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

Grant Agreement no. 101017168

Deliverable D4.3
Framework for seamlessly integration of heterogeneous workload-aware performance improvement

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

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

Dissemination level:	Public
Revision:	1.0

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

Revision History

Date	Editor	Status	Version	Changes
01.11.21	Dimosthenis Masouros	Draft	0.1	Initial ToC
01.12.21	Kostas Siozios	Draft	0.2	Content for Plug&Chip framework in Section 3.2
01.01.22	Argyrios Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou	Draft	0.3	Methodology description for automatic optimization for FPGA/GPU kernels and runtime memory management for FPGA sharing in Section 3.3, 3.4 and 3.5
01.02.22	Argyrios Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou	Draft	0.4	Initial evaluation results for automatic optimization for FPGA/GPU kernels and runtime memory management for FPGA sharing in Section 3.3, 3.4 and 3.5
14.02.22	Anastasios Nanos	Draft	0.5	Initial ToC for section 4
21.02.22	Argyrios Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou	Draft	0.6	Final evaluation results for automatic optimization for FPGA/GPU kernels and runtime memory management for FPGA sharing in Section 3.3, 3.4 and 3.5
28.02.22	Anastasios Nanos, Argyrios Kokkinis, Aggelos Ferikoglou, Dimosthenis Masouros	Draft	0.7	Integrated NBFC input for section 4. Finalized all deliverable sections. Revised all sections before internal review.
08.03.22	Ferad Zyulkyarov, Aggelos Ferikoglou	Draft	0.8	Integrated INB comments from internal review
22.03.22	Makis Karadimas, Paraskevas Bourgos, Aggelos Ferikoglou	Draft	0.9	Integrated INTRA comments from internal review
28.03.22	Aggelos Ferikoglou	Public	1.0	Deliverable final refinement

Author List

Organization	Author
AUTH	Argyrios Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou, Dimitrios Danopoulos, Dimosthenis Masouros, Kostas Siozios
NBFC	Anastasios Nanos, Charalampos Mainas, Georgios Ntoutsos
INB	Ferad Zyulkyarov
INTRA	Makis Karadimas, Paraskevas Bourgos

Internal Reviewers

Ferad Zyulkyarov (INB)

Makis Karadimas, Paraskevas Bourgos (INTRA)

Abstract: This deliverable (D4.3) presents the outcomes of *Task 4.3 "Seamlessly integration of heterogeneous architectures for improving developers' productivity in HW/SW co-design of data-intensive applications"*, *Work Package 4 "Cloud and Edge Acceleration"* of the SERRANO project, during the first iteration of the incremental implementation plan. The deliverable presents the Plug&Chip extensions for automatic optimizations for FPGA and GPU kernels as well as the runtime memory management mechanism for FPGA sharing. It also provides the initial results for the aforementioned extensions. Moreover, the SERRANO abstractions for HW acceleration of short-lived tasks are introduced.

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

Disclaimer: *The information, documentation and figures available in this deliverable are written by the SERRANO Consortium partners under EC co-financing (project H2020-ICT-101017168) and do not necessarily reflect the view of the European Commission. The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The reader uses the information at his/her sole risk and liability.*

Table of Contents

List of Figures

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
EDA	Electronic Design Automation
EDE	Event Detection Engine
EU	European Union
FaaS	Function as a Service
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

GCP	Google Cloud Platform
GPU	Graphics Processing Unit
HDL	Hardware Definition Language
HLRS	High Performance Computing Center Stuttgart
HLS	High-Level Synthesis
HPC	High Performance Computing
HW	Hardware
IO	Input/Output
IPC	Inter Process Communication / Instructions Per Cycle
MAAS	Metal as a Service
ML	Machine Learning
MOM	Message-Oriented Middleware
MPI	Message Passing Interface
MTU	Maximum Transmission Unit
NIC	Network Interface Controller
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
TLM	Transaction Level Modeling
TLS	Transport Layer Security
TPM	Trusted Platform Module
TTM	Time to Market
UC	Use Case
URL	Uniform Resource Locator

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

1 Executive Summary

H2020 SERRANO project's purpose is to successfully provide a transparent way of deploying tasks in the Edge-Cloud-HPC computing continuum. More specifically, SERRANO develops a rapid prototyping framework for enabling efficient HW/SW co-design realization of data-intensive applications

This document presents the SERRANO deliverable D4.3 entitled "Framework for seamlessly integration of heterogeneous workload-aware performance improvement" focusing on the first version of SERRANO's framework for automatically fine-tuning applications in terms of performance.

In this deliverable, we first present the SERRANO's vision for seamless HW/SW co-design. Afterwards, we describe our work on automatic optimization for FPGA and GPU accelerators and the runtime memory manager we developed for FPGA sharing. Finally, we conclude with abstractions for HW acceleration of short-lived tasks.

2 Introduction

Nowadays, the ever-increasing demands for high performance computations and low power designs have fundamentally changed the human perspective on developing and executing applications on both the edge and the cloud. Next-generation analytics, Machine Learning (ML) algorithms and workloads that require high-performance are deployed on heterogenous computing ecosystems with diverse computational capabilities and power requirements. Edge computing is typically hindered by power consumption, area footprint and cost while high-performance computations in data centers require secure and high-speed data transfer from the local platforms to the processing infrastructures. To meet the user requirements for energy-efficiency, high throughput and security in the new computing paradigm, adaptive computing has been introduced. Acceleration platforms such as GPUs and FPGAs can achieve higher performance than most of the typical processing systems without compromising the user requirements for energy-efficiency and security. The task of efficiently designing and deploying computationally intensive applications on those highly specialized compute units is not always clear even for the most-experienced developers. For that reason, ML-enabled automated and semi-automated solutions that allow the rapid and optimized deployment of different algorithmic tasks on hardware platforms are developed. In this work we present Plug&Chip framework and its components that contribute to the field of design automation and memory management for GPUs and FPGAs. In sub-section 3.1, we present the Plug&Chip framework. In sub-section 3.2, a mechanism towards exploring efficiently the optimizations' design space (DSE) for FPGA accelerators is suggested. In sub-section 3.3, a ML-based tool that applies the optimal block coarsening factors in a CUDA kernel is proposed. In sub-section 3.4, framework that allows multiple accelerators to be synthesized on the same FPGA and share their memory resources is presented. Finally, in section 4 a framework that allows hardware acceleration on serverless systems is discussed.

3 Rapid prototyping for efficient HW/SW co-design of compute-intensive applications

Serrano envisions to provide transparent acceleration of compute intensive tasks. However, realizing this goal in a seamless manner is not a trivial process, since it typically requires deep expertise of both hardware infrastructure and software optimizations. Serrano aims to break this wall by providing a set of automated tools for efficient HW/SW co-design of compute intensive applications. Specifically, the SERRANO platform combines both offline and online optimization techniques that provide for increased performance and enhanced energy efficiency for deployed applications. All the developed extensions are integrated with the Plug&Chip framework, which provides system modelling and simulation between system's kernels mapped onto different platforms. In the rest of this section, we give an overview of the Plug&Chip framework and we also describe in detail the automatic optimization techniques for seamless GPU and FPGA acceleration of compute-intensive applications.

3.1 Plug&Chip Framework

In the embedded system domain, there is a continuous demand towards providing higher flexibility for application development. This trend strives for virtual prototyping solutions capable of performing fast system simulation. Among others, such a solution supports concurrent hardware/software system design by enabling to start developing, testing, and validating the embedded software substantially earlier than it has been possible in the past.

The existing Electronic Design Automation (EDA) tools that support these tasks are crucial for deriving an optimum solution. Most of these software tools are built on the fundamental premise that models are freely interchangeable amongst vendors and have interoperability amongst them. In other words, this imposes those models can be written, or obtained from other vendors, while it is known a priori that they will be accepted by any vendor tool for performing different steps of physical prototyping (e.g., architecture's analysis, simulation, synthesis, etc). Even though this concept seems straightforward and promising, it has been proven completely elusive in the world of Electronic System Level (ESL). Specifically, the existing ESL solutions do not provide neither model interoperability, nor independence between model and software tools. Consequently, the adoption of ESL flows between different vendors could be though as a desired feature.

Towards this direction, and as research pushes for better programming models for multi-processor and multi-core embedded systems, Virtual Platforms (VP) address one of today's biggest challenges in physical design: to enable sufficient software development, debug and validation before the hardware device becomes available. More specifically, with the virtualization feature, it is possible to model a hardware platform consisted of different processing cores, memories, peripherals, as well as interconnection schemes, in the form of a

simulator[1]. Furthermore, as the task of hardware development progressively proceeds, it is feasible to redistribute to software teams updated versions of the VP that enable even better description of target architecture.

The concept of virtualization is also important for hardware architects, as it enables easier verification of IP (Intellectual Properties) kernels. This feature could be employed both in the case where only a few of the application's kernels have to be developed in hardware, as well as if incremental system prototyping is performed. In both cases, the virtualization feature provides all the necessary mechanisms for performing co-simulation and verification between the IPs developed in RTL (Register Transfer Level) and the rest application's functionalities executed onto the VP.

Plug&Chip framework automates the procedures dealing with system modelling and simulation between system's kernels mapped onto different platforms, namely the host PC, the VP and the target implementation medium (e.g., FPGA, ASIC, etc). Since Plug&Chip handles multiple platforms, a generic communication scheme for providing the desired data transfers under various parameters (e.g., packet size, maximum transmission unit (MTU), etc) have been developed.

3.1.1 Plug&Chip Methodology

Even though there are plenty of design tools that tackle software (SW) and hardware (HW) problems individually, there are only a few approaches that leverage problems arising in systems that tightly integrate SW and custom HW. This mainly occurs due to the challenges related to system integration that have to be addressed. Even limited, there are EDA approaches which promise to alleviate the integration problem in RTL simulation, emulation and prototyping environments. However, these solutions are often too complex, slow and expensive. Usually, it is the communication link between the host computer and prototyping HW that is mostly constrained.

This subsection describes the Plug&Chip framework that enables product development jointly by SW and HW teams in a way that close interaction is allowed during the development phases. Figure 1 shows the overall flow of the proposed Plug&Chip methodology consisted of three consecutive design stages: (i) system modelling, (ii) rapid virtual prototyping and (iii) system integration. The competitive advantage of this framework is the provided PC-based co-simulation, which trade-offs between speed (functional simulation) and accuracy (cycle-accurate simulation), depending on designer requirements.

[1] OVP: Open Virtual Platforms (http://www.ovpworld.org)

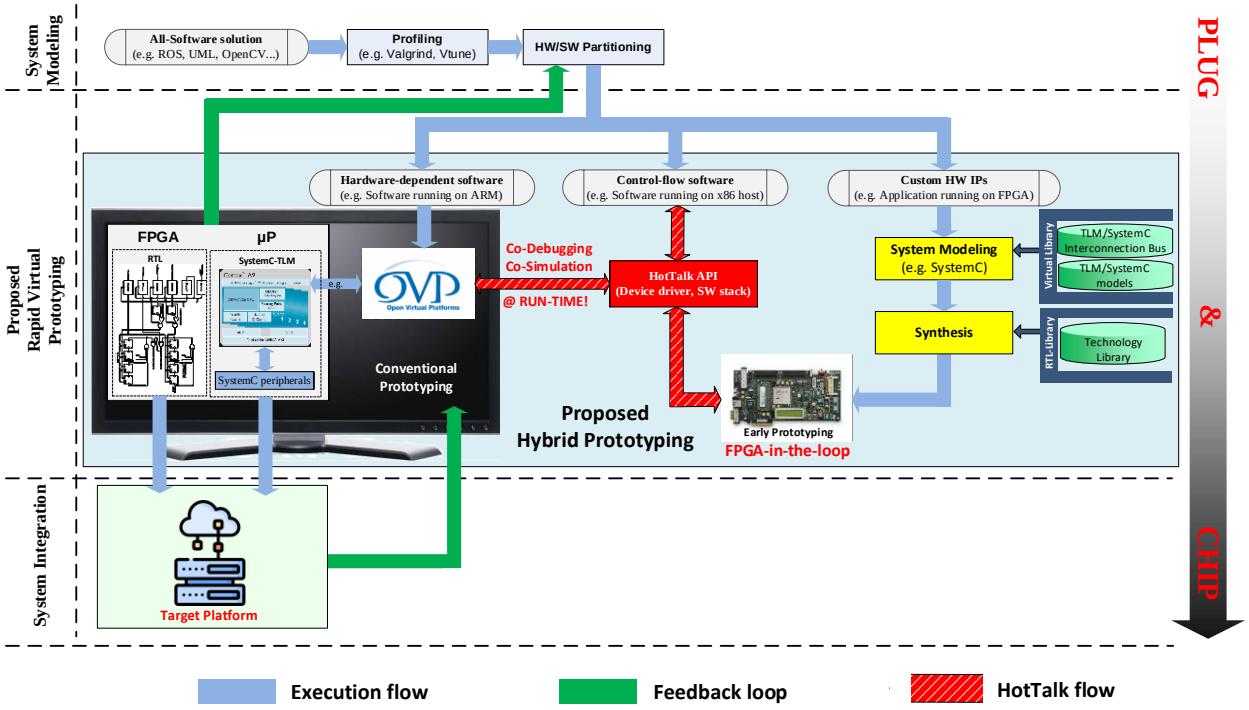

Figure 1: The enhanced Plug & Chip rapid prototyping framework

System modelling describes the stage where the development team provides an abstract description of the system's architecture and starts planning the way that this functionality has to be modelled into HW and SW kernels. Also, during this stage, all the top-level performance constraints are met. Since this is an early design step, the functionality of the final system is described with an all-software solution.

Starting from this all-software solution, initially we profile the application's algorithms to determine those kernels that highly affect the system's performance. Different criteria might be incorporated for this task, while the most common ones affect the determination of computationally intensive tasks, the tasks with increased demand for communication (I/O), as well as those tasks that can be executed much faster if we extract their inherent parallelism. For this purpose, a number of software tools can be employed (e.g., Valgrid, VTune, etc). Based on the conclusions derived from profiling task, it is possible to perform a HW/SW partitioning to the pure software implementation, depending on the criteria discussed previously. The output of partitioning step classifies the “all-software” solution to three categories: (i) the HW-dependent software, (ii) the control-flow SW and (iii) the custom HW IPs.

Custom HW IPs are peripherals and hardware accelerators, which provide platform connectivity to off-chip world and acceleration to software functions, respectively. The HW-dependent software refers to the algorithms executed onto the embedded target CPU. While our methodology relies on a PC-based development infrastructure, there is a software stack running on a native host (e.g., x86 compatible), which is responsible for establishing the communication layer between HW-dependent software and custom HW IPs, as well as to

provide all the necessary synchronization for the computation tasks. This software stack is referred in Figure 1 as “control-flow software”. Apart from the development stage discussed previously, the host PC could also be part of the final system. In such a case, the control-flow software includes also the SW that will be executed onto the host PC.

