler and the output is synthesised using the Xilinx Vitis tool chain [28], and **c)** the synthesis outputs of the population are passed to NSGA-II to build the next generation configurations. Steps **b)** and **c)** are executed iteratively until the termination criterion is reached. Configurations that result in designs that exceed the available resources of the target FPGA architecture or require an unreasonable amount of time for the synthesis process (*1h*) are marked as infeasible so that GA can produce feasible solutions when the algorithm converges. Assuming that the designs of one generation are evaluated in parallel, the *1h* threshold for time-consuming synthesis, and the termination criterion, the near-optimal designs are generated in *24h* in the worst case. Readers can find more information in deliverable D4.3, which analyses the individual components of the proposed methodology.

5.1.2 Evaluation

We evaluate GenHLSOptimizer with C/C++ kernels provided by SERRANO. In particular, we use the following kernels: (a) the Black-Scholes algorithm (INB), (b) the Kalman filter (INB), (c) the Savitzky-Golay filter (INB), (d) the Wavelet Transform (INB), (e) the Dynamic Time Warping Distance Calculation (IDK), and (f) the Encoding part of Erasure Coding (CC).

For the synthesis of the studied kernels, we use Xilinx Vitis HLS 2021.1, a State-of-the-Art framework capable of synthesising source codes for edge and cloud devices. We target the Xilinx Alveo U50 and MPSoC ZCU104 FPGAs available on the SERRANO platform. The target clock frequency was set to 300 MHz. The genetic algorithm was implemented using the

PyMOO library (version 0.5.0). Finally, our experiments were performed on an Intel Xeon Gold 5218R (@2.10GHz) server with 256GB RAM memory.

We compare the design provided by GenHLSOptimizer with **i)** the design provided by the kernel without directives (Vitis), which highlights the impact of the optimizations provided natively by Vivado-HLS, and **ii)** the design provided when the source code is optimised by the developers of AUTH. **iii - iv)** We synthesise each of these baselines with/without enabling Vitis' default HLS optimizations, i.e. *config_compile -pipeline_loops*, *config_unroll -tripcount_threshold* and *config_array_partition -complete_threshold*, creating 4 baselines in total.

5.1.2.1 Optimised Design Latency

Figure 100 shows the relative mean speedup of GenHLSOptimizer compared to each baseline for the Xilinx Alveo U50 and MPSoC ZCU104 FPGAs. Our solution outperforms all baselines. It achieves an average relative speedup of 425.35x and 2.43x compared to the Vitis and AUTH versions, respectively, and an average relative speedup of 5.17x and 1.13x when these versions are optimised with Vitis' HLS optimizations for the Xilinx MPSoC ZCU104 FPGA. The same picture emerges for the Xilinx Alveo U50 FPGA, where GenHLSOptimizer achieves an average relative speedup of 234.53x and 2.4x compared to the Vitis and AUTH versions, and an average relative speedup of 5.17x and 1.1x when these versions are optimised with Vitis' HLS optimizations.

An interesting observation is that GenHLSOptimizer is able to achieve higher relative speedup on average on the MPSoC ZCU104 compared to the Alveo U50, especially for the Vitis baseline. This is due to the limited resources of the Edge compared to the Cloud device. In particular, when an application is synthesised for a resource-constrained edge FPGA, the final design is likely to have higher latency than the design for a cloud FPGA equipped with more Block RAMs, DSPs, Flip-Flops, and Look-Up Tables, especially if no specific optimizations are instructed via HLS directives.

Figure 101 shows the relative speedup per application for both FPGAs to provide insight into how our approach optimises each kernel. Note that GenHLSOptimizer is able to achieve higher relative speedup for each application when using the MPSoC ZCU104 compared to Alveo U50 for the Vitis baseline.

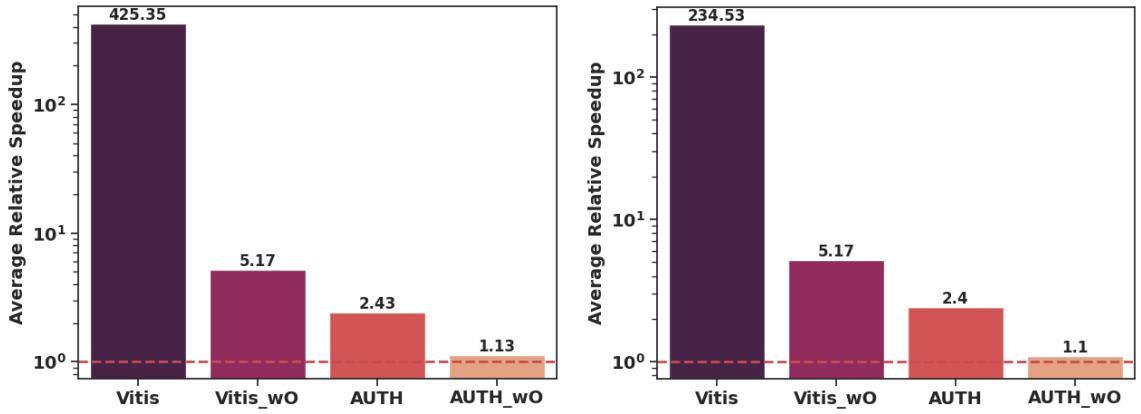


Figure 100: Average relative speedup for Xilinx Alveo U50 (Right) and MPSoC ZCU104 (Left)

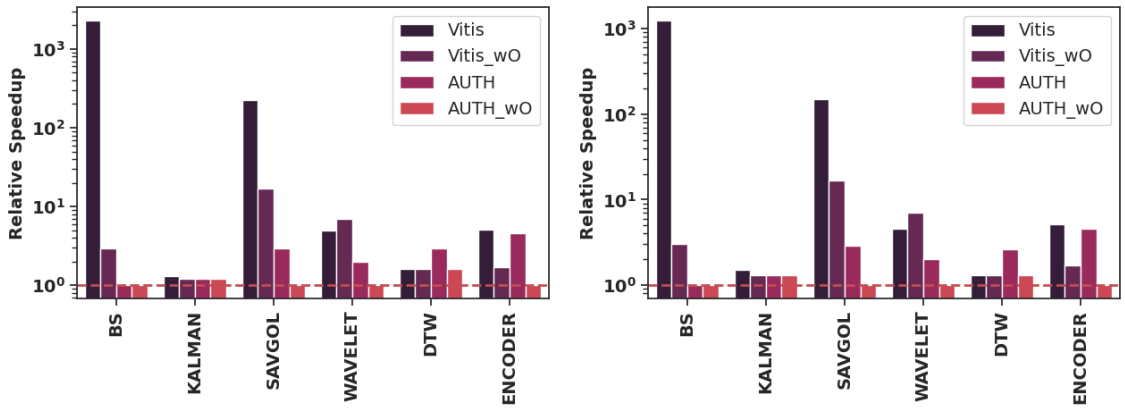


Figure 101: Relative speedup per application for Xilinx Alveo U50 (Right) and MPSoC ZCU104 (Left)

5.1.2.2 Design Space Exploration Time

GenHLSOptimizer uses a meta-heuristic to determine the approximate Pareto frontier. Therefore, the time to perform the Design Space Exploration is an important aspect of our approach. Figure 102 shows the time it takes GenHLSOptimizer to create the optimised design for each application for both FPGAs. Our approach takes an average of 11.9 hours and 7.1 hours to create the optimised designs for the MPSoC ZCU104 and Alveo U50 FPGAs, respectively. Average design space exploration times are below the 24-hour threshold, highlighting the system's ability to converge much faster. We can also see that the average DSE time for the Alveo U50 FPGA is 1.68x lower than that of the MPSoC ZCU104. This also happens due to the resource limitations of the edge FPGAs. Since the directive configurations are initialised randomly, it is more likely that a directive combination will result in an infeasible design on a resource-constrained edge device than on a cloud device. This complicates the work of the genetic algorithm to find the approximate Pareto frontier and consequently requires more time.