The introduced framework also addresses limitations posed during the profiling step. More specifically, in case where the profiling procedure is performed on a different platform from the actual target system, this might lead to inaccuracies in the derived conclusions. For instance, a different platform imposes changes in the Instruction Set Architecture (ISA), the microarchitecture, the compiler, resulting in variances in the executable that is profiled. In order to strengthen the partitioning decisions, HW-dependent information has to be provided, while based on existing solutions, the most accurate profiling information is available after the first design prototype is developed. Typically, once a project has reached this stage, most of the budget has been sunk, which is usually beyond the point of “no return”. In contrast, the proposed framework offers the flexibility of incremental HW development and system testing under real world constraints, so that accurate profiling information can also be provided through development stage with a feedback loop.

The second stage of the proposed framework deals with the Hybrid Virtual Prototyping, which is actually the core stage of the proposed methodology. In order to support the interaction between SW and HW development teams, we adopt the usage of TLM-SystemC models. Different ISSs can be employed for this purpose, however Plug&Chip framework is based (without affecting the generality of introduced methodology) on OVP, since it is a publicly available and easily extensible approach. Additionally, the increased simulation speed provided by OVPSim ensures that complex systems can be modelled in reasonable amount of time (hundreds of millions of simulated instructions per second). As the OVP models are pre-built, they support fully functional simulation of a complete embedded system. Also, since these models are binary-compatible with the simulated HW, the developed software can be executed onto the target (final) system without any modifications. This enables faster iteration for the software development teams.

Similarly, HW developers are also benefited from the adoption of hybrid VP discussed throughout this section. Since this platform is composed of OVP and TLM/SystemC models, it exhibits increased flexibility which in turn alleviates many constraints that designers face during the architecture design. More specifically, the former models (related to OVP) describe the software part of the target system (e.g., executed onto an embedded processor), while the TLM/SystemC models provide the design functionality that has been mapped to custom HW IPs, after system partitioning.

After having a high-level system modelling that meets the design's specifications, we proceed to the HDL development. As long as new IPs are developed, the HW design team is able to incrementally test these IPs by replacing a functionality of the employed SystemC/TLM model with the equivalent HDL prototype mapped onto FPGA boards. The connection between VP and FPGA is established with HotTalk API. More specifically, this API provides the connectivity between the VP prototype and the target hardware, as well as between Host software and VP, through a physical interface found on the host PC (i.e., Ethernet, PCI, USB, etc). Next subsections describe in more detail the functionality of this API.

Offering the HotTalk API, the proposed framework provides a wide class of middleware stack, composed of device drivers on host PC, libraries in OVP and transactors in FPGA, so that designers can efficiently test the entire system from early design iterations down to the final system validation with real-world testbenches, with the minimum possible effort. As mentioned previously, such an incremental design flow provides all the necessary information about meeting the system's specifications, which in turn can be used for performing additional optimizations of the whole system or re-partitioning the software (through the feedback loop).

The last stage of the proposed methodology deals with the system integration. During this stage, the different cores of target system, including among others the embedded CPU and the hardware accelerators (e.g., FPGAs and/or GPUs) are integrated to form the target platform.

3.1.2 HotTalk API: Host2VP and VP2HW Communication Infrastructure

This subsection describes in more detail the proposed communication scheme for realizing the communication between the host PC and the VP, as well as between VP and a hardware board (e.g., FPGA). These two libraries, named Host2VP and VP2HW respectively, form the core part of HotTalk API. In addition, HotTalk API provides an FPGA transactor, i.e., a hardware module mapped on the reconfigurable platform, which realizes the physical link with the FPGA device.

3.1.3 Host2VP

Figure 2 gives a functional overview of the tasks implemented in the Host2VP library. This library provides a universal coding style for the host side in order to avoid modifications when the VP is replaced with a real board. Towards this goal, the library provides a high-level interface for realizing four main tasks: (i) open device, (ii) close device, (iii) send data to the device and (iv) receive data from the device. These calls are implemented as generic wrappers which can be adapted to any communication mechanism between the host and the embedded system (either virtual or physical) without imposing any modification to the host's software. In particular, in case of using a real hardware board (case 1 of Figure 2, which is recognized as a character device by the O/S, the I/O calls manipulate the respective O/S system calls (*open*(), *close*(), *write*() and *read*() respectively), while the overall communication is performed through the device driver.

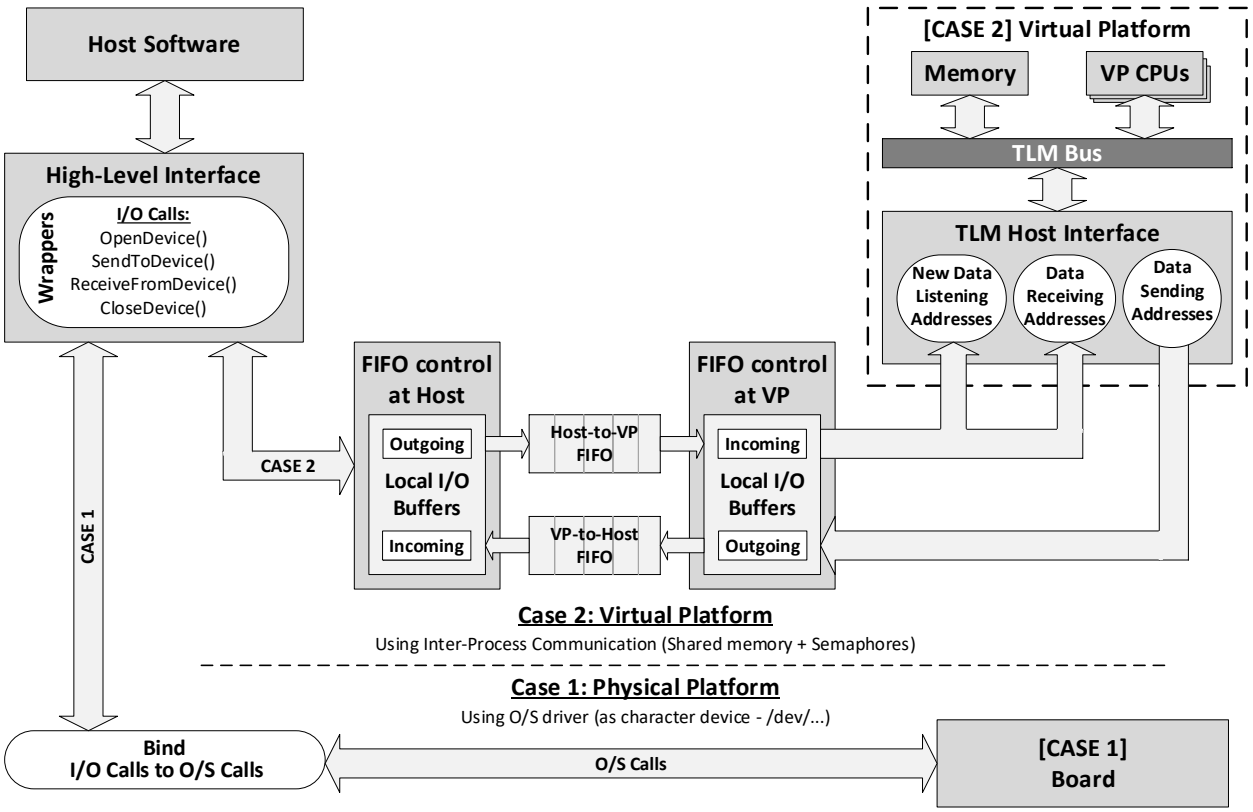

Figure 2: Communication mechanism between Host and VP

On the other hand, the second case of Figure 2 assumes that VP is used instead of a real hardware. In such a case, those I/O calls manipulate the Inter-Process Communication (IPC) mechanisms for realizing the data transfer between host and VP. For this purpose, two FIFO (First-In/First-Out) queues are employed, as it is depicted in Figure 2, each of which is implemented as a shared memory segment. The size of these queues is defined at compile time depending on the connectivity requirements between host PC and VP posed by the target architecture. Specifically, the usage of data packets with increased size leads to reduced IPC overhead in case of massive data transfers, whereas the smaller packet size is preferable whenever there is a limited amount of data to be transferred.

The efficient synchronization for data transfers between the host and the VP is also crucial for the Host2VP library. More precisely, both for the host and the VP, the data synchronization can be realized with the usage of O/S semaphores. Since each semaphore operation is an atomic action, our framework guarantees that no racing conditions will occur. This imposes that if a host has to send and receive data simultaneously, then only one operation will be committed. The second operation will take place only when the currently committed operation is accomplished.

Another feature of introduced library is its applicability to any other VP framework and/or ISS, leading to a universal communication library. For this purpose, the Host2VP library provides an easy-to-use C API, which includes standardized I/O calls for the data manipulation to the VP side. In order the VP side to utilize these calls, the library provides a TLM2.0 VP peripheral, referred as TLM Host Interface, which binds each of these I/O calls with a bus address. Three types of addresses are provided: (i) *New Data Listening Addresses* for checking if new incoming

data exist, (ii) *Data Receiving Addresses* for reading the incoming data from host and (iii) *Data Sending Addresses* for writing the outgoing data to host. Hence, the VP software can use each call by accessing the corresponding bus address.

Finally, in order to provide a fully adaptive interface at TLM level, Host2VP is implemented as an hierarchical library, based on a template class, which includes the core methods that realize the TLM transactions. These methods manipulate a set of I/O calls for checking whether new data is available, receiving the incoming data and sending the outgoing ones. With this template class, the designer can develop new TLM Host interfaces with the minimum possible effort.

3.1.4 VP2HW

The functionality of VP2HW library, which realizes the communication between the VP and a real hardware board, is depicted in Figure 3. Similarly, to the previous case, VP2HW has to support a wide variety of hardware boards. Towards this goal, the target hardware board is recognized as a character device, whereas the communication is performed through the associated device driver. Such a selection provides the appropriate synchronization for guaranteeing uncorrupted data transfer. Although for the scope of this paper the employed device driver establishes the communication between the host and the FPGA through Ethernet, any other protocol could also be used. Furthermore, this library supports raw data transfer between VP and target hardware, through the physical layer (PHY) of Ethernet (there is no requirement for protocol, or service).

In addition to this, the VP2HW library supports the communication between the VP software and a real hardware board through a TLM interface connected with the rest of VP as a TLM2.0 peripheral. In particular, the TLM interface exposes to the VP software a number of registers, which trigger the data transfer to and from the hardware board. As Figure 3 depicts, the VP software uses these registers in order to receive (send) the incoming (outgoing) data respectively, after setting the length of the transferred data. Also, a status register indicates if the data transfer between VP and hardware was successful. Whenever the VP software uses the registers for incoming or outgoing data, *getIncoming*() or *setOutgoing*() methods are invoked respectively.

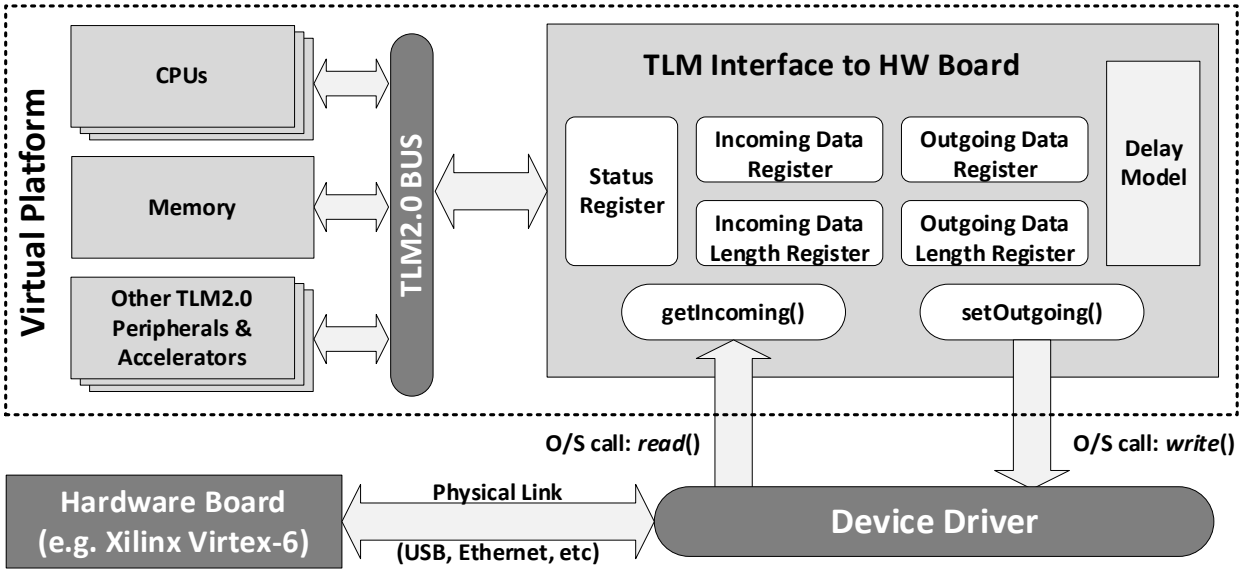

Figure 3: Communication mechanism between VP and hardware board

Another feature provided by the developed VP2HW library affects the delay of hardware board, which does not affect the timing of the TLM transactions. In other words, the designer can define the desired timing resolution. For this purpose, the TLM interface of introduced framework provides a delay model which corresponds to the desired timing accuracy of TLM transactions. Such an approach is important in case the employed hardware device is not the implementation medium of final system. Finally, the developed library is protocol-independent, and consequently any potential/customized communication protocol between VP and hardware can be selected

3.1.5 Implementation of the HotTalk FPGA Transactor

In order to realize the VP2HW communication, the HotTalk API provides a communication protocol consisted of a transactor module mapped onto the FPGA. This transactor handles the incoming or outgoing data, thus establishing the connection between the hardware IP and the communication port (in our case Ethernet). As long as custom HW IPs have been developed, it is possible to verify their functionality under real-world scenarios, using the HotTalk FPGA transactor. This approach was proven that detects bugs and performance issues that cannot be foreseen when solely software testbenches are employed.

The overall communication flow established by the transactor relies on a producer-consumer scheme and it is depicted in Figure 4. More specifically, the testbenches to the host side (either in native code or through a VP) generate the necessary input traces for HW IPs. Then, the device driver assembles input data to protocol frames. The protocol in its current version is built on top of raw Ethernet frames. In order to maximize the communication throughput, the size of the header packets is limited to 8 bytes, referring to the destination address (6 bytes) and the length of the frame (2 bytes). The rest packet (MTU-8 bytes) is used for data transfer. On the FPGA side, the HotTalk FPGA transactor is responsible for serving the RX requests from host's network interface controller (NIC) (initiated from device driver) to the custom HW IPs. A similar approach is followed for the reverse data direction, in the case that FPGA is forwarding results to the testbench.

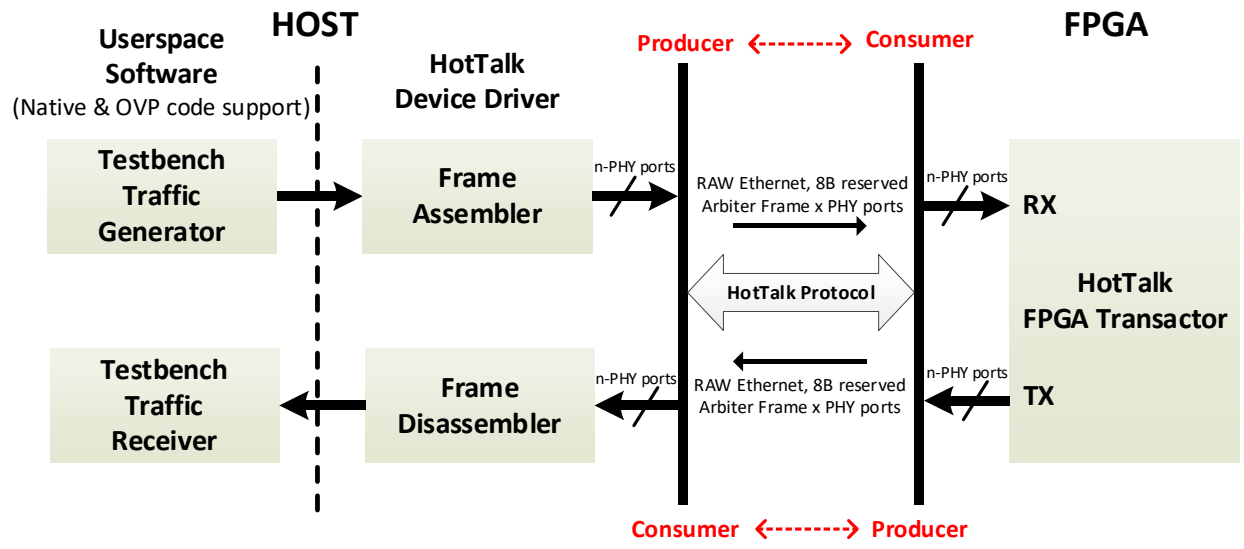

Figure 4: The communication flow established in the transactor

As we have already mentioned, the communication between the developed IP kernels and the VP plays an important role in the overall system's performance. To facilitate the data transfer, we designed a communication scheme depicted in Figure 5, which combines off-the-self and custom-made kernels to implement raw Ethernet connection. This scheme is supported by (i) a communication back-end IP, named ComCore, (ii) an arbiter and (iii) the Design Under Test (DUT).

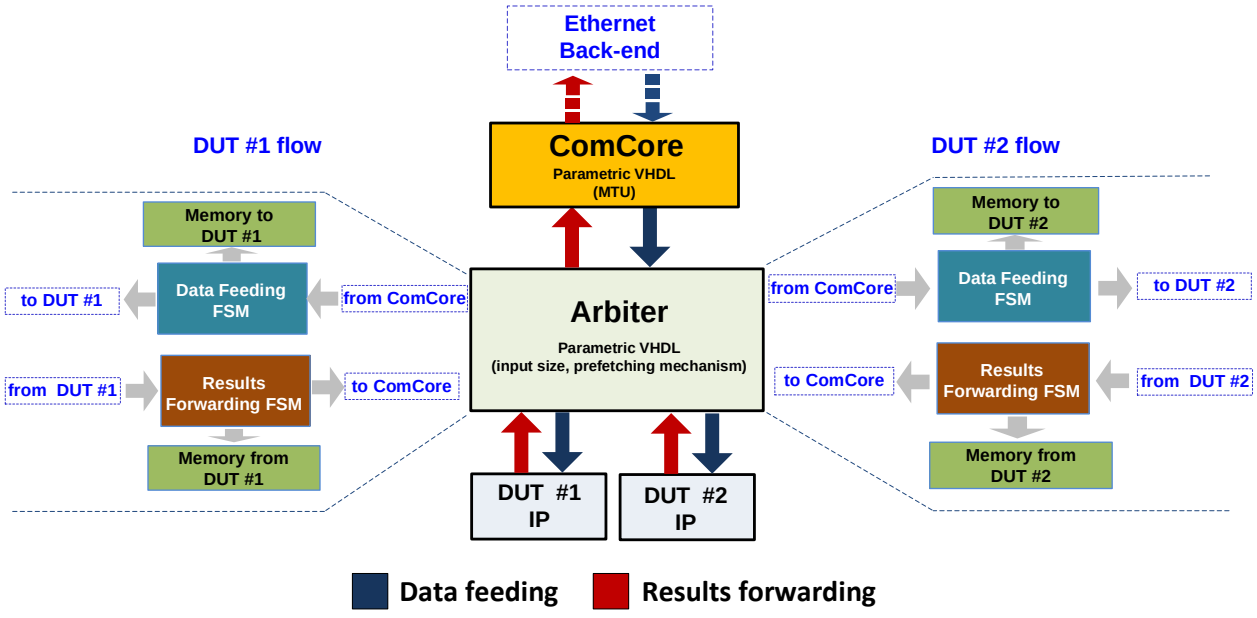

Figure 5: Architecture of the employed FPGA transactor

For this purpose, we utilized the integrated Ethernet physical layer (PHY) chip found in FPGA to perform character encoding, transmission, reception and decoding. Building on top of PHY, we developed a custom Rx/Tx controller based on the Ethernet MAC IP core from OpenCores[2], which implements the CSMA/CD LAN in accordance with the IEEE 802.3 standards. The resulting back-end component, as shown in Figure 5 , supports the custom-made kernels used to construct and distribute heterogeneous packets to the various hardware modules of the target system. Specifically, we use “ComCore” for the split/synthesis of large data packets to frames of 1,500 bytes. Moreover, ComCore handles the MAC controller’s signalling to provide a simple Wishbone interface to the remaining FPGA modules.

The *Arbiter* component controls the communication channel to avoid conflicts between the processing modules. Among others, this arbiter provides prioritized communication for these modules based on a round-robin polling. In order to support a pipeline operation, the Arbiter pre-fetches packets while the remaining modules process already fetched data. Such a functionality exploits the DMA capabilities of state-of-the-art off-the-self NICs: Rx and Tx operations can be performed in parallel by dedicated Ethernet PHY chips, while at user space, distinct threads can execute independent tasks of the algorithm being scheduled dynamically by the operating system. Additionally, we have to notice that the proposed scheme can support multiple channels of Ethernet communication between the VP and the FPGA, more precisely one channel per DUT, depending on the distinct Ethernet ports found on the target FPGA board. For instance, in Figure 5 an arbiter with two DUTs is depicted. In such a case, the arbiter uses headers to designate the receiving/transmitting module from/to the VP. Regarding the arbiter's architecture, it incorporates two distinct finite state machines (FSMs), each of which is dedicated to transmission and reception procedure respectively. Such a design approach allows independently data feeding and results forwarding, as long as the FIFO memory dedicated to each FSM is big enough to store the data. Thus, the arbiter could support a parametric pre-fetching mechanism (with configurable amount of pre-fetched data frames), so that the DUT IP can work on high utilization ratio without waiting the VP2HW communication at every iteration step.

As compared to existing implementation**,** the ComCore developed for the scopes of Plug&Play features batch mode transmission-reception of parametric-length frames, error and collision detection. In conjunction with a Linux kernel driver developed to cooperate with the introduced framework, it also offers recovery from system collisions. On the software side, we developed a device driver to support the communication between the Ethernet Network Interface Card (MII standard compatible NIC) and the VP. The driver provides the interrupt handling of the NIC to enable the asynchronous communication within the system via a transparent interface to the user space. In more detail, the deriver performs read and write operations to a Linux character device file, whereas it was developed as a loadable kernel module for supporting raw Ethernet frames of 1,500 bytes (MTU).

[2] OpenCores, Ethernet MAC 10/100 Mbps (https://opencores.org/projects/ethmac)

3.2 Automatic optimization for FPGA accelerated kernels

Field Programmable Gate Array (FPGA) devices have been proven to be a promising acceleration alternative when programmed with optimal configuration[3]. Designing hardware for FPGAs is a demanding and time-consuming process. Although algorithmic level methodologies, such as High-Level Synthesis (HLS), have provided abstraction layers compared to Register Transfer Level (RTL) (i.e., Verilog and VHDL), developers still need to familiarise with hardware concepts in order to efficiently accelerate computationally intensive parts of an application. Selecting manually the appropriate HLS directives is a difficult task even for human experts, mainly because of the large design decision space and its variation through different FPGAs. In this direction, Xilinx introduced Vitis[4], a framework that provides a unified OpenCL interface for programming edge (e.g., MPSoC ZCU102, ZCU104) and cloud (e.g., Alveo U50, U200) devices. Even though Vitis simplifies the designing process and lets developers focus on performance optimizations, it is not able to take into account the architectural characteristics of the target FPGAs (available resources etc.). In this context, it is essential, for both academic and industrial purposes, to build tools that are able to automatically specify the appropriate HLS directives with respect to a specific architecture.

The SERRANO project aims to fill the gap in the hardware designing process for FPGAs by providing a source-to-source compiler that automatically applies HLS directives on C/C++ kernels and identifies the optimal kernel based on a set of pre-defined criteria for a specific device.

3.2.1 Basic concepts of source-to-source compilers

A source-to-source compiler translates the source code of a program written in a programming language to an equivalent representation in the same or a different programming language. Source-to-source translators convert between programming languages that operate at approximately the same level of abstraction, while a traditional compiler translates from a higher to a lower-level programming language. For example, it may translate a program from Python to JavaScript, while a traditional compiler translates from a language like C to assembly or Java to bytecode.

[3] Danopoulos, D., C. Kachris, and D. Soudris. "A Quantitative Comparison for Image Recognition on Accelerated Heterogeneous Cloud Infrastructures." *Heterogeneous Computing Architectures*. CRC Press, 2019. 171-189.

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

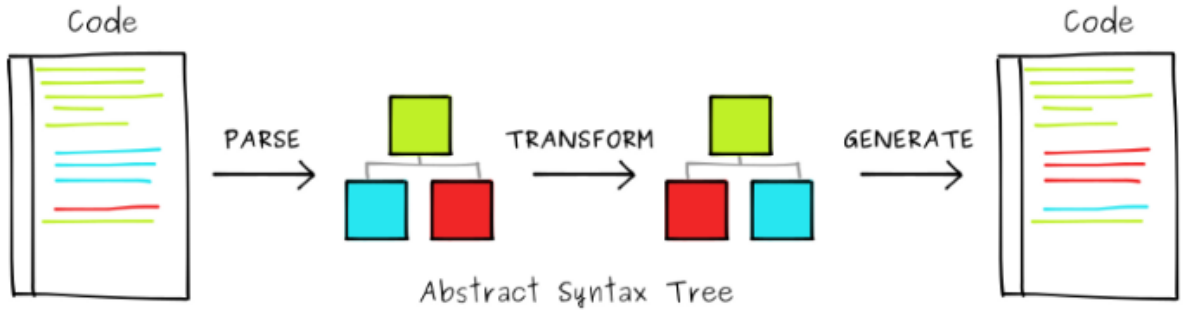

Figure 6: Source-to-source compiler architecture

Figure 6, shows the components of the source-to-source compiler (also referred to as transpiler) architecture. Transpilers parse the source code and extract the tokens (language keywords, variables, literals, operators etc.), which are the basic building blocks of the language. This phase is a combination of lexical analysis (conversion of the high-level input source code into a sequence of tokens) and syntax analysis (syntactical structure analysis and verification that the given input is in the correct syntax of the programming language or not). Transpilers know how to do this because they understand the syntax rules of the input language. Given this understanding, transpilers build the Abstract Syntax Tree (AST). The next phase is to transform the AST to suit the target language. This is used to generate the code in the target language.

Source-to-source compilers are extensively used in translating legacy code to use the next version of the underlying programming language or an application programming interface (API) that breaks backward compatibility. They are able to perform automatic code refactoring which is useful when the programs to refactor are outside the original implementer's control (for example, converting programs from Python 2 to Python 3, or converting programs from an old to a new API) or when the program size makes it impractical or time-consuming to refactor it by hand. In addition, transpilers are used as automatic parallelization tools that take as input a high-level language program and then transform the source code using parallel code annotations (e.g., OpenMP[5]) or language constructs (e.g., Fortran's `forall` statements).

3.2.2 Proposed optimization methodology

Employing High-Level Synthesis (HLS) for Field Programmable Gate Array (FPGA) design consists of applying different directives on a C/C++ source code. In this way developers are able to instruct the HLS compiler on how to synthesise the kernel that is going to be implemented on the Programmable Logic (PL). Leveraging the benefits of source-to-source compilers, SERRANO aims to propose a methodology for automatically optimising kernel using HLS. The optimisation process will consider the unique characteristics of different devices, providing optimised kernels for different FPGAs without human intervention.

[5] Mosseri, Idan, et al. "ComPar: optimised multi-compiler for automatic OpenMP S2S parallelization." *International Workshop on OpenMP*. Springer, Cham, 2020.

The proposed mechanism will take as input the source code of a C/C++ kernel and provide as output the kernel with added HLS directives, optimised with respect to a specific device. In this section, the fundamental components of the proposed methodology are analysed.

3.2.2.1 Kernel Source Code Analysis

The first step of the proposed methodology consists of analysing the kernel source code. This analysis mainly consists of identifying the points where High-Level Synthesis (HLS) directives will be added. The tool mainly focuses on kernel loops and arrays as the majority of performance related HLS directives concern loop (e.g., pipeline) and array (e.g., array partition) transformations.

After the parsing phase, the input source code is transformed and labels are added to the lines that the directives will be added. In addition, the iterations of each loop as well as the size of each dimension of each array are specified. This information will be used to create all the different HLS directives that will be applied to the corresponding loop or array. Figure 7 depicts the output of the kernel source code analysis phase for a simple input source code. As the example shows, the component adds label L1 after the array A declaration and label L2 after the loop. It also extracts information for each label. In particular, it specifies that the L1 label concerns an array named A, that is one-dimensional with size 256. As for the L2 label, it specifies that it concerns a loop with 512 iterations.