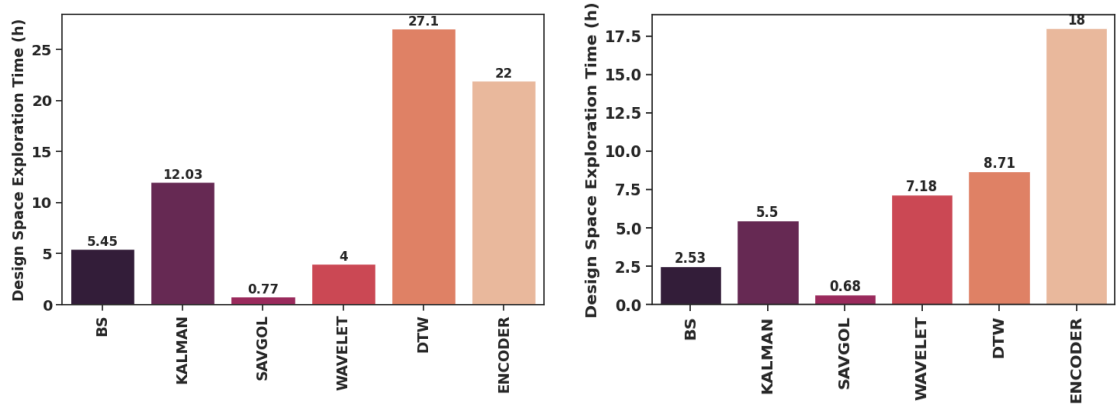


Figure 102: DSE time per application for Xilinx Alveo U50 (Right) and MPSoC ZCU104 (Left)

5.1.2.3 Latency and Resources Distributions

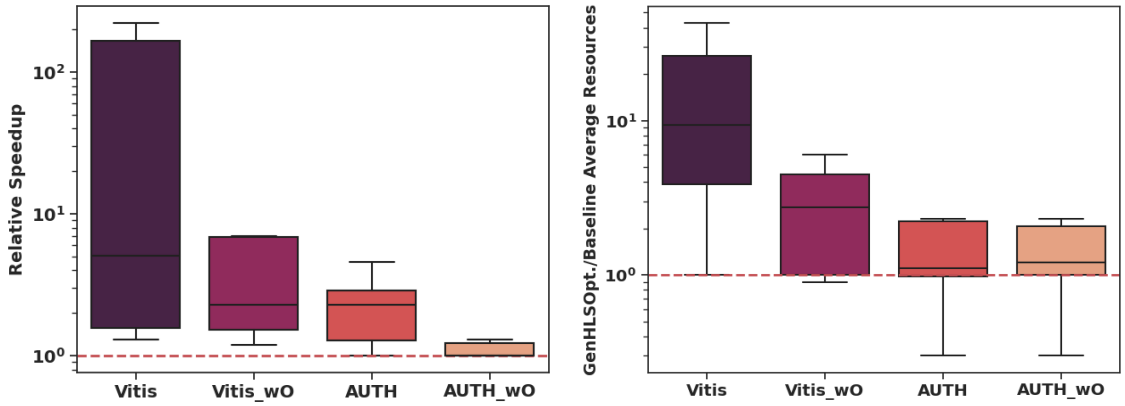


Figure 103: Relative Speedup (Left) and Average Resources Utilisation (Right) distributions

Figure 103 shows the distribution of relative speedup and resource utilisation for the FPGAs studied for all baselines. GenHLSOptimizer provides lower latency designs compared to all baselines by efficiently utilising the available resources of the target FPGA, as shown by the average resource utilisation distribution of each baseline. Finally, as described in deliverable D4.3, our approach can always produce a feasible design in terms of resources, which is an important differentiator from the Vitis_wO baseline that cannot always produce feasible designs.

5.1.3 Conclusion

In this section, we present GenHLSOptimizer, an end-to-end tool for optimising C/C++ kernels with respect to the underlying architecture of the target FPGA without human intervention. Our experimental evaluation shows that our approach is able to outperform all studied baselines in terms of latency, considering the resource constraints of an edge and cloud FPGA available on the SERRANO platform.

5.2 Dynamic Memory Management in High-Level Synthesis (HLS)

SERRANO offers a tool and a co-design methodology for the implementation of memory efficient many-accelerator platforms on Xilinx FPGAs. This tool [29] [30] is publicly available through the SERRANO's repository [31].

5.2.1 Many-accelerators Platforms in HLS

The implementation and parallel execution of many accelerators on a single FPGA device has been suggested as a candidate approach for increasing the application's throughput [32]. This is an execution scheme that allows multiple accelerators to process data batches in parallel leading to a significant decrease in the overall execution latency. However, this execution scheme implies the synthesis and implementation of multiple circuits on the same FPGA platform which may not be feasible on small platforms with limited computational and memory resources. Studies [33] have shown that the rapid saturation rate of the platform's on-chip memories is the primary reason that makes the many-accelerator schemes not feasible.

Dynamic memory management in HLS allows accelerators to share and reuse on-chip memory resources at their execution time. In those design methodologies heap structures are implemented on the FPGA platform for the dynamic memory allocation and deallocation. Figure 104 shows the number of the erasure-coding encoder accelerators that can be implemented in the Xilinx MPSoC ZCU104 FPGA platform when the typical static and the dynamic memory sharing schemes are used. While the static allocation methodology can deliver only up to three encoder accelerators before over-utilizing the platform's memory resources, the dynamic memory schemes can deliver up to 28 parallel executed encoder accelerators instead.

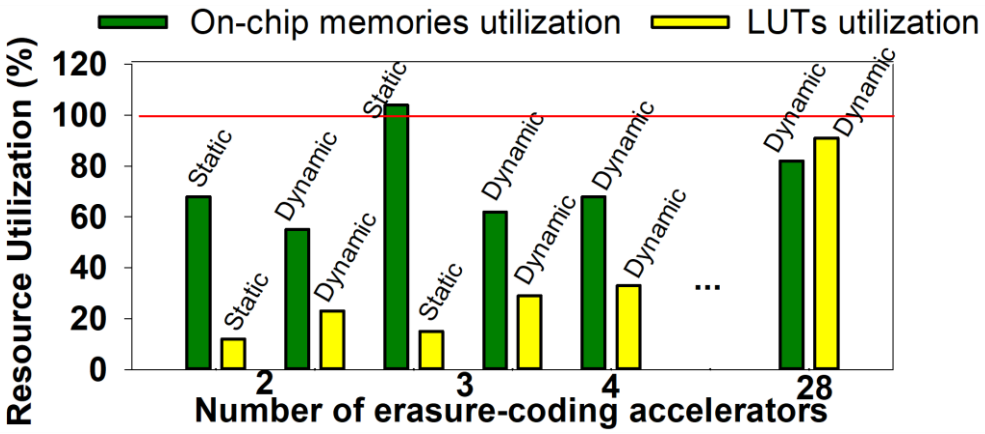


Figure 104: Dynamic memory allocation for erasure coding encoder accelerators

5.2.2 On-chip Defragmentation Methodology

Despite the benefits of the dynamic allocation methodology, as the number of implemented and parallel-executed accelerators increases, there is a higher likelihood of encountering heap fragmentation issues. These issues can result in memory allocation failures (MAFs) and consequently lead to stalls and exceptions in the execution of the accelerators. Figure 105 [34] demonstrates this phenomenon. As the number of synthesised and parallel executed K-means accelerators rises the percentage of the MAFs due to heap fragmentation also increases. Therefore, a design methodology that nullifies this fragmentation induced MAFs is proposed for the design of the SERRANO's memory-shared accelerators.

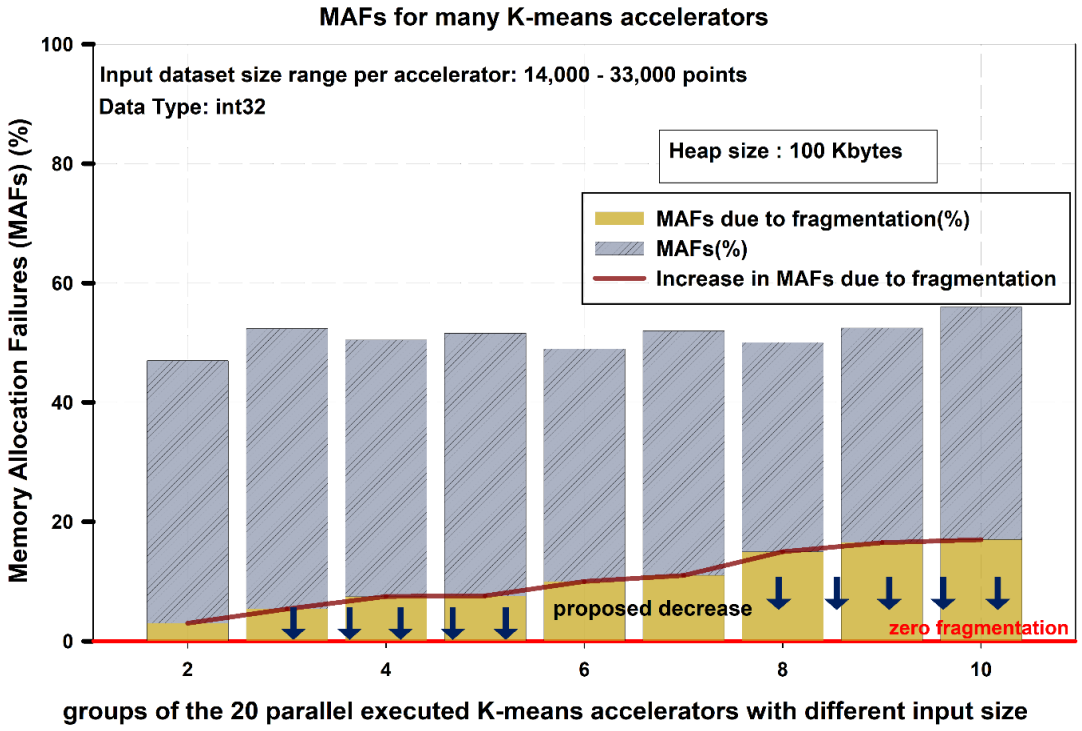


Figure 105: Memory allocation failures due to execution of multiple K-means accelerators

The proposed methodology consists of two stages:

- An offline analysis stage where the conditions that enable the mechanism that optimises the heap usage are determined.
- An online execution stage where a garbage collection mechanism is executed.

Figure 106 illustrates this design flow.