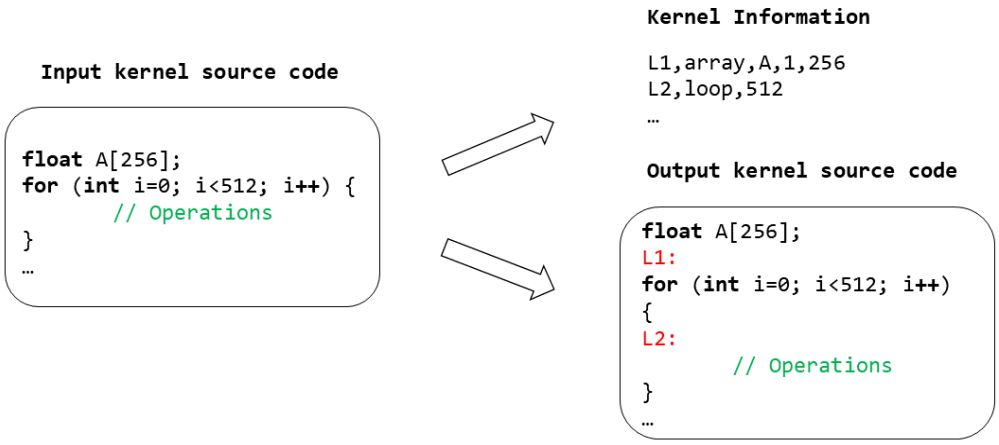

Figure 7: Kernel source code analysis phase example

3.2.2.2 High-Level Synthesis Directives Generation

After specifying the kernel action points and all the necessary information, the High-Level Synthesis (HLS) directives are generated for each label. In this version of our methodology variants of the pipeline and unroll directives are added to loops while variants of the partition directive are added to arrays. The aforementioned directives are briefly analysed in the following section.

Applied High-Level Synthesis Directives

- **Loop Pipeline**

The pipeline directive is used for reducing the initiation interval (II) of loops by allowing the concurrent execution of operations. A pipelined block of code can process new inputs every k clock cycles (cc), where k is the initiation interval. For instance, when an II of one is achieved, the loop is able to process new inputs every clock cycle.

In Vitis HLS 2021.2[6], the default behaviour of the pipeline directive is to generate a design with the minimum initiation interval according to the specified clock period constraint. The emphasis is on meeting timing, rather than on achieving II. If the tool cannot create a design with the specified II, a warning is issued and the design with the lowest possible II is created. This warning can be used to further analyse the design and determine what steps must be taken to create a new one that satisfies the required II (e.g., remove loop carry dependencies).

- **Loop Unroll**

The unroll directive creates multiple copies of the loop body in the RTL design, allowing some or all loop iterations to occur in parallel. Consequently, it leads to an increase in throughput as well as in the required device resources. The unroll directive allows the loop to be fully or partially unrolled. Fully unrolling a loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can run concurrently. Partially unrolling a loop lets the user specify a factor N, to create N copies of the loop body and reduce the loop iterations accordingly.

- **Array Partition**

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

High-Level Synthesis provides three types of array partitioning:

[6] Vitis High-Level Synthesis Documentation (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls)

1) Block where the original array is split into equally sized blocks of consecutive elements of the original array
2) Cyclic where the original array is split into equally sized blocks interleaving the elements of the original array
3) Complete which splits the original array to its individual elements

Loop Directives Generation

For each kernel loop the *#pragma HLS pipeline* and *#pragma HLS pipeline II=1* directives are generated regardless of the number of iterations. Their main difference relies on the fact that in *#pragma HLS pipeline*, the emphasis is given on meeting timing while in *#pragma HLS pipeline II=1* on achieving the lowest possible initiation interval. Creating a design with the lowest possible II, in certain cases, may lead to a decrease in operating frequency and hence an overall performance degradation. For this reason, both directives are applied.

Unrolling directives with different factors are also generated. In this case, the produced directives highly depend on the number of iterations. The proposed tool applies unroll factors that are powers of two (2, 4, 8 etc.) until a value called maximum unrolling factor. For loops with more than 64 iterations, the maximum factor is specified as 64, while for loops with less 64 iterations the maximum factor is defined from the following formula.

$$Maximum\ Unrolling\ Factor = \begin{cases} \dfrac{Loop\ Iterations}{2}, & Loop\ Iterations = Even \\ \dfrac{Loop\ Iterations}{2} - 1, & Loop\ Iterations = Odd \end{cases}$$

As the higher the unrolling factor is the more time-consuming the kernel synthesis will be, loops with more than 64 iterations are not completely unrolled.

Array Directives Generation

A similar approach is followed for the partitioning factor of each array dimension. In particular, for the block and cyclic partitioning types, if the array size is greater than 1024 the maximum factor is equal to 1024. For array sizes smaller than 1024, the maximum partitioning factor is defined as follows:

$$Maximum\ Partitioning\ Factor = \begin{cases} \dfrac{Array\ Size}{2}, & Array\ Size = Even \\ \dfrac{Array\ Size}{2} - 1, & Array\ Size = Odd \end{cases}$$

As for the complete partitioning type, it is not applied for arrays with more than 1024 elements which is the default of Vitis HLS 2021.2.

Figure 8 shows the output of the High-Level Synthesis directives generation phase for the kernel source code of Figure 7. For the label L1 directives, the maximum partitioning factor is equal to 128 (the size of A is less than 1024 and an even number) meaning that the applied factors for block and cyclic partitioning will be 2, 4, 8, 16, 32, 64 and 128. As the A size is less

than 1024 complete partitioning is also performed. On the other hand, for the label L2 directives, the maximum unrolling factor is equal to 64 (the loop iterations are more than 64) meaning that the applied factors for unrolling will be 2, 4, 8, 16, 32 and 64. Complete unrolling is not performed due to the number of iterations. Finally, the pipeline directives are added.

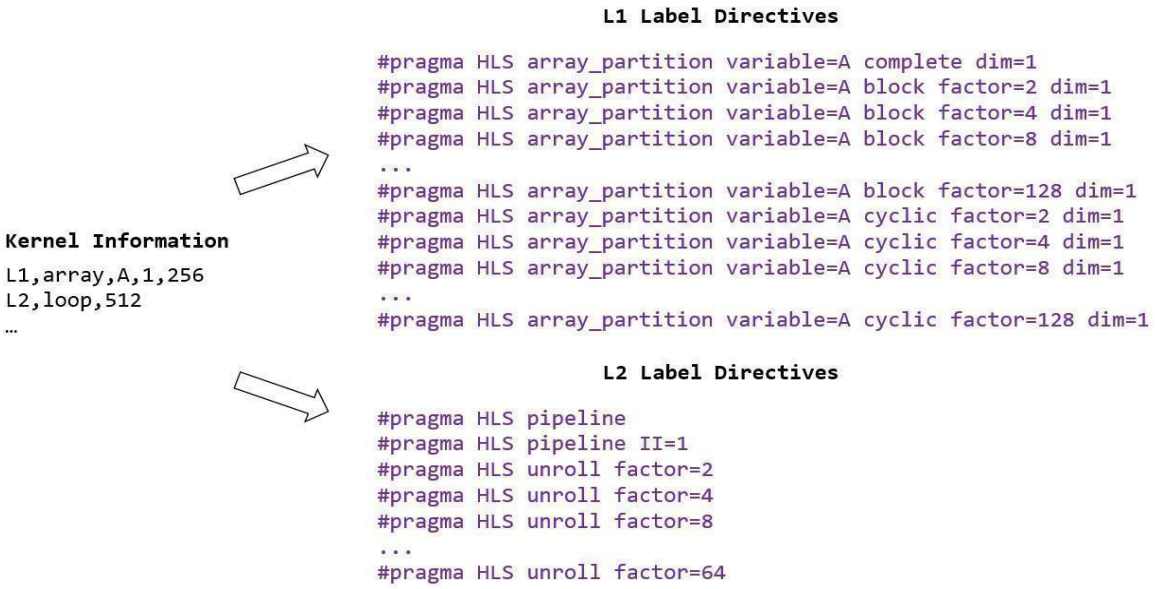

Figure 8: High-Level Synthesis directives generation phase example

3.2.2.3 Emulation of High-Level Synthesis Configuration

When the High-Level Synthesis directives are generated for all the identified action points in the kernel source code, different directives configurations are created. Suppose that the kernel source code has N action points and hence **N labels**. We also suppose that for the label with index **i**, $\mathbf{k_i}$ different directives were generated. A configuration can be represented as a **N-dimensional vector** where each coordinate is the HLS directive that is going to be added to the corresponding label of the kernel. For instance, a possible configuration for the previous example can be:

$$(\#\boldsymbol{pragma\ HLS\ array\_partition\ variable} = \boldsymbol{A\ complete\ dim} = \mathbf{1}, \#\boldsymbol{pragma\ HLS\ pipeline})$$

Each of these configurations, will be applied to the labelled kernel source code creating the modified kernel source code. The configuration can be simply applied by parsing the labelled kernel and adding each directive to the corresponding label. Figure 9, shows the modified kernel for the aforementioned configuration.

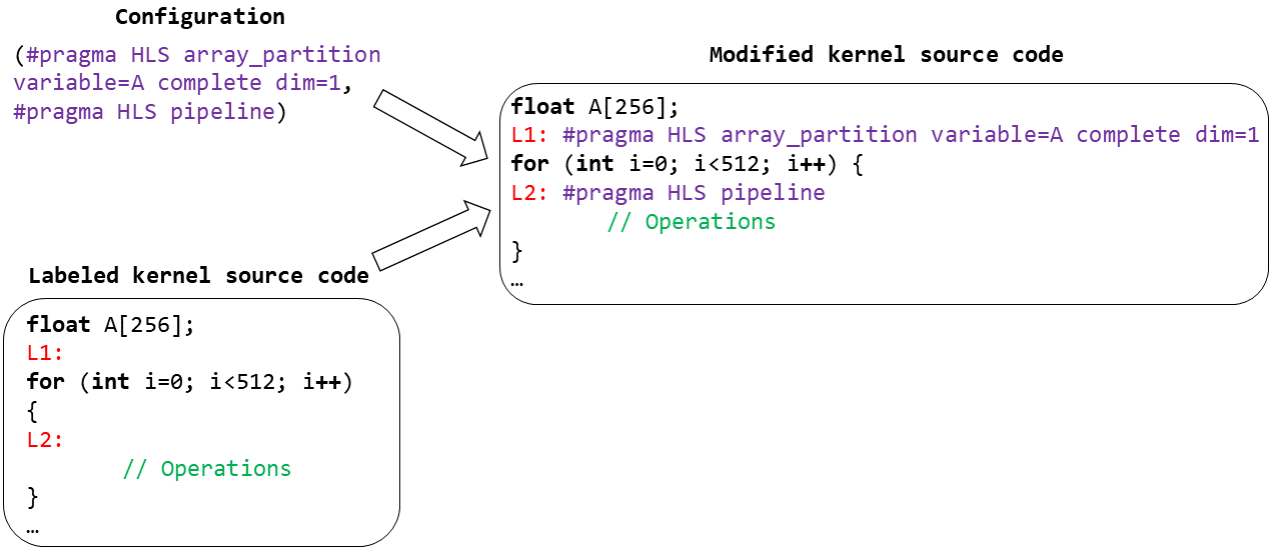

Figure 9: Modified kernel source code with HLS directives

The modified kernel source code is then synthesised using the Vitis HLS 2021.2 tool. The synthesis phase aims to provide an estimation of the kernel's latency and consumed device resources. In particular, it provides the utilisation percentage of the Block Random Access Memories (BRAM), the Flip Flops (FF), the Lookup Tables (LUT) and the Digital Signal Processing (DSP) units.

Besides the modified kernel source code, the top-level function (the kernel function from which the synthesis will start), the target Xilinx FPGA device (e.g., Alveo U50 and MPSoC ZCU102) as well as the operating frequency are specified. This information is defined in a TCL script, created from a generator, that is used as input for the vitis_hls command. The overall high-level synthesis configuration emulation phase is presented on Figure 10.

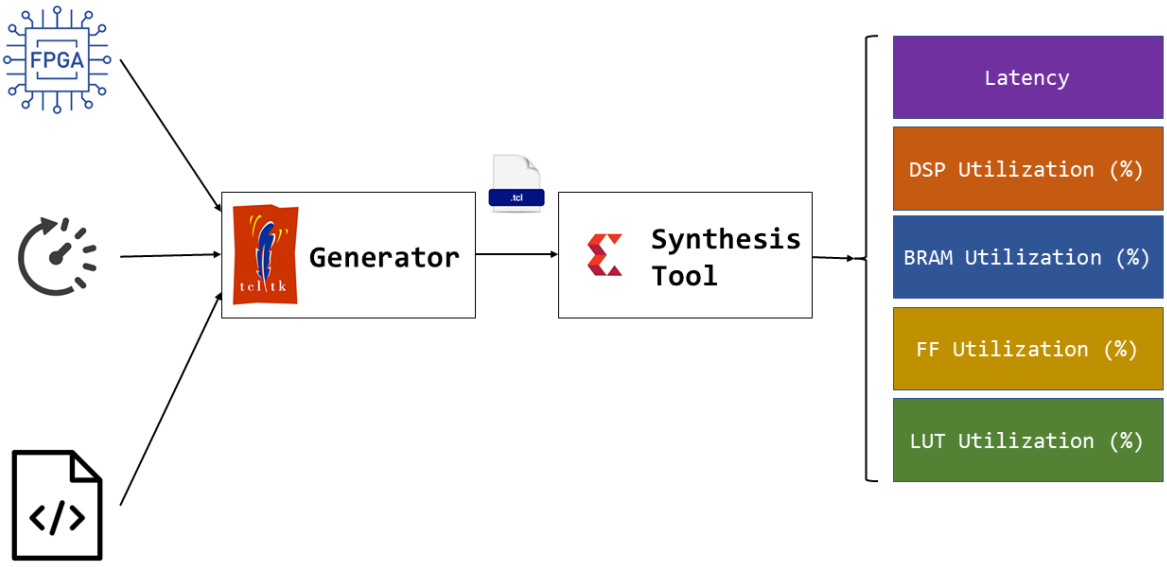

Figure 10: High-Level Synthesis configuration emulation overview

After specifying the way, the modified kernel is synthesised and the estimated latency and consumed resources are acquired, to find the optimal (minimum latency and consumed resources) all of the different configurations have to be checked.

3.2.2.4 Genetic Algorithm based Design Space Exploration

Performing exhaustive Design Space Exploration (DSE) is a prohibitive process. Suppose that the kernel source code has N action points and for the action point with index i, k_i different directives were generated. The computational complexity of the exhaustive DSE is given from the following formula.

$$Exhaustive\ DSE\ Complexity = O(k_1 * k_2 * \dots * k_N)$$

In addition, the kernel synthesis is a time-consuming process, as it takes at least 30 seconds to execute. For instance, for a kernel with 1 for loop with 8 produced directives and an array declaration with 15, 120 different configurations will be examined and the exhaustive DSE will last at least 1 hour. In realistic scenarios where kernels with multiple loop definitions and array declarations exist, the DSE may need multiple hours or even days. Furthermore, the more High-Level Synthesis optimizations we apply, the more time-consuming the emulation becomes. It is evident that the proposed methodology requires a more efficient way to traverse through the configurations of the design space.

Genetic algorithms (GA) are good candidates to tackle such complex explorations since their behaviour is very similar to the one of a designer: they iteratively improve a set of solutions (i.e., the alternative designs) using the results of their evaluations as a feedback to guide the search in the solution space. The main advantage is that a GA-based approach can be fully

automated, providing better performance and faster explorations[7]. The following section presents some basic concepts from the genetic algorithm theory.

Genetic Algorithm Theory

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This category of algorithms reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce the offspring of the next generation. The process of natural selection starts with the selection of fittest individuals from a population. These individuals produce offspring which inherit the characteristics of their parents. The parents with the better fitness, produce offspring that have a better chance at surviving in the next generation. This process keeps on iterating and at the end, a generation with the fittest individuals is found. This notion can be applied to search problems.

Each genetic algorithm can be divided in five distinct phases:

- **Initial population definition**

The process begins with a set of individuals which is called a *Population*. Each individual is a solution to the problem you want to solve. An individual is characterised by a set of parameters (variables) known as *Genes*. Genes are joined into a string to form a *Chromosome* (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet.

- **Fitness Function**

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

- **Selection**

The idea of the selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chances to be selected for reproduction.

- **Crossover**

The crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. *Offspring* are created by exchanging the genes of parents among themselves until the crossover point is reached. The new offspring are added to the population.

[7] Ferrandi, Fabrizio, et al. "A multi-objective genetic algorithm for design space exploration in high-level synthesis." *2008 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2008.

- **Mutation**

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped. Mutation occurs to maintain diversity within the population and prevent premature convergence.

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem. Figure 11 presents the way the previously analysed phases are executed on a genetic algorithm.

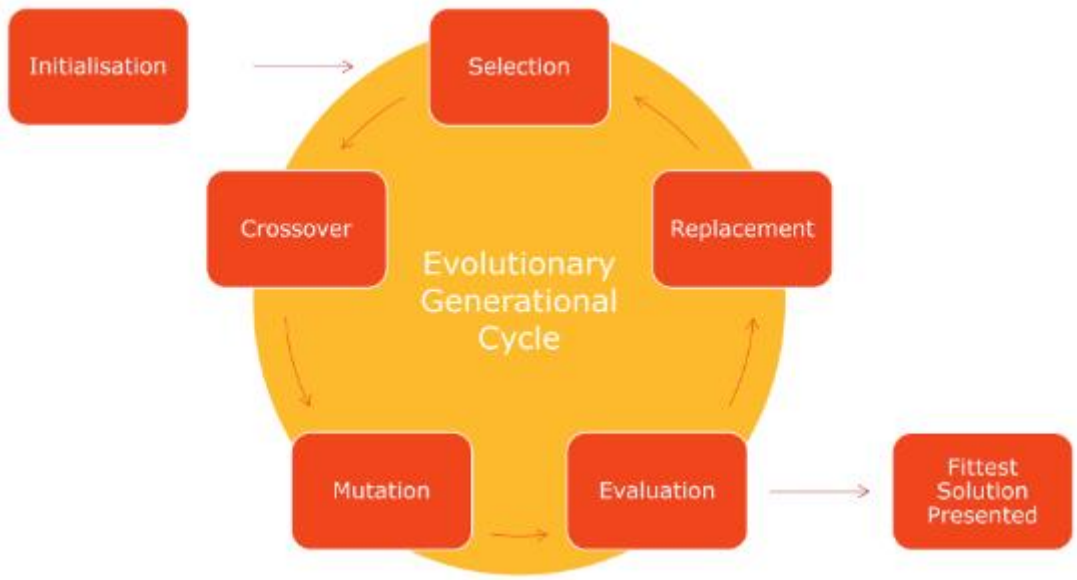

Figure 11: Phases of a genetic algorithm

Genetic algorithms and DSE

Genetic algorithms have been extensively used for solving search problems over the years. To simplify their usage, multiple genetic algorithm libraries have been proposed. In our methodology the Python Multi-Objective Optimization (PyMOO[8]) library is used, due to its easy-to-use Application Programming Interface (API).

To use the PyMOO library we have to define our problem in the form of a multi-objective optimization problem. Suppose that a kernel source code has **N labels**. We also suppose that for the label with index **i**, $\mathbf{k_i}$ different directives were generated. As it was previously mentioned, a configuration can be represented as a **N-dimensional vector** where each coordinate is the HLS directive that is going to be added to the corresponding label of the

[8] Python Multi-Objective Optimization Library (https://pymoo.org)

kernel. To comply with PyMOO's terminology, the directives of each label are mapped to an integer in the **[1, k_i]** interval where **i=1, 2, ..., N**. Figure 12, shows the aforementioned mapping for the HLS directives of the sample code loop.

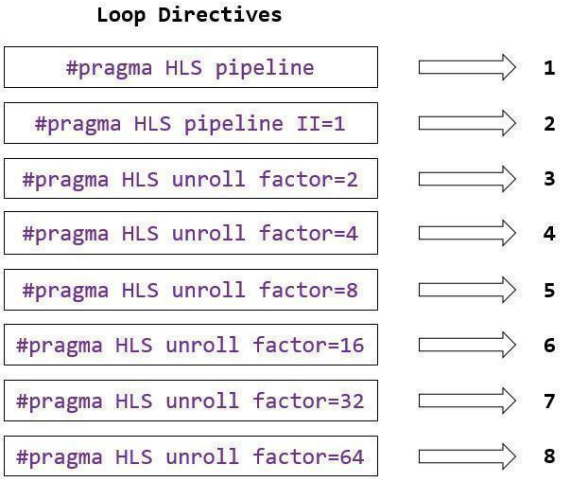

Figure 12: Mapping example

By mapping a configuration vector to a vector of integers we are able to define the multi-objective optimization problem, as follows:

$$Latency(x_1, x_2, \dots, x_N)$$

$$BRAMUtil(x_1, x_2, \dots, x_N)$$

$$LUTUtil(x_1, x_2, \dots, x_N)$$

$$FFUtil(x_1, x_2, \dots, x_N)$$

$$DSPUtil(x_1, x_2, \dots, x_N)$$

$$s.t \quad BRAMUtil(x_1, x_2, \dots, x_N) < 100$$

$$LUTUtil(x_1, x_2, \dots, x_N) < 100$$

$$FFUtil(x_1, x_2, \dots, x_N) < 100$$

$$DSPUtil(x_1, x_2, \dots, x_N) < 100$$

$$where \qquad 1 \leq x_1 \leq k_1$$

$$1 \leq x_2 \leq k_2$$

$$\dots$$

$$1 \leq x_N \leq k_N$$

$$x_1, x_2, \dots, x_N \in N$$

In this form the problem can be simply implemented using the problem definitions provided by PyMOO library. After defining the problem, one of the available algorithms[9] is initialised, the termination criterion is defined and the genetic algorithm starts its execution. Then by executing multiple experiments the algorithm's parameters are specified. It should be noted that the evaluation of the configurations of a generation are executed in parallel to further speedup the DSE process.

3.2.3 Evaluation

To evaluate the proposed methodology, an application provided by SERRANO's Use Case (UC) providers was used. In particular, the Kalman filter algorithm from InbestMe's UC was optimised through the proposed methodology for the MPSoC ZCU104 Xilinx FPGA. In the following paragraph, a brief description of the Kalman filter is provided. More information concerning the proposed parallelisation strategy can be found in deliverable 4.1.

Kalman filtering, also known as Linear Quadratic Estimation (LQE), is an algorithm that uses a series of measurements observed over time, including statistical noise and other inaccuracies, and produces estimates of unknown variables. The algorithm works by a two-phase process. For the prediction phase, the filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with greater certainty. Kalman filtering has numerous technological applications varying from guidance, navigation and control to time series analysis used for topics such as signal processing and econometrics.

3.2.3.1 Kalman Filter Acceleration Baselines

To evaluate the proposed methodology, the execution time and the consumed resources of the Kalman filter algorithm have to be identified. As a first step, the kernel is synthesised (using vitis_hls command) without applying High-Level Synthesis directives. The default optimisations that Vitis HLS 2021.2 applies (e.g., automatic loop pipelining/unrolling) were also disabled. The clock frequency was specified to 300 MHz. The kernel latency estimation is equal to **45.2 ms**. Table 1 presents the kernel synthesis results for the target device.

Table 1: Resources utilization for unoptimized Kalman filter kernel

Resource	Used	Total	% Utilisation
BRAM_18K	*606*	*624*	***97***
DSP	*2*	*1728*	***~0***
FF	*4198*	*460800*	***~0***
LUT	*12239*	*230400*	***5***

[9] Python Multi-Objective Optimization Library Algorithms (https://pymoo.org/algorithms/index.html)

URAM	*0*	*96*	***0***

The unoptimized kernel version of the Kalman filter consumes 97% of the available BRAMs while the DSP, FF and LUT consumption is not significant. To identify how the default Vitis HLS 2021.2 optimisations affect the kernel performance as well as the resources consumption, the kernel was synthesised with the optimisations enabled. The kernel latency estimation is equal to **1.4 ms** while the resources consumption is presented in Table 2.

Table 2: Resources utilization for Kalman filter kernel with default optimizations

Resource	Used	Total	% Utilisation
BRAM_18K	***2238***	*624*	***358***
DSP	*2*	*1728*	***~0***
FF	*11299*	*460800*	***2***
LUT	*19326*	*230400*	***8***
URAM	*0*	*96*	***0***

Although Vitis HLS 2021.2 significantly decreases the kernel latency, by achieving a **x32.3** speedup, it is not able to create a design that can be mapped to the MPSoC ZCU104 FPGA (BRAM utilisation is greater than 100%). It is evident that the default optimisations performed by the tool do not take into account the characteristics of the target device, showcasing the need for human intervention to the design process.

The Kalman filter kernel was finally optimised by AUTH's team. After analysing the kernel and identifying the action points, the appropriate High-Level Synthesis directives were added to the kernel source code. In this kernel's version, the latency estimation is equal to **1 ms** while the resources consumption is presented in Table 3.

Table 3: Resources utilization for Kalman filter kernel optimized by AUTH's team

Resource	Used	Total	% Utilisation
BRAM_18K	*604*	*624*	***96***
DSP	*128*	*1728*	***7***
FF	*122649*	*460800*	***26***
LUT	*111729*	*230400*	***48***
URAM	*0*	*96*	***0***

AUTH's team was able to achieve a **x45** speedup compared to the fully unoptimized kernel. In addition, the DSP, FF and LUT utilisation are increased by **5%**, **26%** and **42%** respectively.

Nevertheless, the Kalman filter acceleration was a time-consuming process. Identifying the appropriate HLS directives with respect to the target device's available resources took more than **48 hours**. In the following section, the kernel optimisation results using the proposed methodology are presented.

3.2.3.2 Kalman Filter Automatic Acceleration

The proposed tool is compared with two different baselines. On one hand, the output is compared with the optimised kernel from AUTH's team. On the other hand, a naive automatic optimiser is used. The naive optimiser creates random configuration vectors (i.e., vectors with High-Level Synthesis directives) and evaluates as many as possible in a predefined amount of time.

The proposed methodology is based on the elitist Non-dominated Sorting Genetic Algorithm II (NSGA-II[10]). As in each Genetic Algorithm (GA), in NSGA-II the number of generations and the population size have to be defined. After some parameter experimentation, the number of generations was set to **12** while the population size (number of different High-Level Synthesis configurations that are evaluated in each generation) was set to **40**. To speed up the evaluation of a population, the synthesis of the different configurations is parallelised. Finally, as the synthesis of some configurations is time consuming, whenever the synthesis process takes more than **1 hour**, it is stopped.

To have a fair comparison, the naive optimiser is able to evaluate simultaneously **40** configurations. In addition, it synthesises random configurations for **12 hours** which is the maximum time the NSGA-II is going to be executed. Figure 13 shows the kernel latency for all the different acceleration approaches.

[10] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.

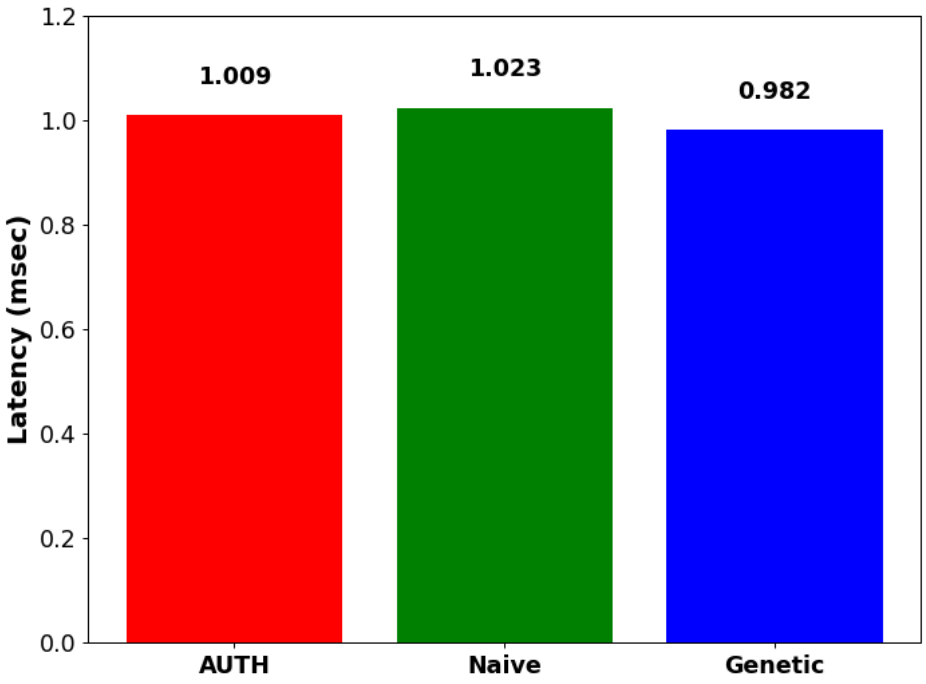

Figure 13: Latency for the accelerated Kalman filter kernel

The naive optimiser, after a 12-hour operation, creates an optimised Kalman filter version with a **13.7%** higher latency compared to the version AUTH provided. On the other hand, the proposed methodology, for a 12-hour execution, leveraging NSGA-II is able to provide an optimised kernel version with a **26.8%** lower latency compared to AUTH's version. These results showcase that the proposed methodology is able to optimise kernels using High-Level Synthesis without human intervention.