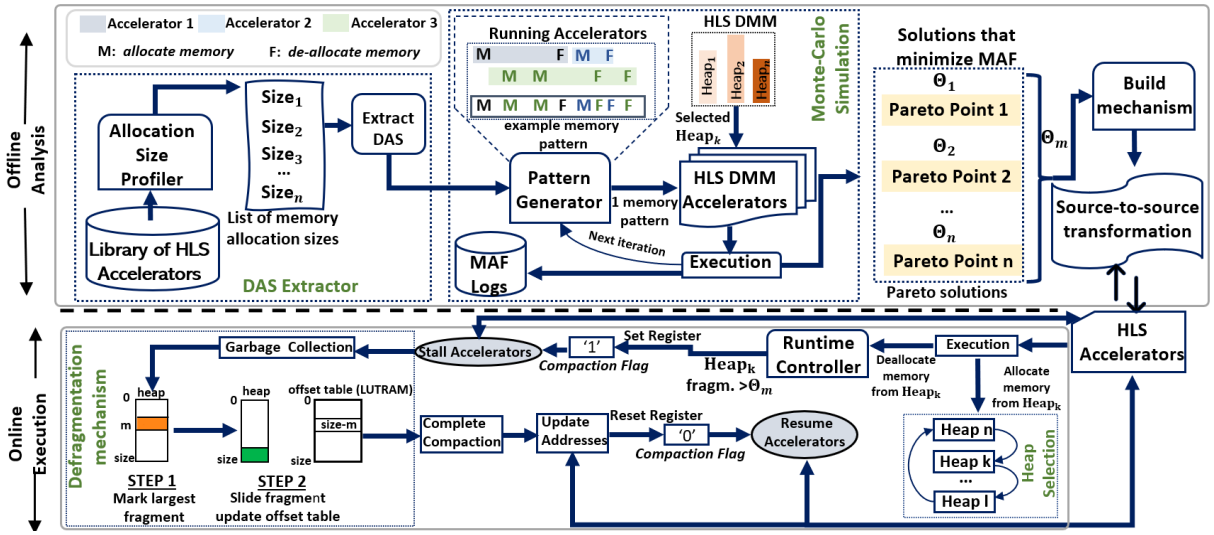


Figure 106: Design flow for on-chip defragmentation methodology

The offline analysis phase aims to determine the heap's fragmentation ratio Θ that triggers the FPGA designed garbage collection mechanism. Initially, the memory allocation sizes from all the parallel executed HLS accelerators are extracted and their different values form the Distinct Allocation Sizes (DAS). Then, a Monte-Carlo simulation analysis is performed to compute the fragmentation ratios Θ that minimise the allocation failures. The input to this simulation is memory patterns that correspond to an overlapping execution of those accelerators, as these are derived from the pattern generator block. This block pseudo-randomly generates Malloc/Free sequences of the extracted DAS, forming multiple memory patterns.

In this simulation the HLS accelerators and the HLS garbage collector are executed in a software emulation mode. The output of this offline simulation is the Pareto front that trades-off decrease in fragmentation induced memory allocation failures to estimated execution latency. The designer selects the Pareto solution Θ_m that meets their requirements for execution latency and memory fragmentation and synthesises on the FPGA the garbage collector for the specific n Θ_m parameter.

During the online execution stage, the accelerators are executed on the FPGA platform on a shared heap and the garbage collector is executed every time that the heap's fragmentation ratio exceeds the user-defined Θ_m threshold. Details on the implementation of the garbage collector on the FPGA can be found in the corresponding publications [35] [36].

5.2.3 Evaluation

The experimental analysis that is presented in this subsection shows the effect of different Θ values in reducing the heap's fragmentation when multiple K-means and multiple moving average filters (i.e., an almost identical kernel with the Savitzky-Golay and Wavelet filters) accelerators are executed using a dynamic memory allocation scheme and share one heap.

Figure 107 shows that the higher the Θ value, the more likely fragmentation-induced MAFs are to occur. As Θ increases, the MAFs approach the reference solution (i.e., the one without an on-chip garbage collector). Therefore, to design a fragmentation-free shared memory system, the Θ should be set as low as possible. However, as it is depicted in Figure 108, the lower the Θ value, the higher the execution latency. This happens due to the frequent executions of the garbage collector that cause frequent stalls at the accelerator's execution.

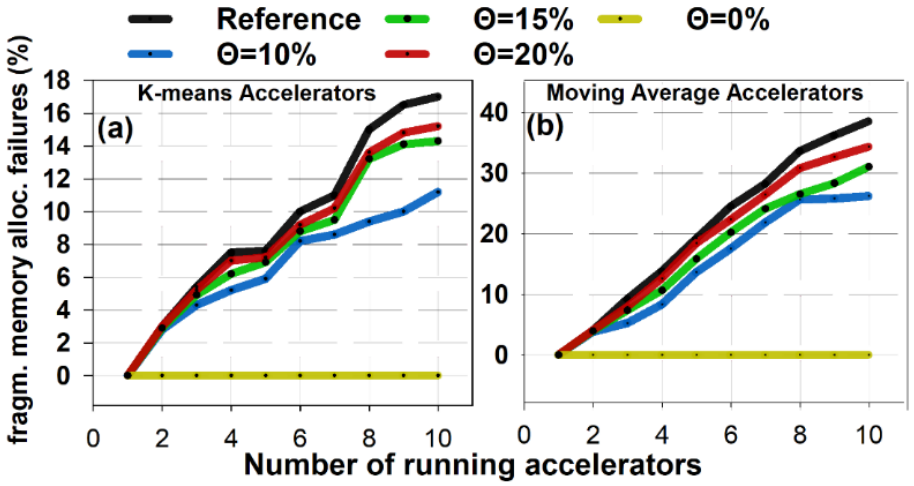


Figure 107: Allocation failures for different Θ thresholds

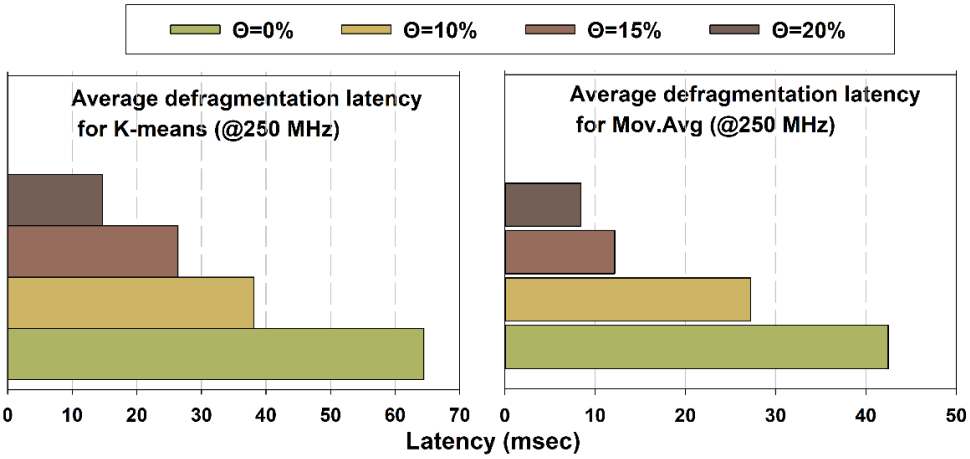


Figure 108: Defragmentation latency for different Θ thresholds

5.2.4 Conclusion

In this section an HLS co-design methodology and the corresponding framework were presented. This tool allows the HLS designers to develop many-accelerator solutions with controllable fragmentation of the shared on-chip memories. The evaluation analysis shows that it is up to the designer to select the optimal solution that is generated from the offline Monte-Carlo analysis that will fulfil their requirements for memory efficiency and performance.

5.3 Automatic Optimization for CUDA Kernels

An automatic optimization tool was developed, targeting to produce accelerated and energy efficient, HW aware kernels in an automated manner. Generally, the GPUs' programmability has been improved the last few years and the kernels' implementation and execution process has been simplified, but the task of optimising the kernels targeting to achieve close-to-close peak performance remains complex and time consuming, due to the fact that the variety of kernels and different GPU architectures keeps increasing.

To this end, we developed an automatic kernel optimization tool. The developed tool implements an automatic optimization process for the block coarsening transformations across different applications, workload input sizes and GPU architectures. The tool is machine learning based (uses regression models) and it also consists of an in house source-to-source compiler. The tool was tested on Polybench benchmark on 5 different devices and was able to achieve speedups up to x2.3 in terms of performance for unseen GPUs and unseen CUDA kernels in comparison with native implementations.

As mentioned the main optimization task of the tool is the block coarsening transformation. The term block coarsening transformation refers to the number of blocks' reduction, leaving the block size (threads per block) the same. To succeed this reduction, multiple neighbouring blocks' work loads need to be merged in order to deal with problems associated with extensive fine-grained parallelism. Generally, blocks are mapped to SMs (multiprocessors) from the GPUs and threads are organised in wraps at CUDA cores. Thus, adopting block coarsening reduction, will also reduce the number of wraps scheduled by each SM, as the SMs' workload is reduced.

In order to implement the block coarsening transformation, the CUDA code needs also to be transformed. It has to be mentioned that the whole process, if it is manually implemented, is in most cases a lot more complex and time consuming and thus, leads to suboptimal kernels' block reductions.

For this purpose, the developed tool includes a source-to-source compiler-tool based on some rules, as depicted in Figure 109. Based on these rules, the tool automates the transformation process for all the tested uses-cases and for new unseen kernels.

initial kernel	block coarsened kernel
—	for (int index=0; index¡bc; index++)
blockIdx.x	bc×blockIdx.x + index
gridDim.x	bc×gridDim.x

Figure 109: A source-to-source compiler-tool based on rules

To implement the block coarsening transformation, a PERL script was developed. The above script can apply block coarsening transformation with a given coarsening factor, on every CUDA kernel. Also, an extra exploration part of all the possible coarsening factors is

implemented and finally, the tool picks the optimal factor, in terms of performance. Both the host and the device programs' source codes are inputted to the tool in order to auto-tune the new grid size and the device kernel located in host and device files respectively.

As mentioned before, the purpose of the tool is to automatically select the more efficient block coarsening transformation. After all the candidate kernels were implemented from the PERL script, supervised learning was used to predict each kernel's version performance and then select the more efficient.

In order to train the unsupervised learning model a dataset that could represent a big variety of CUDA programs, various input sizes and different architectures, should be used. The input features included, a static features part, that represented the structure and the body of the CUDA kernel and a Hardware features part that described the GPU architecture, the potential block coarsening factors and the size of the kernel's input.

We took advantage of a work that was published in 2019 (Guerreiro) [37], that automatically extracts features from PTX files by extracting the number of occurrences of each different instruction per GPU kernel. Also, to automatically convert the CUDA kernels to the PTX assembly file we used a python interface. Finally, the input format included an 101-size vector that counts the kernel's number of 101 different representative PTX instructions and extra different architectural features. The next Figure 110 introduces the adopted GPU specification for both computation and memory description of each GPU. The final feature was a 114-size, 1-D vector that also included the workload input size of the input vector and the block coarsening factor of the kernel's version.

compute specs	SMs cores/SM GPU frequency (MHz)
memory specs	global memory (GB) mem. bandwidth (GB/sec) memory frequency (MHZ) shared memory / SM (KB) L1 Cache/SM (KB) L2 Cache (MB) memory bus (bit) register file /SM (KB)

Figure 110: Adopted GPU specification for both computation and memory description

Figure 111 represents analytically the training and the prediction process. As depicted for the training process, a regression model was adopted to be the Machine Learning predictor. More specifically, four different regression models were trained and tested and the best one was selected. The candidate models were, Simple Linear Regression, Decision Tree Regression, Random Forest Regression and Extreme Gradient Boosting (XGBoost) through the Scikit-Learn machine learning framework. The final one, XGBoost was able to reach the best MSE and R2 score on the test set. Also, as Figure 111 depicts, for the prediction, after the training process the tool is ready to predict the latency of a given unseen kernel, block coarsening factor and

specific GPU and then to decide the optimal block coarsening factor for each kernel on a given architecture.

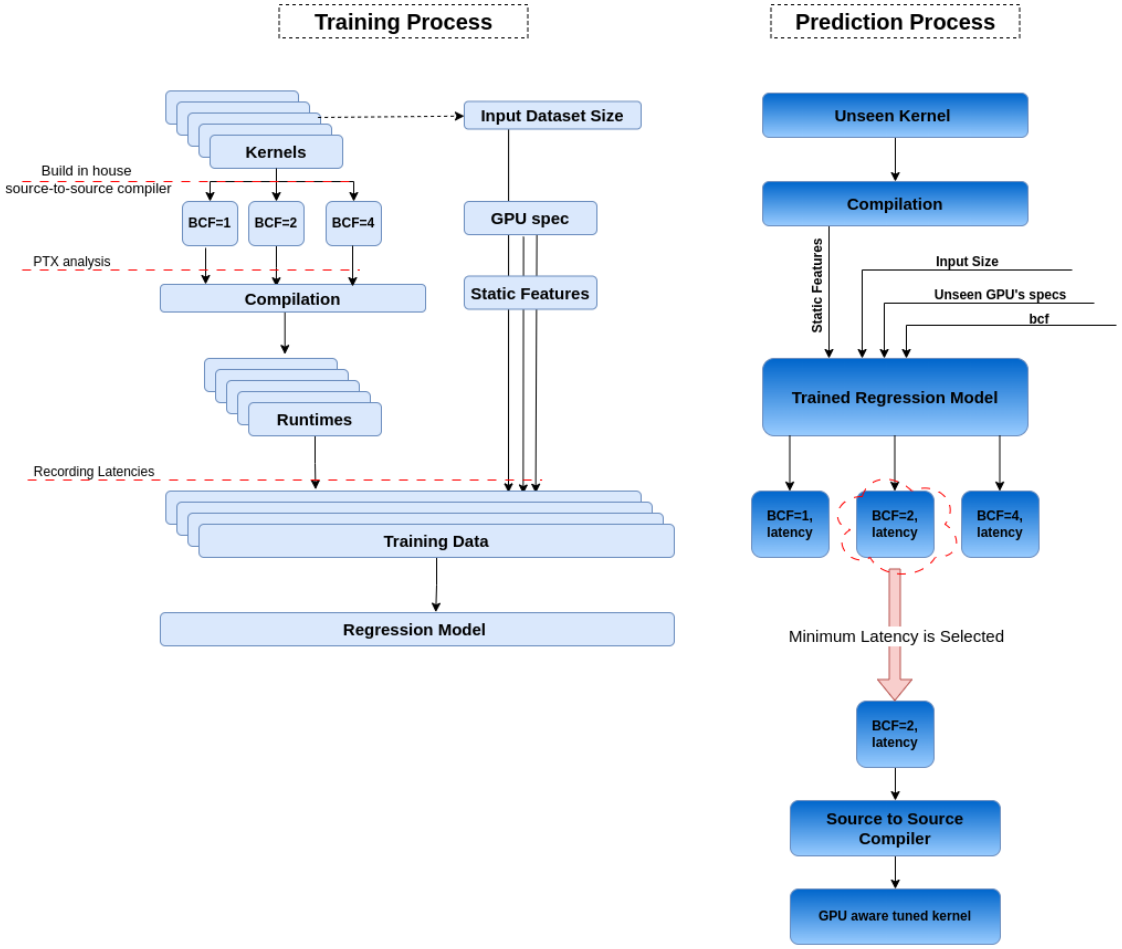


Figure 111: Training and the prediction process

For evaluation purposes of the introduced tool, the Polybench-ACC open suite [38] was adopted. Polybench-ACC contains different CUDA kernels. We run the different CUDA kernels to five different NVIDIA GPUs in order to measure their execution latencies for different platforms. The used GPUs include TX1, Xavier NX, Xavier AGX, GTX 1070 and V100. For more information about the datasets and the GPUs please also refer to deliverable D4.3. It has to be mentioned that 90% of the Polybench-ACC dataset was used for training and the rest 10% for testing purposes.

The next Figure 112 represents the experimental results, MSE and R2 score for the different regression models that we tested on the dataset. The best one was XGB and was able to achieve 0.02 MSE and 0.88 R2 and therefore constitutes our selected regression model. Also, it was able to create optimal coarsened kernels with speedups up to x2.3, for new unseen GPUs and new unseen kernels, in comparison with native implementations.

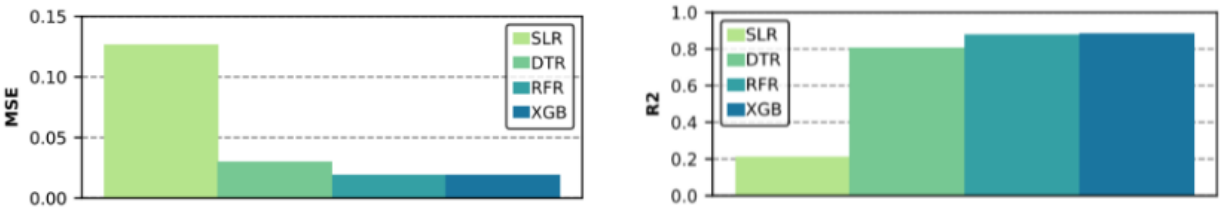


Figure 112: Experimental results, MSE and R2 score for the different regression models

6 Hardware Acceleration for Serverless Workloads

During the past decade, there has been a shift in terms of the ownership of the hardware resources on which applications are being deployed. Increasingly, application execution is being delegated to infrastructures outside the organisation of the application owner to reduce costs related to the ownership, operation, maintenance, and deployment of software.

This paradigm dramatically changed how we develop, package, and deploy applications. Migrating application execution to public cloud infrastructures means our code will run side-by-side with the code of other platform tenants. To tackle issues related mainly to security (we want our code and data to be safe from potentially malicious users running on the same system) but also resource allocation (we would like to avoid a single user hogging all the resources of the underlying system), our applications run inside Virtual Machines or containerized environments which provide different degrees of isolation.

In this environment, the underlying system, i.e., the hypervisor or the container runtime, monitors and restricts the user application from accessing resources they do not own. However, neither of those systems can control the access to hardware acceleration devices with the same granularity or isolation guarantees as they can with other resources such as CPU, Network, or Storage.

The problem is exacerbated by the way we program hardware accelerators nowadays. Such devices typically provide hardware drivers and APIs, which they expose to application developers. These APIs are device-specific, and sometimes they are incompatible even across devices of the same vendor. This has two significant side effects: On one hand, user application implementations end-up being device-specific, hindering portability and programmability, whereas, on the other hand, the lack of uniform APIs across devices renders it extremely difficult to virtualize them in an abstract and efficient way.