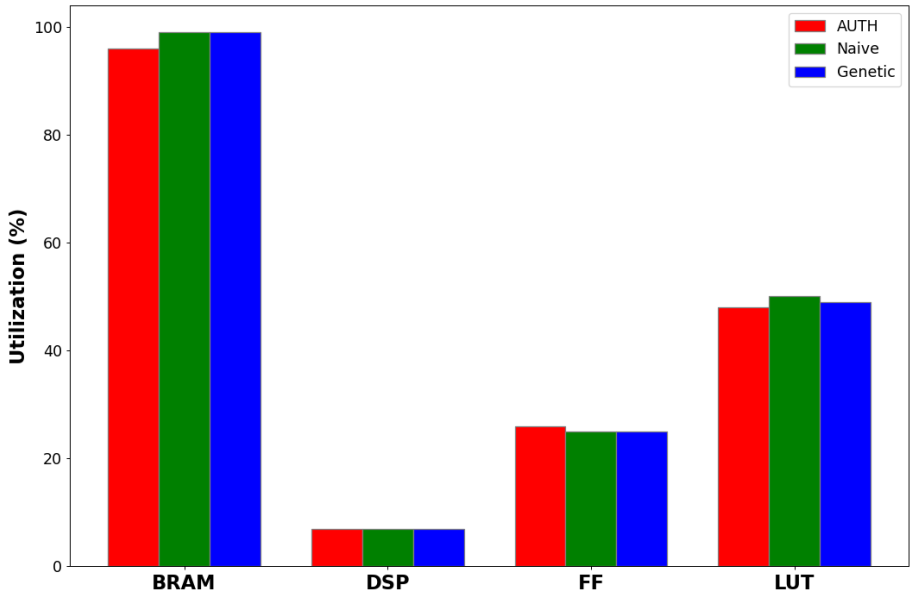

Figure 14: Resources utilization for the accelerated Kalman filter kernel

Figure 14, presents the resource utilisation percentage for the Block RAMs (BRAM), Digital Signal Processing units (DSP), Flip Flops (FF) and Look-Up Tables (LUT) for the previously mentioned acceleration strategies. It is evident that the BRAM, DSP, FF and LUT utilisation percentages are not significantly differentiated. Nevertheless, it is observed that the proposed mechanism is able to provide designs that respect the resources of the target device.

It should be noted that the proposed methodology does not simply provide the kernel with the minimum latency estimation. It provides the set of pareto optimal kernels, enabling users to choose between a set of kernels based on their requirements.

3.3 Automatic optimization for GPU accelerated kernels

In order to provide acceleration and energy efficiency and meet the desired requirements of SERRANO, at both the edge and the cloud, we need to apply extra optimizations on GPU kernels in order to fine-tune them in terms. However, and even though GPUs' programmability has seen significant improvement over the last few years, achieving close-to-peak performance remains a time consuming and demanding task even for the experts. Fine-tuning GPU programs for maximising performance or minimising power consumption remains a non-trivial and demanding task that has not been fully solved yet.

Therefore, we manage to auto-tune our CUDA accelerated use-cases through an auto-tuning framework we developed for the scope of this work. We consider kernels auto-tuning an automatic optimization process that automatically applies optimal (in terms of performance) block coarsening transformations across different applications, workload input sizes and GPU architectures. Our approach constitutes a machine learning based framework we developed that consists of an in-house built source-to-source compiler and a regression model that addresses the tuning problem automatically in a successful way. We evaluate our approach on Polybench benchmark on 5 different devices, achieving speedups up to x2.3 in terms of performance for unseen GPUs and unseen CUDA kernels in comparison with native implementations.

3.3.1 Block coarsening transformation

Block coarsening transformation is an optimization for parallel applications such as GPU programs. It refers to the kernel transformation that merges together the workload of 2 or more thread blocks and therefore reduces their total number by leaving the number of threads per block the same. Consequently, it merges multiple neighbouring blocks in order to deal with the problems associated with extensive fine-grained parallelism. Experimental results show that it can effectively reduce programs' execution latencies in almost 50% of the cases. However, coarsening's impact seems to be highly hardware and application dependent.

From an architectural perspective, GPUs map blocks to SMs (multiprocessors) and threads grouped in warps to CUDA cores. Therefore, reducing blocks per kernel with block coarsening, reduces the block's workload to be occupied on SMs and therefore the number of warps scheduled by each SM. Figure 15 depicts a simplified architectural view of mapping between software and hardware resources when a kernel is launched after block coarsening transformation.

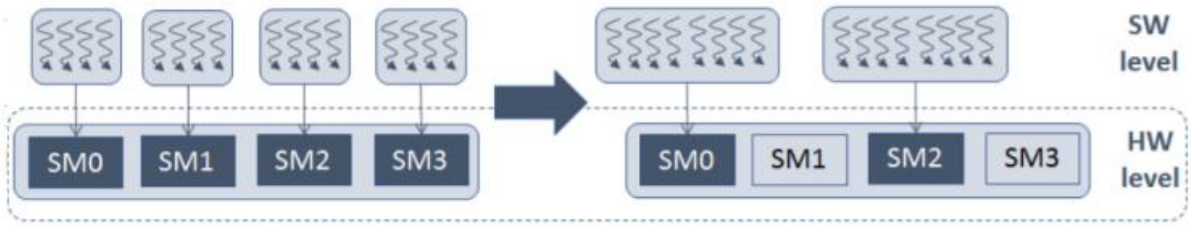

Figure 15: Block coarsening from hardware perspective

```
1 __global__ void square (float *in, float *out){
2     int gid=blockIdx.x*blockDim.x+threadIdx.x
3   out[gid]=in[gid]*in[gid]; }
```

Figure 16: Original squared CUDA kernel

Additionally, from a software perspective, in the CUDA kernel body, code duplication is based on calls to blockIdx.x, and includes all dependent instructions as depicted in Figure 16 and Figure 17. The rules for all CUDA applications are outlined in Figure 18.

```
1 __global__ void square_bc (float *in, float *out, int bc){
2     for (int index=0; index<bc; index++){
3     int gid=(blockIdx.x*bc+index)*blockDim.x+threadIdx.x;
4   out[gid]=in[gid]*in[gid]; }
```

Figure 17: Block coarsened squared CUDA kernel

However, applying block coarsening transformations in every accelerated use-case constitutes a time consuming and demanding task that often leads to suboptimal solutions and incorrect transformed codes. Therefore, for the scope of this work in order to automate the transformation process for all use-cases and for new unseen CUDA kernels, we develop a source-to-source compiler-tool based on the rules referred in Figure 4. Our tool constitutes a PERL written script that can successfully apply block coarsening transformation with a given coarsening factor to every CUDA kernel.

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

Figure 18: Rules for block coarsening transformation

Additionally, it can automatically explore all possible block coarsening factors in order to evaluate their performance and pick the optimal. We feed our tool with both the host and the device programs' source codes in order to configure the new grid size and the device kernel located in host and device files accordingly.

3.3.2 Auto-tuning model

After having the various block coarsened versions of the input kernels, our model aims to automatically select the best block coarsened version for each new unseen CUDA kernel for a specific NVIDIA GPU. The model works using program and hardware features in order to estimate with supervised learning each kernel version's performance and finally select the optimal.

In order to train an efficient supervised learning model, we need to feed it with features that successfully represent a big variety of CUDA programs, various input sizes and different architectures. Therefore, the full list of our model's selected features consists of the static features that represent the structure and the body of the CUDA kernel, the hardware features that constitute the GPU architecture, the potential block coarsening factors and the size of the kernel's input.

In order to extract the CUDA kernel's static features we base our work on the work published in 2019 (Guerreiro)[11] where they automatically extract features from PTX files by extracting the number of occurrences of each different instructions per GPU kernel. From a Python interface we automatically convert the CUDA kernels to PTX assembly files from which we extract a 101-size vector that counts the kernel's number of 101 different representative PTX instructions. Additionally, in order to build a portable model to adapt to the change of architectures, we need to feed our model with different architectural features. Figure 19 presents the full list of specifications, from both the computation and the memory scope, we choose to represent each GPU.

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 19: Hardware features extracted for each GPU

Finally, after concatenating both vectors, we add the workload input size of the input vector and the block coarsening factor of the kernel's version. The 114-size resulted 1D vector constitutes the selected features of each CUDA kernel's version.

After having prepared the features that will represent both applications and GPUs, we apply our source-to-source compiler to our kernels. The CUDA kernels we use for the evaluation of our framework in this work belong to Polybench-ACC opensuit[12], an open-source benchmark

[11] J. 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.

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

suite that contains CUDA kernels from datamining, linear algebra and stencils domain. Afterwards, we run the resulted coarsened CUDA kernels on 5 different NVIDIA GPUs in order to detect their execution latencies for each platform. Their hardware specifications are presented in Figure 21.

Finally, we use the selected features and the recorder latencies to train the model. We investigate 4 different regression models in order to select the most efficient. We make use of Simple Linear Regression, Decision Tree Regression, Random Forest Regression and **Extreme Gradient Boosting (XGBoost)** through the Scikit-Learn machine learning framework. Figure 6 depicts an overview of the training process of our framework. Note that we use the 90% of **Polybench** kernels to train our model and 10% to evaluate our model's prediction. Figure 20depicts the overview of the full training process.

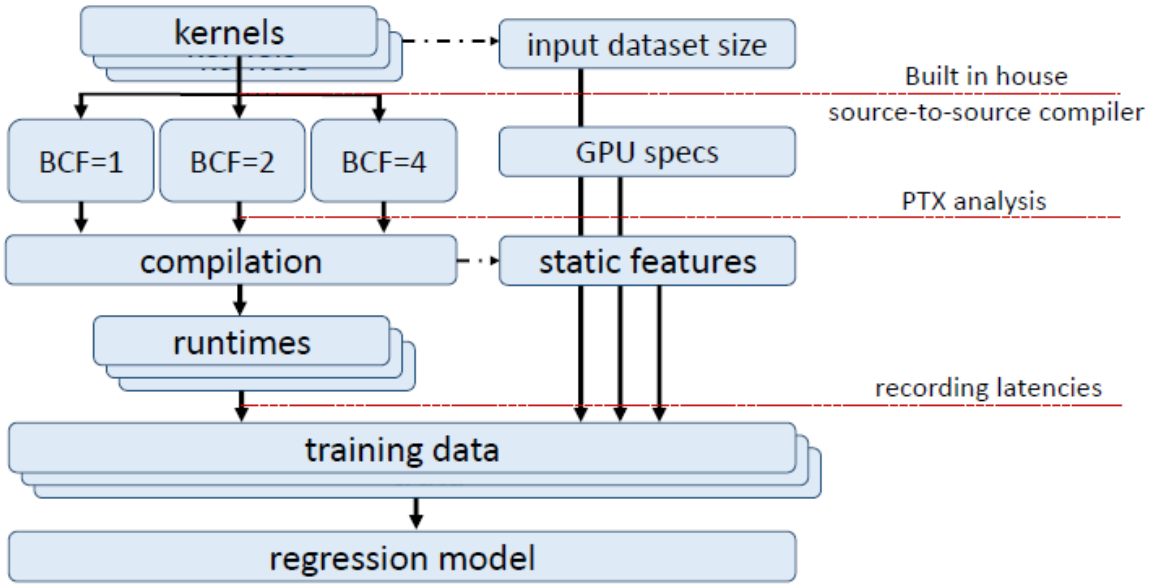

Figure 20: Training process

After the training process, our model is ready to make its predictions. It manages to predict the latency of a given unseen kernel, block coarsening factor and specific GPU and finally decide the optimal block coarsening factor for each kernel on a given architecture. The total prediction process is presented in Figure 22.

	TX1	Xavier NX	Xavier AGX	GTX 1070	V100
Architecture	Maxwell	Volta	Volta	Pascal	Volta
Comp. Cap.	5.3	7.2	7.2	6.1	7.0
SMs	2	6	8	15	80
CUDA cores	256	384	512	1920	7680
Memory size (GB)	4	8	32	8	32
Shared Mem / SM (KB)	48	48	48	48	96

Figure 21: HW specifications of used GPUs

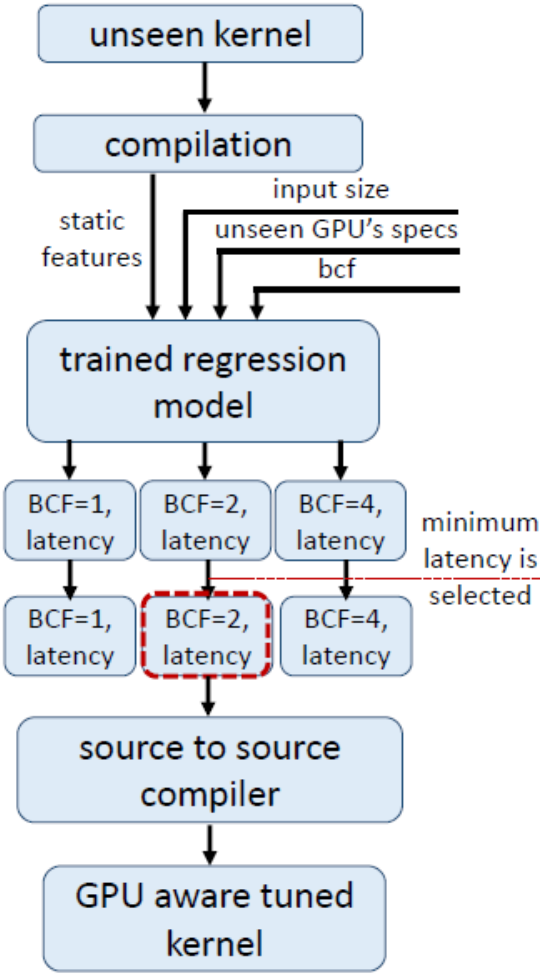

Figure 22: Prediction process

3.3.3 Evaluation

We apply our trained model to the 10% of Polybench suite's kernels versions and we evaluate its prediction. In order to evaluate the 4 different regression models examined in this work, we use mean square error (MSE) and R2 comparison. Figure 23 and Figure 24 present MSE and R2 of each model we used. As depicted, XGB presents the most efficient solution with 0.02 MSE and 0.88 R2 and therefore constitutes our selected regression model.