In SERRANO, we introduce vAccel [39], a framework that enables virtualized workloads to access hardware accelerators securely and efficiently. vAccel is addressing this situation in two ways. Firstly, it enables the development of hardware-independent applications by logically separating an application into two parts: (i) the user code, which is part of the application logic itself, and (ii) the hardware-specific code, which is part of the application that runs on a hardware accelerator. Second, it enables hardware acceleration within virtualized guests by employing an efficient API remoting approach at the granularity of function calls to delegate accelerable code in a vAccel agent on the host system.

The vAccel software framework has been described in detail in D4.3 (M15). Additionally, OpenFaaS is also described in the deliverable above. In the following sections, we provide a

brief overview of the software stack, along with the developments of porting the various SERRANO kernels to vAccel and OpenFaaS.

6.1 vAccel

vAccel enables workloads to enjoy hardware acceleration while running on environments that do not have direct (physical) access to acceleration devices.

The design goals of vAccel are:

1. **portability**: vAccel applications can be deployed in machines with different hardware accelerators without re-writing or re-compilation.
2. **security**: A vAccel application can be deployed, *as is*, in a VM to ensure isolation in multi-tenant environments. QEMU [40] AWS Firecracker [41]and Cloud Hypervisor [42] are currently supported
3. **compatibility**: vAccel supports the OCI container format through integration with the Kata containers [43] framework [downstream].
4. **low-overhead**: vAccel uses a very efficient transport layer for offloading "accelerate-able" functions from inside the VM to the host, incurring minimum overhead.
5. **scalability**: Integration with k8s allows the deployment of vAccel applications at scale.

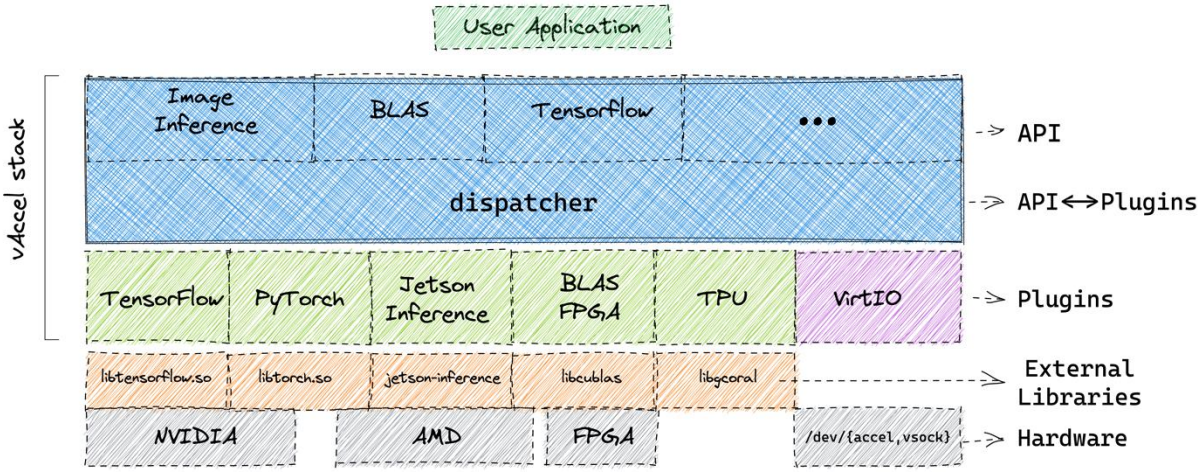


Figure 113: vAccel software stack

The core component of vAccel is the vAccel runtime library (vAccelRT). vAccelRT is designed in a modular way: the core runtime exposes the vAccel API to user applications, and dispatches requests to one of many *backend plugins*, which implement the glue code between the vAccel API operations on a particular hardware accelerator.

The user application links against the core runtime library, and the plugin modules are loaded at runtime. This workflow decouples the application from the hardware accelerator-specific parts of the stack, allowing for seamless migration of the same binary to different platforms with different accelerator capabilities without recompiling user code.

6.1.1 Virtualization Abstraction

Hardware acceleration for virtualized guests is, still, a real challenge. Typical solutions involve device pass-through or paravirtual drivers that expose hardware semantics inside the guest. vAccel differentiates itself from these approaches by exposing coarse-grain "accelerate-able" functions in the guest over a generic transport layer.

The semantics of the transport layer are hidden from the programmer. A vAccel application that runs on baremetal with an Nvidia GPU can run *as is* inside a VM using our appropriate *VirtIO* backend plugin.

We have implemented the necessary parts for our VirtIO driver in our forks of QEMU [44] and Firecracker [45] hypervisors.

Additionally, we have designed the above transport protocol over sockets, allowing vAccel applications to use any backend, if there is a socket interface installed between the two peers. Existing implementations include vsock and TCP sockets. Any hypervisor supporting `virtio-vsock` can support vAccel.

6.1.2 Container Runtime Integration

To facilitate the deployment of vaccel-enabled applications, we integrate vAccel to a popular container runtime, kata-containers [46].

Kata Containers enable containers to be seamlessly executed in sandbox Virtual Machines. Kata Containers are as light and fast as containers and integrate with the container management layers while also delivering the security advantages of VMs. Kata Containers is the result of merging two existing open-source projects: Intel Clear Containers and Hyper runV.

vAccel integration to kata comes in both modes: virtio and vsock. An overview of the software stack is shown in Figure 114.

Our current downstream implementation for Kata-containers v3 includes support for both the AWS Firecracker sandbox and their custom, tailor-made Dragonball backend, using the vsock mode of vAccel.

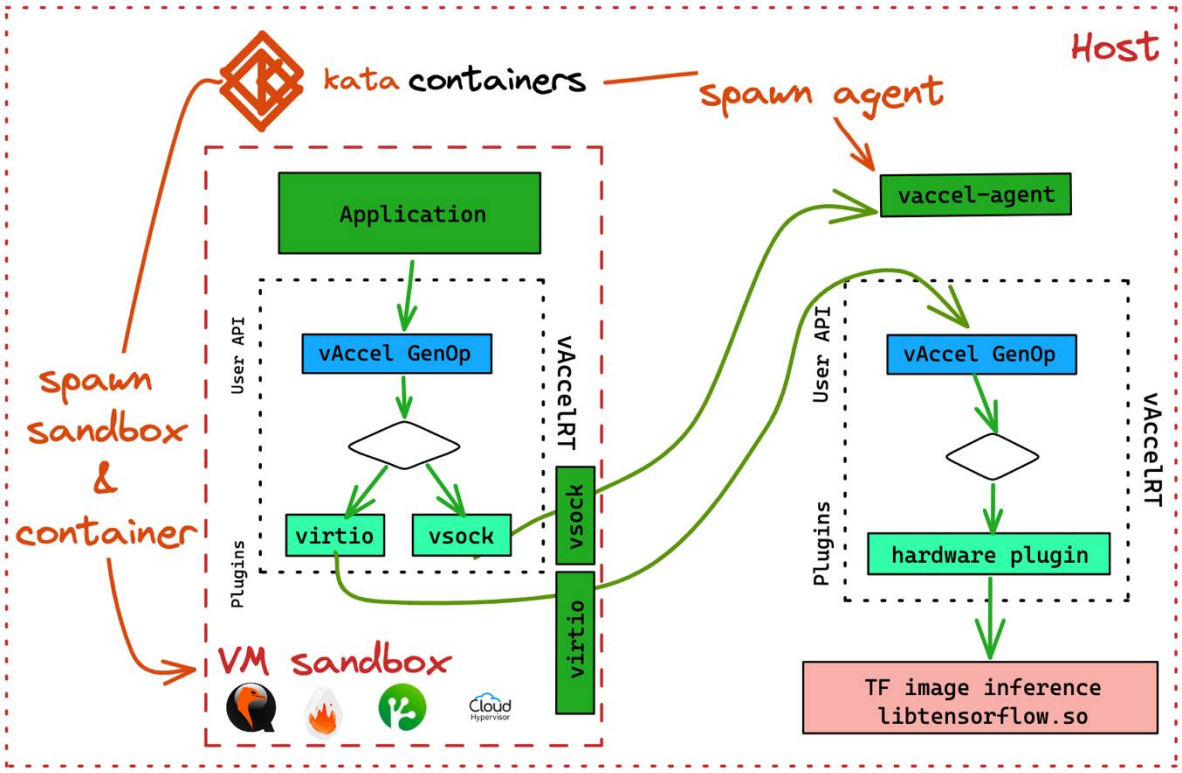


Figure 114: vAccel integration with container runtimes

6.1.3 Framework and Language Bindings

To facilitate the use of vAccel, we provide bindings for popular languages apart from C. Essentially, the vAccel C API can be called from any language that interacts with C libraries. Building on this, we provide support for Python [47] and Rust while working on extending support for various other high- or low-level languages. In SERRANO, the serverless function implementation for all kernels uses the vAccel Python bindings.

Additionally, we have implemented a subset of Tensorflow [48] and PyTorch APIs so that the user can execute an application written for those frameworks over vAccel with minimal and/or no changes.

6.1.3.1 Python Bindings

The vAccel Python API enables communication and interaction between application code and the underlying `libvaccel.so`, offering the ability to import the API into a Python codebase and gain access to a wide range of functions, classes, and utilities. By utilising this API and harnessing the power of vAccel plugins, developers can build Python applications that seamlessly interface with `libvaccel.so`, leverage its specialised functionality, and execute custom operations tailored to their specific requirements.

The Python bindings offer developers a wide range of operations, giving developers direct access to the rich functionalities of the underlying library or framework. These operations facilitate integration and interaction with the core features.

In the context of SERRANO, we use the `vaccel-exec` operation, so we will present the python bindings for these specific functions. All the available bindings are available at the documentation website [49].

6.1.3.2 Generic Executor

6.1.3.2.1 Genop

`Genop` is a class that implements `vaccel_genop()`, the generic function of vAccel that issues a generic operation request to the core library.

Input parameters:

- ***session***: A Python Session instance to manage and maintain the state and context of the operation
- ***arg_read***: A list of "struct vaccel_args" containing a pointer to a buffer and its length. These arguments are parsed from the plugin to form the actual function arguments used in the plugin implementing the respective functionality
- ***arg_write***: A list of "struct vaccel_args" objects that specify the arguments to be written or modified by the *`genop`* operation

Output results:

- A list that contains the output or results of the *`genop`* operation. The specific content of the result list depends on the implementation of the *`genop`* method and the purpose of the operation.

6.1.3.2.2 Exec with Resource

`Exec` with resource expands the capabilities of the Python bindings by enabling seamless integration with a shared library, giving the opportunity of extending use-case scenarios. By utilising *`exec_with_resource()`*, we can execute code stored within the shared library while effectively managing a specific resource tied to a given symbol.

Input parameters:

- ***object***: The path of a shared library (.so file) containing the desired operations and functions
- ***symbol***: The identifier associated with the object called for the execution
- ***arg_read***: A list of "Any" type that can accept a variable number of arguments of any type. We use the class "Vaccel_Args" to transform those arguments to "struct vaccel_args", containing a pointer to a buffer and its length. These arguments are

parsed from the plugin to form the actual function arguments used in the plugin implementing the respective functionality

- ***arg_write***: A list of "Any" type that can accept a variable number of arguments of any type. We use the class "Vaccel_Args" to transform those arguments to "struct vaccel_args" objects that specify the arguments to be written or modified after the execution

Output results:

- A list that contains the output or results of the exec operation. The specific content of the result list depends on the implementation of the `exec` method and the purpose of the operation.

The input parameters `object` and `symbol` are provided as strings to the `exec_with_resource()` method, allowing flexibility in specifying the shared library and the associated symbol. These strings are processed by the class `Object` we have created to ensure the proper handling and utilisation of these strings. This class provides methods that facilitate the loading and interaction with the shared library, as well as the identification of the desired symbols.

The class "Object" provides the following methods:

- ***__parse_object__***: Parses a shared object file and returns its content and size
- ***create_shared_object***: Creates a shared object from a file and returns a pointer to it
- ***object_symbol***: Transforms the given symbol
- ***register_object***: Registers the object for further processing
- ***unregister_object***: Removes the object from the class
- ***destroy_shared_object***: Destroys the object

6.1.4 SERRANO Kernels on vAccel

To port the SERRANO hardware accelerated kernels on vAccel we focused on hardware interoperability and ease-of-deployment.

6.1.4.1 Interoperability

One of the key merits of the vAccel framework is the fact that users write their code using the vAccel API and the underlying plugin executes this code in the respective accelerator device. This enables hardware interoperability as the user does not need to rewrite, or even re-compile their code if they want to run on a different hardware accelerator. This greatly facilitates the scaling of hardware-accelerated applications throughout the cloud-edge continuum, as the user builds a container image with their vAccel API code, deploy it in the SERRANO platform and this code can use hardware accelerators in the Cloud (Generic, NVIDIA

GPUs), at the Edge (Jetson GPUs, Orin/Xavier/Nano), or even CPUs when there is no hardware accelerator available (eg. on a RPi4).

With this in mind, we ported KNN, K-MEANS, Black-Scholes, and SavGol to vAccel, developing plugin implementations for CPU, GPU, and FPGA hardware accelerators. In the following sections we briefly elaborate on the porting methodology and the performance implications this integration imposes.

6.1.4.1.1 Libification

The main way of allowing applications to run on the vAccel framework is by separating the part we want to abstract away from the core I/O part of the application. Since the actual application is essentially the kernel to be abstracted, nearly all the code from the kernel resides in the plugin part of the vAccel stack. Instead of developing separate API calls and plugins for all the available kernels and execution modes, we chose to abstract this functionality to a simple exec operation: we "libify" the hardware-accelerated part of the application and build it using the same methods as the generic kernel (e.g. for GPU code, we use nvcc, and the output binary is a shared library, eg `libknn_app_gpu.so`, exposing the symbol of the kernel we are porting, eg `knn_app`).

We followed the above method for all kernels. The summary of kernels and libraries available is the following table.

Table 27: SERRANO kernels ported to vAccel

Kernel	Symbol	Library	Hardware
k-NN	knn_app	libknn_app_cpu.so	CPU
		libknn_app_gpu.so	GPU
		libknn_app_fpga.so	FPGA
k-MEANS	kmeans_app	libkmeans_app_cpu.so	CPU
		libkmeans_app_gpu.so	GPU
		libkmeans_app_fpga.so	FPGA
BS	bs_app	libbs_app_cpu.so	CPU
		libbs_app_fpga.so	FPGA
SAVGOL	savgol_app	libsavgol_app_cpu.so	CPU
		libsavgol_app_gpu.so	GPU
		libsavgol_app_fpga.so	FPGA

Essentially, to port the kernels to vAccel, we followed the steps below:

- Step1: use the host application as the “frontend”: we replaced the call to the relevant function with a library call implemented by all modes of execution for the specified kernel. We implemented “plugin” libraries for each of the core code versions (CPU, GPU, FPGA) and verified the execution is exactly the same as the original code.
- Step 2: we replaced this library call with a vAccel-specific call. This library, essentially, the “frontend library”, enabled us to set up the necessary data structures to ensure input and output consistency. Afterwards, using the same plugin libraries as before, we were able to specify which plugin library we want to use for each execution example: as we used the vaccel-exec operation, all we needed to do is provide the frontend with the shared object to be executed on the host, and a symbol (summarised in Table 27).

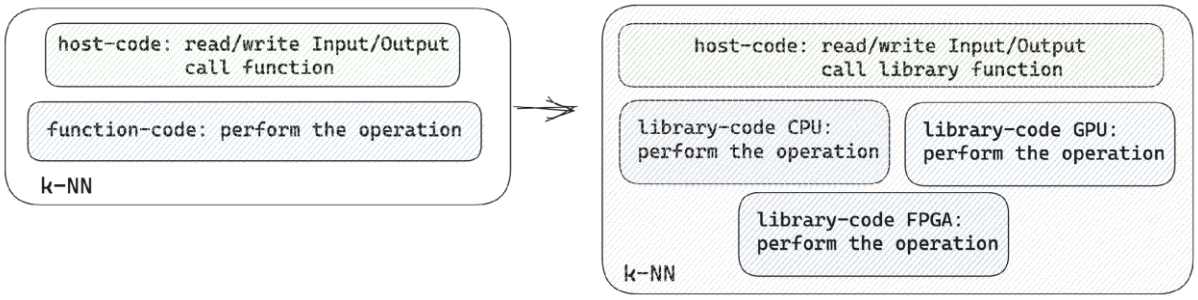


Figure 115: Libification of original kernel

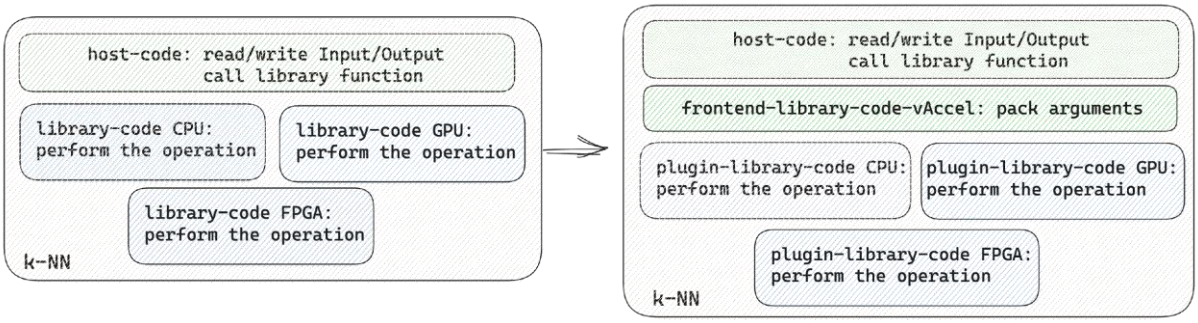


Figure 116: vAccel port

Figure 115 and Figure 116 illustrate the above process as steps 1 and 2.

To assess the overhead imposed by this process to the specific kernels, we performed an initial evaluation on a Jetson Xavier AGX system (CPU and GPU execution). We measured execution time with the identical input provided by the partners that developed the kernels.