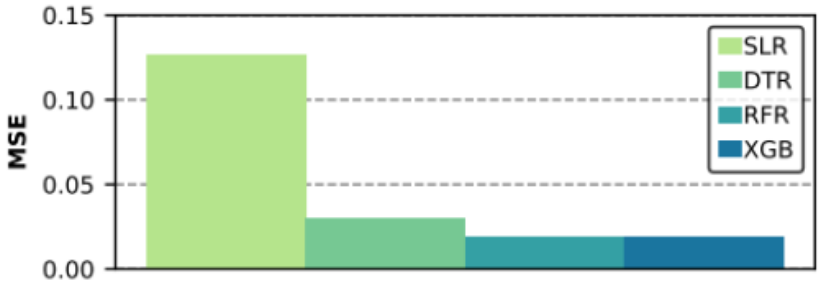

Figure 23: MSE of regression models

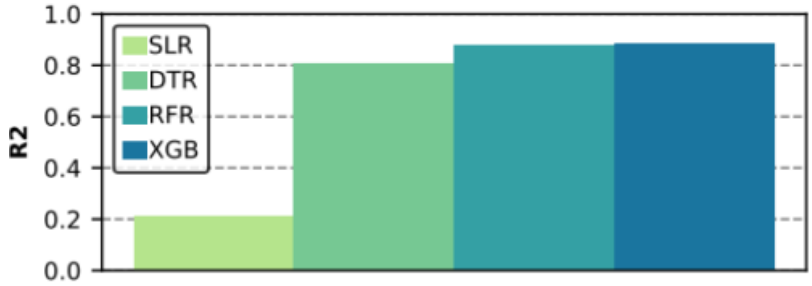

Figure 24: R2 of regression models

Using XGB regression model, our predictions lead to selection of optimal coarsened kernels with speedups up to x2.3, for **new unseen GPUs** and **new unseen kernels** to the training model, in comparison with native implementations.

3.4 Runtime memory management for FPGA sharing

3.4.1 Integration of Multiple Accelerators on FPGAs

The process of implementing designs on FPGAs is performed by the synthesis tools after the designers have tested and refined their circuits at algorithmic and functional level and have validated the circuit's behaviour at different input and timing conditions. The synthesis tools are responsible for placing the provided circuits on the hardware platform and assigning memory (BRAM), computational (DSP) and reprogrammable (Slices) resources to each circuitry design. The number of utilized resources is defined by the circuit's requirements for storing and computing operations, and the number of the utilized resources have a direct impact on the power consumption, with more resources leading to increased power expenses.

In a typical designing methodology of many accelerators on a FPGA device, the designers synthesize each kernel independently and the synthesis tools assigns to each accelerator the resources that are required for its storing and computing operations. Once, those resources have been allocated to a specific kernel, they are considered utilized and they cannot be shared with other designs or used for different purposes, until the FPGA platform is reprogrammed. This synthesis mechanism is illustrated on the Figure 25 below.

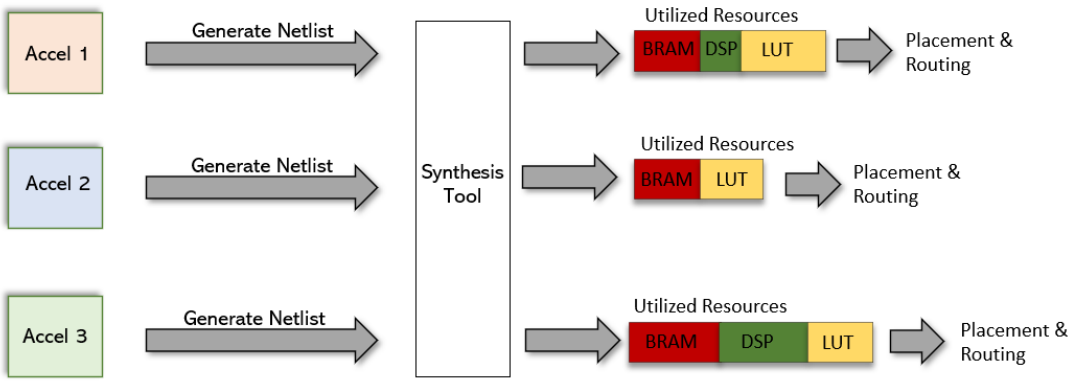

Figure 25: Typical flow of synthesizing multiple accelerators

In many cases the designers are required to incorporate multi accelerators on the same device in order to achieve their designing goals. Some of the scenarios that highlight the need of having many accelerators sharing the same platform are listed below.

- Designing accelerators that implement different flavours of the same or of multiple algorithms. There are application scenarios that require the same computational kernel to be executed multiple times for different use-case scenarios (e.g., executing multiple filters) and each kernel version to be optimized for a specific set of parameters. In addition, approximate accelerators can be designed and deployed on the same platform trading-off performance for accuracy. In the aforementioned cases, multiple accelerators can be designed on the same hardware platform and be executed in rotation to one another based on the provided user requirements. By having multiple kernel versions implemented on the same device, the latency overhead by reprogramming the FPGA device at runtime can be eliminated. The Figure 26 below depicts the described execution scenario.

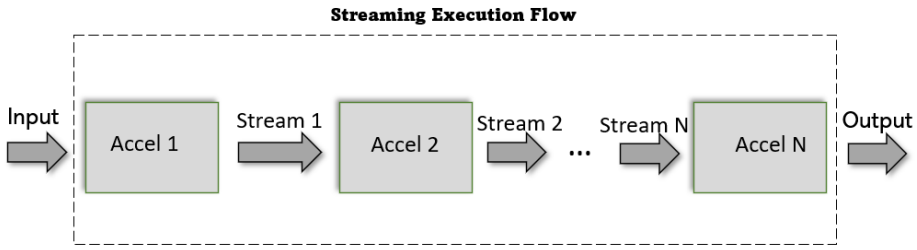

Figure 26: Streaming execution flow

- To increase the overall application's performance. FPGA platforms that are used in data centers have more available resources than the typical embedded FPGAs. Those hardware resources can be allocated to multiple kernels in order to increase the application's throughput. In this execution scenario, the designers implement the same computational kernel multiple times with each accelerator performing computations on chunks of the input data. The parallel execution of

multiple accelerators processing the same input leads to an increase in the application's throughput.

- To accelerate successive execution of multiple algorithms. There are applications that require multiple algorithms to process the input data in a sequential flow. For example, executing a pre-processing kernel first and then applying a classification algorithm on the pre-processed data. In those execution scenarios multiple accelerators are implemented on the same hardware platform and are launched once they have received through streams the input that is required for their execution.

The

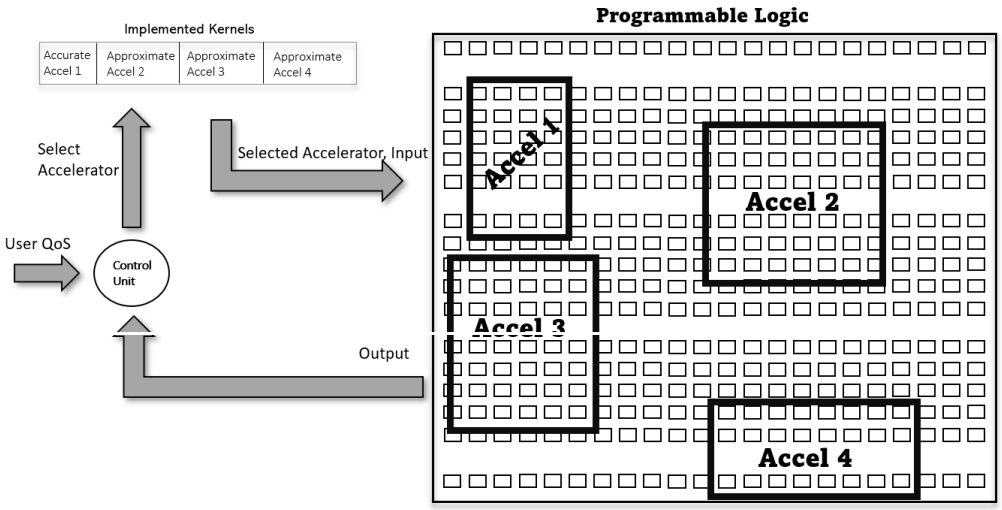

Figure 27 below shows the described execution flow.

Figure 27: Implementation of Multiple Accelerators on FPGA

3.4.2 Limitations of shared hardware platforms

When multiple accelerators are implemented on the same device and their execution order is not completely in parallel then an amount of hardware resources is under-utilized. The phenomenon of resource under-utilization is due to the standard synthesis mechanisms of binding memory and computational resources during the accelerator's RTL synthesis and placement process.

Among the different types of FPGA resources (BRAM, LUT, DSP), the "resource that starves faster"[13] has been proved to be the BRAM memories, creating a bottleneck to the deployment of multiple accelerators on one device. Similarly, a survey[14] has shown that on average 69% of the total accelerator's area is occupied by the memory units. The Figure[15] 28 below shows the variations in the utilization of different FPGA resources when multiple accelerators that perform K-means clustering are integrated on the same FPGA.

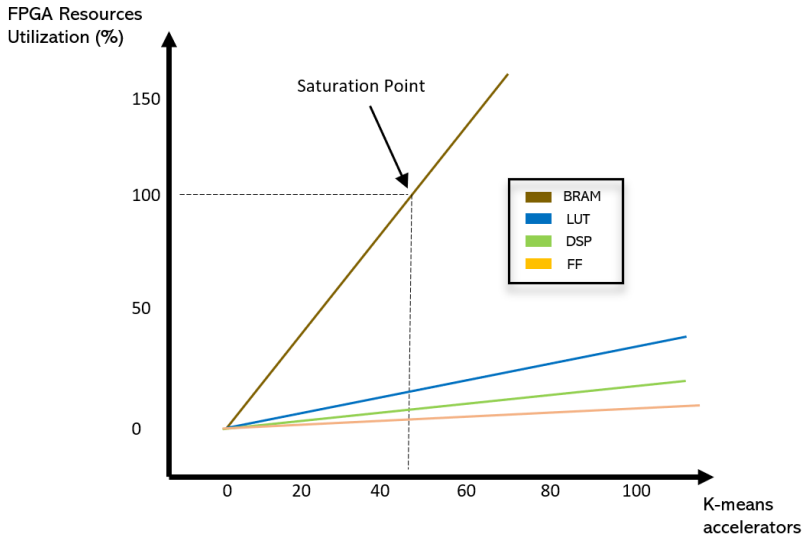

Figure 28: FPGA's resources saturation

Current High-Level Synthesis workflows don't support memory management techniques that are used widely by software engineers to dynamically allocate and de-allocate memory resources at runtime. State-of the art HLS methodologies require from the designer to perform static analysis of the application and determine the worst-case memory utilization[16]. Once the application's memory needs have been found then the designer assigns statically the BRAM resources that are required for the application's execution.

3.4.3 Dynamic Memory Management in Software Applications

Memory management techniques that are used in software engineering can be adjusted and used in HLS in order to mitigate the effects of resource under-utilization and deploy more accelerators on the FPGA's fabric. In software applications, functions that are commonly used to dynamically handle the application's requests for memory allocation and de-allocation are *malloc()* and *free()*. Through those functions the developers manage the heap without directly

[13] Diamantopoulos, Dionysios, et al. "Mitigating memory-induced dark silicon in many-accelerator architectures." *IEEE Computer Architecture Letters* 14.2 (2015): 136-139.

[14] Lyons, Michael J., et al. "The accelerator store: A shared memory framework for accelerator-based systems." *ACM Transactions on Architecture and Code Optimization (TACO)* 8.4 (2012): 1-22

[15] Diamantopoulos, Dionysios, et al. "High-Level-Synthesis Extensions for Scalable Single-Chip Many-Accelerators on FPGAs." *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015

[16] Xue, Zeping, and David B. Thomas. "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems." *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015

handling the memory's structure and addresses. In many cases, the dynamic memory management systems perform

transparently defragmentation operations for optimizing the heap utilization, avoiding allocation failures and use garbage collection mechanisms. Moreover, typical memory management methods utilize hash tables and use proxy memories structures to perform efficiently the aforementioned operations.

3.4.4 Dynamic Memory Management in High-Level Synthesis

In HLS, BRAMs can be grouped together creating a pool of memory structures (heaps) that are shared among multiple accelerators. The heaps are synthesized and implemented on the FPGA alongside the accelerators, meaning that at runtime each accelerator can allocate blocks of the shared memory space in order to store elements in BRAMs. After the accelerator's completion, the memory that was allocated from the heaps can be de-allocated, allowing other accelerators to use the same BRAMs. The concept of having a shared memory space of a predefined size which can be used by multiple accelerators minimizes the BRAM resources that are required for implementing many accelerators on a single FPGA.

Each heap is composed of two components; ***i)*** a first-fit memory allocator that is used for allocating words from the RAM structure that is implemented with BRAMs. ***ii)*** a freelist bitmap array that is used for storing the allocation status of each byte. The size of each heap and the length (*L*) of the heap's words are parameters specified by the designer during the designing phase.

The freelist bitmap array consists of a group of 1-bit wide registers. Each register is one-to-one mapped with a heap byte, indicating whether this byte is occupied or free. When the content of a register is '1' then the corresponding byte is considered allocated, and hence cannot be allocated by an accelerator. Otherwise, it is unallocated and available for allocation. The Figure[17] 29 below shows an overview of the described components.

[17] Diamantopoulos, Dionysios, et al. "High-Level-Synthesis Extensions for Scalable Single-Chip Many-Accelerators on FPGAs." *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015

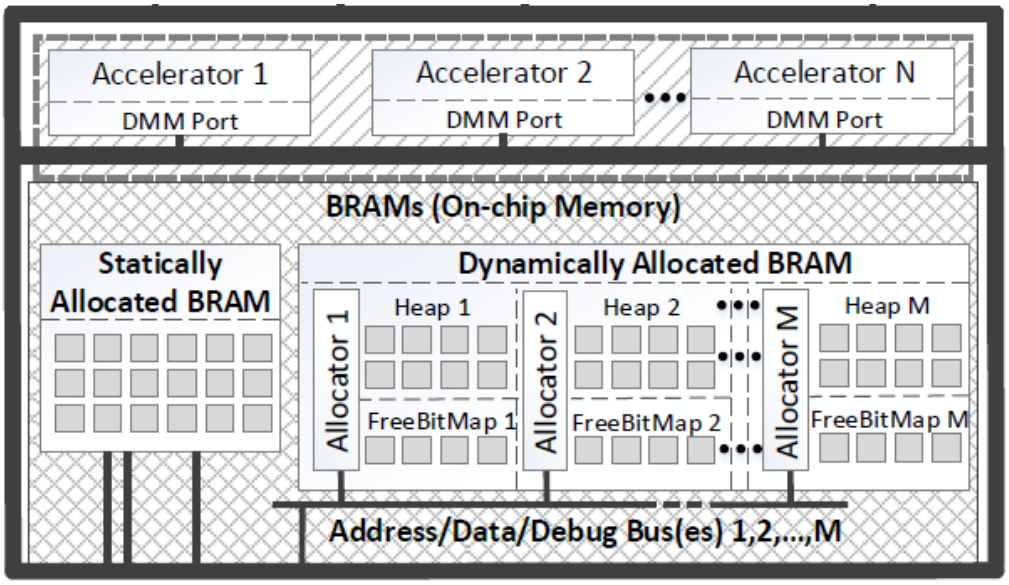

Figure 29: Dynamic Memory Management Architecture

When an accelerator performs an allocation request for N bytes from a specific heap, then the first-fit allocator scans the requested heap's freelist array until it finds the first N consecutive free bytes. Once an unallocated memory region has been found then the content of the freelist registers that are mapped to the found memory block is set to '1', changing the status of those bytes to occupied. It is noted that the granularity of an allocation request cannot be smaller than the length of a heap word (*L*). In case that an accelerator requests fewer than *L* bytes, then a number of extra bytes are allocated as well in order to meet the minimum allocation granularity which is *L*. The Figure 30 example depicts the process of allocating 9 bytes from a heap with *L = 4* bytes. The designed accelerators communicate through the DMM Port (API[18]) with the described mechanism and perform requests for memory allocation and de-allocation targeting a selected heap.