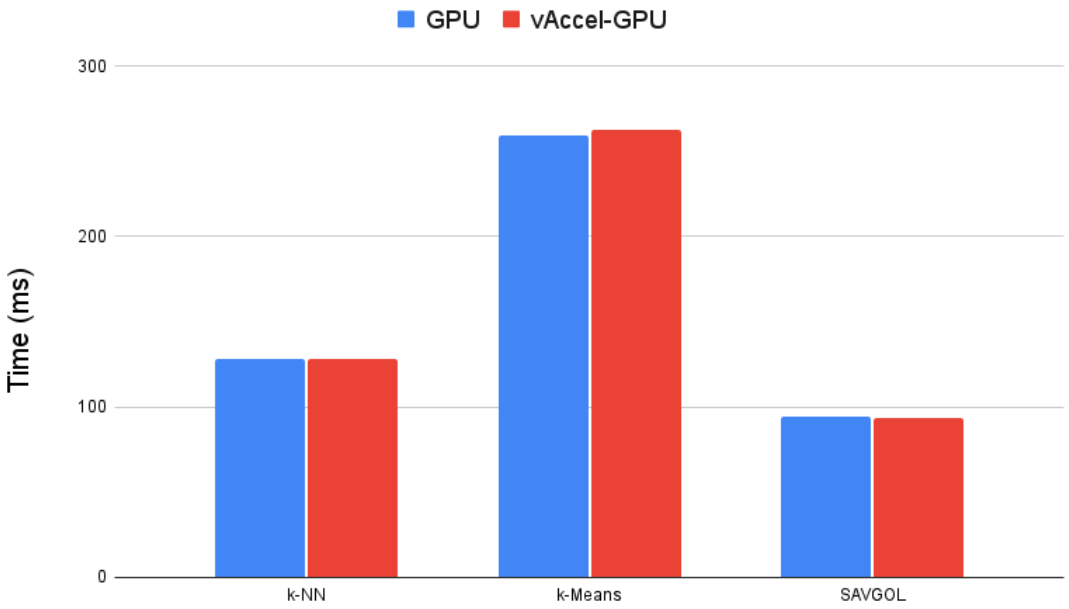


Figure 117: Performance overhead of vAccel on local execution (library overhead)

Figure 117 presents the absolute execution time (in ms) for the GPU version of each of the three kernels studied, k-NN, k-Means, and SAVGOL. The blue bars present the execution time of the stock kernels provided by the partners vs the vAccel-ported ones. Figure 117 shows that running the kernels via vAccel on the same host imposes negligible overhead.

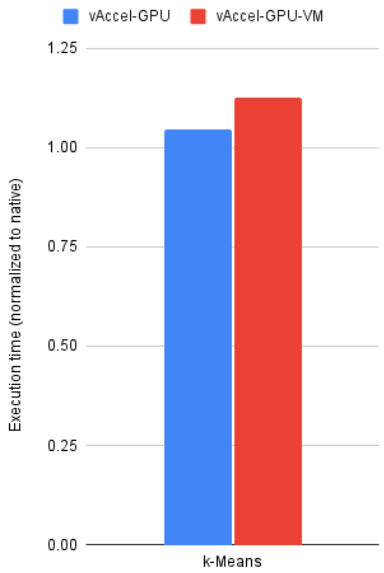


Figure 118: Performance overhead of vAccel for VM execution

Figure 118 shows the normalised execution time of the k-Means kernel to native execution, when running on the host (vAccel-GPU, blue bars) and on a virtual machine (vAccel-GPU-VM, red bars). We are investigating the source of the overhead imposed on the VM execution. Part of this is accounted to the data transfer between the VM and the host.

6.2 OpenFaaS

In SERRANO, we build on OpenFaaS [50] to provide short-lived task execution functionality. OpenFaaS is a serverless framework that allows users to deploy functions written in any language to a Kubernetes cluster or standalone VM inside containers. It provides auto-scaling and metrics for the deployed functions. It abstracts the underlying infrastructure and allows users to deploy their services using a high-level CLI tool or Web UI.

6.2.1.1 Porting the SERRANO Kernels to Serverless Functions

To accommodate the diverse input/output modes of the kernels, as well as the various modes of execution, we used the vAccel python bindings to facilitate the process of porting the kernels to serverless functions.

Essentially, the logic of the execution remains the same; the only thing that changes is the way we get the input and we provide the output.

Since the plugin libraries for executing different algorithms are the same as described in Subsection "Libification", we can use them over the vAccel API by executing the `exec_with_resource` function. We have developed tests to ensure the proper interaction and integration between the algorithm and the plugin library, through vAccel which enables them to interact efficiently.

KNN

For the KNN test, after loading the necessary libraries for the interaction with vAccel, we must convert the `.csv` files that will be processed into a format suitable for execution. We establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format we execute the `exec_with_resource` function with the necessary input arguments:

- object: `libknn_app` library
- symbol: The symbol that implements the k-NN algorithm in the context of the plugin, eg: `knn_app`
- arg_read: The converted read arguments we have packed appropriately.
- arg_write: The converted write arguments we have packed appropriately.

K-Means

For K-Means we are working again in a similar way. After loading the necessary libraries for the interaction with vAccel, we convert the `.csv` files that will be processed into the format we want. After doing that we establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format we execute the `exec_with_resource` function with the necessary input arguments:

- object: `lib_kmeans_app` library
- symbol: The symbol that implements the k-Means algorithm in the context of the plugin, eg: `kmeans_app`
- arg_read: The converted read arguments we have packed appropriately.
- arg_write: The converted write arguments we have packed appropriately.

SAVGOL

For SAVGOL we are working again in a similar way. After loading the necessary libraries for the interaction with vAccel, we convert the `.csv` files that will be processed into the format we want. After doing that we establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format we execute the `exec_with_resource` function with the necessary input arguments:

- object: savgol_app library
- symbol: The identifier of savgol library
- arg_read: The converted read arguments we have packed appropriately.
- arg_write: The converted write arguments we have packed appropriately.

An example Python program that calls the K-NN kernel using vAccel is shown in Table 28.

Table 28: Python snippet that implements the k-NN execution over Python vAccel

```
def k-NN_vAccel(INPUT_PATH, LABELS_PATH, MODE, iterations):

    t0 = time.time_ns() // 1_000_000
    # Setup input
    start = time.time()
    timeseries = transformed_time_series(INPUT_PATH).astype(np.float32).flatten()
    print('Time for dataset read + transform: ', round(time.time() - start,3), 's')

    labels = load_labels(LABELS_PATH).astype(np.int32)
    golden_labels = labels.copy()
    nr_iter = iterations
    w = 200

    # Setup shared object (plugin) CPU/GPU/FPGA
    obj = "libkmeans_app_%s.so" % MODE

    t1 = time.time_ns() // 1_000_000
    c1 = timeseries[:N_FEATURES]
    c2 = timeseries[N_FEATURES+1:2*N_FEATURES]
    # Setup vAccel parameters
    pa = ffi.cast(f"float[{len(timeseries)}]", ffi.from_buffer(timeseries))
    pc1 = ffi.cast(f"float[{len(c1)}]", ffi.from_buffer(c1))
    pc2 = ffi.cast(f"float[{len(c2)}]", ffi.from_buffer(c2))
    pc = ffi.cast(f"int [{len(labels)}]", ffi.from_buffer(labels))

    # Pack arguments
```

```
    arg_read_local = [pa, nr_iter, w, pc1, pc2]
    arg_write = [pc]

    t2 = time.time_ns() // 1_000_000
    # execute command
    res       =       Exec_with_resource.exec_with_resource(obj,       "kmeans_app",
arg_read=arg_read_local, arg_write=arg_write)
    t3 = time.time_ns() // 1_000_000

    labels_new = ffi.unpack(arg_write[0],len(arg_write[0]))
    total_elements = len(labels_new)
    matching_elements = sum(a == b for a, b in zip(golden_labels, labels_new))
    convergence_percentage = (matching_elements / total_elements) * 100

    t4 = time.time_ns() // 1_000_000
    print(convergence_percentage)
```

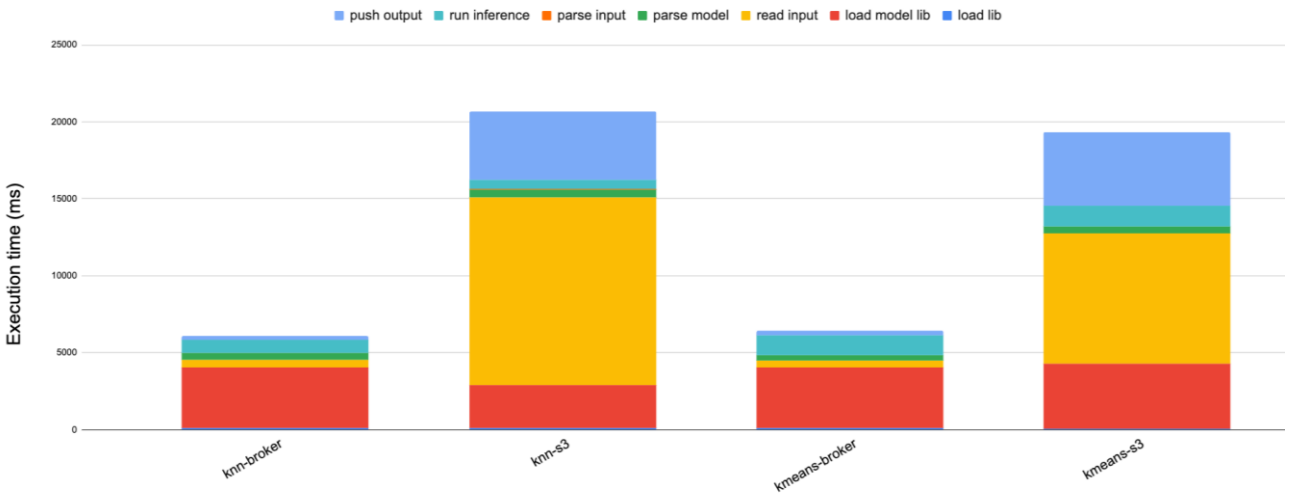


Figure 119: Performance overhead of end-to-end operation with sandboxed OpenFaaS container and vAccel

Figure 119 presents the end-to-end execution time (in ms) for k-NN and k-Means when called as serverless functions. To perform this test, we built a serverless function that receives a JSON object as input in the format that is presented in Table 29.

Table 29: Input format for the serverless function

Parameter		
queue_id		A random UUID, acting as the identifier for the storage backend
arguments	***position***	input data
	labels	input data
	input file	input data
uuid		a unique id, acting as the identifier for the kernel execution
mode		the accelerator to be used (CPU, GPU, or FPGA)

storage		the storage backend to be used (data broker or s3)
creds	***ip***	ip address of the storage backend
	user	username for the storage backend
	pass	password for the storage backend

Figure 119 identifies a number of issues we are currently investigating:

- Loading the python libraries on each function invocation is time-consuming
- fetching and pushing data to the s3 storage backend is almost 10x slower than performing the same operation through the data broker

Overall, spawning the specific kernels from an external client, simulating the end-to-end case is an important milestone achieved in Task 4.4. We are working closely with WP5 and WP6 to integrate our implementation to the SERRANO platform and optimise the time-consuming parts.

7 Conclusion

In conclusion, the SERRANO platform in the WP4 has significantly expanded the range of available accelerators, including energy-efficient devices at the network edge as well as high-performance, massively parallel devices in the cloud and HPC environments. Through the development of different applications' versions, including HPC, GPU, and FPGA-accelerated versions, the platform offers a wide range of options that consider performance and energy efficiency tradeoffs, providing flexibility to the orchestration framework.

The HPC, GPU, and FPGA kernel implementation has been integrated with transprecision and approximation computing techniques. This integration allows adaptive execution with different data precisions, leading to minimised computations and improved resource utilisation. The Verification, Validation, and Uncertainty Quantification (VVUQ) framework further tackles uncertainties and trade-offs by suggesting optimal parameters for kernel execution, optimising runtime and energy consumption.

Using approximation techniques, such as precision scaling, approximate minimisation, and loop perforation, in FPGA-accelerated application versions has enhanced energy efficiency and expanded the library of available use case applications. Additionally, experiments on algorithmic transparencies' adaptation for distributed streaming applications in Edge/Fog computing systems were conducted to reduce network latency and increase bandwidth, addressing the demands of real-time data processing.

The Plug&Chip framework played a vital role in developing FPGA and GPU accelerators, enabling the automatic optimization of kernels for enhanced performance without human intervention. Moreover, a methodology for creating memory-efficient accelerators on FPGAs was introduced, further optimising resource utilisation. The vAccel framework has addressed the scaling of hardware-accelerated operations by exposing hardware-acceleration functionality to isolated serverless functions.

Overall, the SERRANO platform has made substantial advancements in WP4 in the realm of accelerators, optimization techniques, uncertainty quantification, and serverless execution. By integrating various hardware acceleration options, addressing resource limitations, and enhancing flexibility in deployment and execution, the platform has paved the way for high-performance, energy-efficient computing in diverse environments, contributing to advancements in various use case applications.

8 References

[1] MPI Forum. Official website: MPI Forum. This website contains information about the activities of the MPI Forum, which is the standardisation forum for the Message Passing Interface (MPI). Feb. 2022. URL: https://www.mpi-forum.org.

[2] OpenMP. The OpenMP API specification for parallel programming. OpenMP 5.2 Released with Improvements and Refinements. Feb. 2022. URL: https : / / www .openmp.org.

[3] Savitzky, A., and Golay, M.J.E. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." Analytical Chemistry, vol. 36, no. 8, 1964, pp. 1627-1639.

[4] Oroutzoglou, I., Kokkinis, A., Ferikoglou, A., Danopoulos, D., Masouros, D., & Siozios, K. (2022, June). Optimizing Savitzky-Golay Filter on GPU and FPGA Accelerators for Financial Applications. In *2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST)* (pp. 1-4). IEEE.

[5] Welch, G., and Bishop, G. "An Introduction to the Kalman Filter." University of North Carolina at Chapel Hill, 2006.

[6] Daubechies, I. (1992). Ten Lectures on Wavelets. SIAM.

[7] Specifying Arrays as Ping-Pong Buffers or FIFOs, https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/Specifying-Arrays-as-Ping-Pong-Buffers-or-FIFOs

[8] Hull, J.C. "Options, Futures, and Other Derivatives." Prentice Hall, 2018.

[9] Khan, Kamran, et al. "DBSCAN: Past, present and future." The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014). IEEE, 2014

[10] Müller, Meinard. "Dynamic time warping." Information retrieval for music and motion (2007): 69-84.

[11] Brigham, E. Oran. The fast Fourier transform and its applications. Prentice-Hall, Inc., 1988

[12] Xilinx FFT IP Library (https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/FFT-IP-Library)

[13] Dataflow Processing in HLS (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-dataflow)

[14] Keogh, E., and Ratanamahatana, C.A. "Exact indexing of dynamic time warping." Knowledge and Information Systems, vol. 7, no. 3, 2005, pp. 358-386.

[15] Duda, R. O., Hart, P. E., & Stork, D. G. (2001). Pattern Classification and Scene Analysis. Wiley-Interscience.

[16] Chen, Y., and Vrudhula, S. "Approximate Computing: A Cross-Layer Perspective." Morgan Kaufmann, 2014.

[17] Micikevicius, P., et al. (2017). Mixed precision training. arXiv preprint arXiv: 1710.03740.

[18] Xilinx AP Data Types (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Arbitrary-Precision-AP-Data-Types)

[19] Roache, P. J. (1994). "Verification and validation in computational science and engineering." Hermosa Publishers.

[20] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html#:~:text=RAPL%20is%20an%20interface%20for,chip%20(SoC)%20power%20domains.

[21] https://docs.nvidia.com/deploy/nvml-api/index.html

[22] https://www.amd.com/en/products/cpu/amd-epyc-7702

[23] https://perf.wiki.kernel.org/index.php/Main_Page

[24] https://github.com/powerapi-ng/pyJoules

[25] Numan, Mostafa W., et al. "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains." IEEE Access 8 (2020): 174692-174722.

[26] Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." IEEE transactions on evolutionary computation 6.2 (2002): 182-197.

[27] Blank, Julian, and Kalyanmoy Deb. "Pymoo: Multi-objective optimization in python." IEEE Access 8 (2020): 89497-89509.

[28] Xilinx Vitis (https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html)

[29] Kokkinis, A., Diamantopoulos, D., & Siozios, K. (2022). Dynamic optimization of on-chip memories for HLS targeting many-accelerator platforms. *IEEE Computer Architecture Letters*, *21*(2), 41-44.

[30] Kokkinis, A., Diamantopoulos, D., & Siozios, K. (2022, August). Dynamic Heap Management in High-Level Synthesis for Many-Accelerator Architectures. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 287-293). IEEE.

[31] https://github.com/ict-serrano/MC-DMM-Analysis-HLS

[32] Huang, Sitao, et al. "Accelerating sparse deep neural networks on FPGAs." *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.

[33] Diamantopoulos, D., Xydis, S., Siozios, K., & Soudris, D. (2015). Mitigating memory-induced dark silicon in many-accelerator architectures. *IEEE Computer Architecture Letters*, *14*(2), 136-139.

[34] Kokkinis, A., Diamantopoulos, D., & Siozios, K. (2022, August). Dynamic Heap Management in High-Level Synthesis for Many-Accelerator Architectures. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 287-293). IEEE.

[35] Kokkinis, A., Diamantopoulos, D., & Siozios, K. (2022, August). Dynamic Heap Management in High-Level Synthesis for Many-Accelerator Architectures. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 287-293). IEEE.

[36] Kokkinis, A., Diamantopoulos, D., & Siozios, K. (2022). Dynamic optimization of on-chip memories for HLS targeting many-accelerator platforms. *IEEE Computer Architecture Letters*, *21*(2), 41-44.

[37] Guerreiro, A. Ilic, N. Roma, and P. Tomas, “Gpu static modelling using ptx and deep structured learning,” IEEE Access, vol. 7, pp. 159150–159161, 2019.

[38] https://github.com/cavazos-lab/PolyBench-ACC.

[39] https://vaccel.org.

[40] https://www.qemu.org/

[41] https://firecracker-microvm.github.io/

[42] https://www.cloudhypervisor.org/

[43] https://katacontainers.io/

[44] https://github.com/cloudkernels/qemu-vaccel/tree/vaccelrt

[45] https://github.com/cloudkernels/firecracker/tree/vaccel-0.23

[46] https://katacontainers.io/

[47] https://docs.vaccel.org/python_bindings/

[48] https://docs.vaccel.org/tensorflow_bindings/

[49] https://docs.vaccel.org

[50] https://www.openfaas.com/