[18] See section 3.5.6

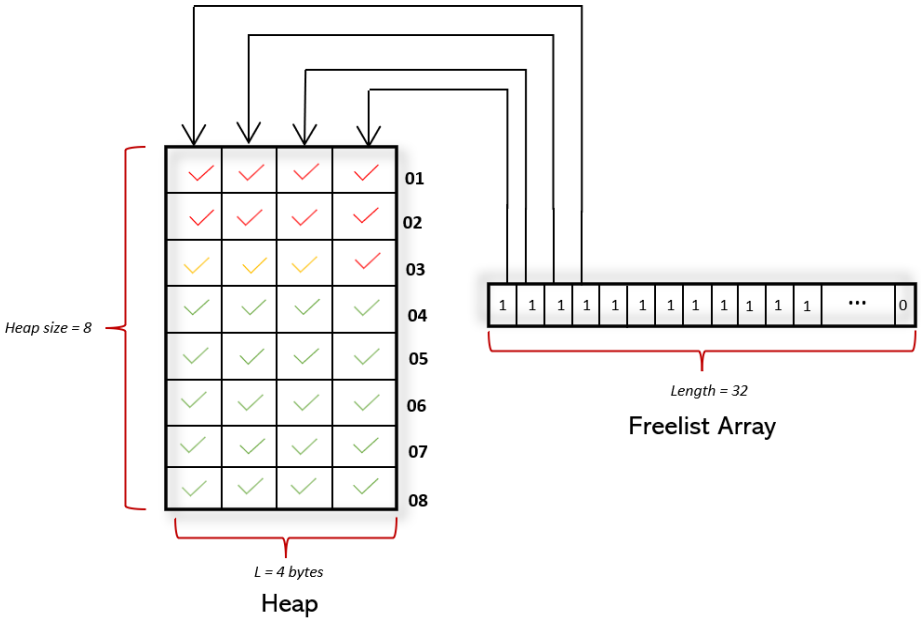

Figure 30: Process of Allocating 9 bytes from a heap

Similarly, when a de-allocation request is performed, the corresponding registers in the freelist array are reset to '0', allowing other accelerators to allocate those bytes.

3.4.5 Dynamic Heap Management in High-Level Synthesis

A mechanism that operates at top of the DMM framework and performs transparently memory management operations on the instantiated heap structures in order to optimize the utilized memory modules, minimize the probability of allocation failure and hence increase the number of implemented accelerators was developed. The purpose of the **Dynamic Heap Management (DHM)** is to eliminate the allocation requests that fail due to heap fragmentation or due to unbalanced heap utilization when multiple accelerators that allocate dynamically memory resources are executed concurrently or in pipeline and therefore perform asynchronous requests for memory allocation and de-allocation on the same or in multiple heap structures. The memory fragmentation issue appears on many-accelerator systems with heterogeneous memory size allocations while asymmetrical heap usage appears when multiple memory allocation requests appear unevenly on the available heaps. Figure 31 below depicts an execution scenario that would lead to unoptimized heap utilization and require methods for optimizing the memory. In the Figure 31 example, three accelerators that are executed concurrently share two heap structures which can lead to unoptimized memory utilization.

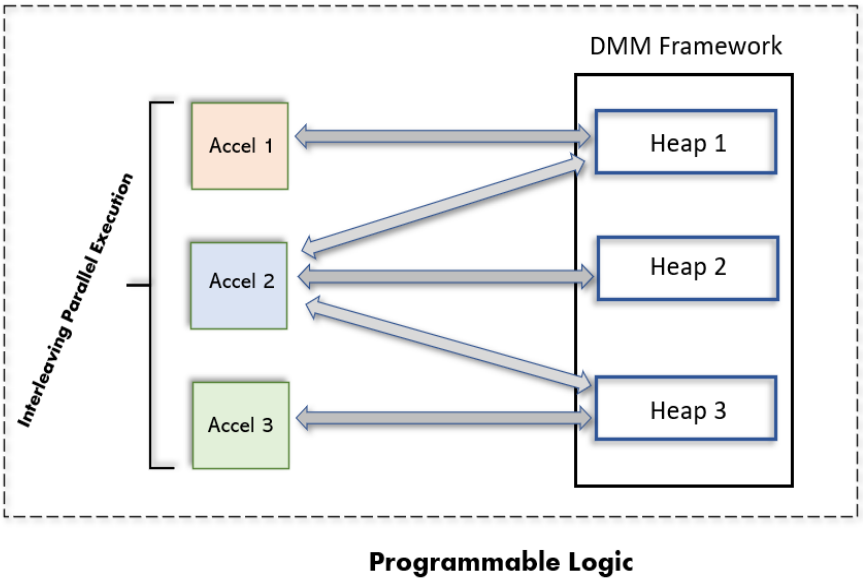

Figure 31: Parallel execution scenario

To minimize the negative impact on the accelerators' performance that is induced from the shared memory space that doesn't allow parallel memory read and write operations due to the BRAMs limited memory ports, ideally a single heap structure would be assigned per accelerator. Although, this allocation scheme would increase the application's throughput and eliminate the need for a DHM mechanism, it has two major defects. ***(i)*** it decreases[19] the number of implemented accelerators. ***(ii)*** requires from the designer to perform static analysis of the accelerator's memory requirements and calculate the required memory sizes for every heap structure. In case that an accelerator performs an allocation request that cannot be served by its assigned heap due to limited available free space, then the allocation request would fail and an application deadlock would occur. For those reasons, DHM is used sacrificing application's performance for accelerators' scalability, improving the robustness of the framework.

In brief, the DHM mechanism performs two operations at runtime:

1. (***Heap Allocation***) When a request for memory allocation is performed by an accelerator. The heap allocator checks whether there are enough available unoccupied memory blocks on the selected heap to serve the current request. In case that the user-defined heap does not have enough free memory then the rest of the heaps are checked until the first heap that can potentially serve this allocation request is found.

2. (***Heap Defragmentation***) When a de-allocation request is performed by an accelerator, the percentage of the fragmented heap is monitored. Once the fragmentation percentage surpasses a threshold, then a compaction mechanism that performs defragmentation by moving the free memory blocks of the corresponding heap is invoked. The heap fragmentation threshold that leads to the execution of the defragmentation mechanism is set by the user during the designing phase.

[19] Diamantopoulos, Dionysios, et al. "Mitigating memory-induced dark silicon in many-accelerator architectures." *IEEE Computer Architecture Letters* 14.2 (2015): 136-139.

Furthermore, the defragmentation mechanism performs two distinct compaction algorithms, depending on the designer's preference, ***(i)*** a complete compaction that nullifies fragmentation, ***(ii)*** a partial compaction that reduces fragmentation.

The Figure 32 below shows an overview of the implemented setup.

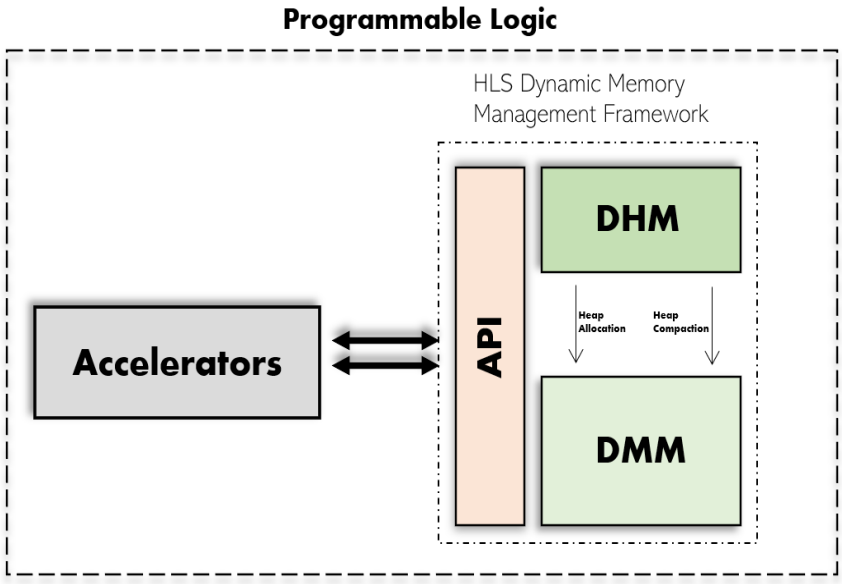

Figure 32: Overview of the designed setup

Heap Allocation. The heap allocation operation is executed transparently when an accelerator performs a memory allocation request. This operation is performed at the first stage of the allocation process, checking if the unoccupied memory space of the requested heap is sufficient for the specific allocation. In case that the requested heap doesn't have enough free space available then the remaining instantiated heaps are checked in a cyclic manner until the first heap that can serve the current request is found. If none of the heaps have enough bytes available, then a malloc fail value is returned to the accelerator. The Figure 33 illustrates an example case of the described operation. In the scenario depicted on Figure 33, an accelerator performs a memory allocation request at heap 1. DHM finds that heap 1 doesn't have enough available space, similarly for heap 2. Heap 3 has available space and hence it is selected to serve the current allocation request.

Heap Defragmentation. The defragmentation mechanism operates by performing memory compaction on the heap that exceeds a user-defined fragmentation threshold. The compaction is performed at runtime by relocating the allocated memory blocks at the bottom of the heap and the fragmented regions at the top. At the same time the free list bitmap array that indicates the heap's memory positions that are occupied is updated correspondingly. The runtime relocation of memory objects creates two issues ***(i)*** the addresses of the dynamically allocated objects for every accelerator should be updated after the compaction. ***(ii)*** accelerators should not be allowed to perform memory read or write operations on the specific heap during the compaction process.

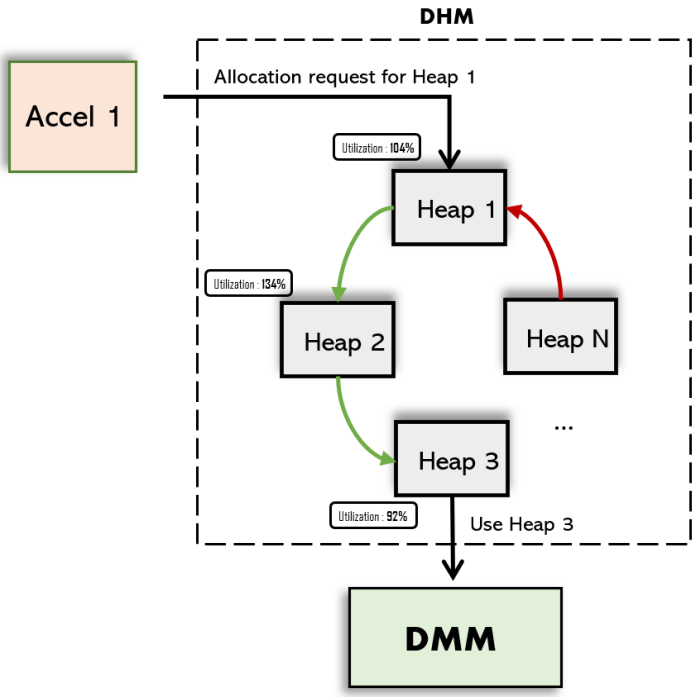

Figure 33: Heap Selection

Typically, in software engineering, applications that require dynamic allocation use addresses from an abstract memory space that are mapped to the physical memory. Hence, when memory management operations are performed the addresses in the abstract memory space remain unchanged and the objects in the physical memory are relocated. However, in HLS such technique would require a pointer-to-pointer arithmetic which leads to non-synthesizable designs. For that reason, the following design is implemented on each memory heap:

A memory structure, called offset table, that is implemented in LUTs is instantiated for every heap. Every object in this table is mapped one-to-one with a heap word, showing the number of positions that this word has been moved within the same heap after the compaction. Through this table the updated addresses of the accelerators' dynamically allocated structures are calculated by adding the content of the offset table to the initially given memory address. The Figure 34 shows an example scenario that demonstrates the purpose of the offset table.

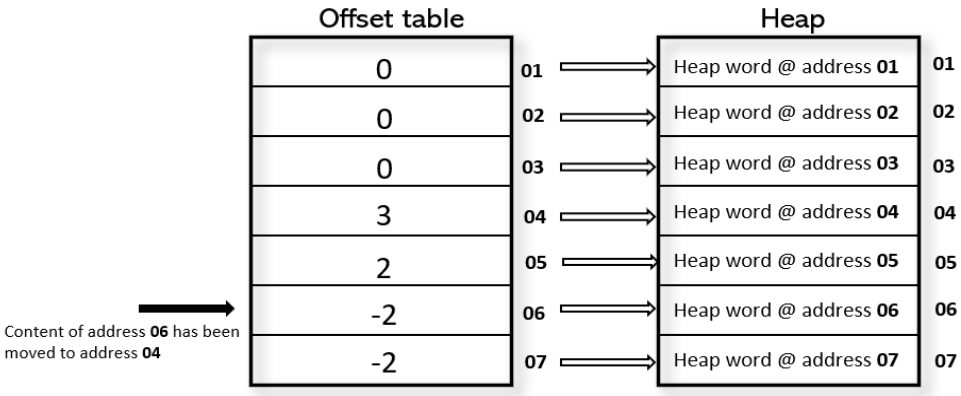

Figure 34: Purpose of offset table

Besides the aforementioned mechanism, each heap contains a 1-bit register called *compactionFlag.* The content of this register is set high when a compaction algorithm is executed on the corresponding heap, blocking the accelerators from memory access operations. The flag is reset when all the accelerators have updated the addresses of their dynamically allocated objects, and so they can access the heap without inducing memory errors.

Currently, two compaction algorithms have been developed. The compaction algorithm that will be applied on each heap is set by the designer during the accelerators' designing phase.

Compaction Algorithm 1: The purpose of the 1st compaction algorithm is to completely defrag the corresponding heap by relocating the fragmented memory regions at the top memory addresses. This algorithmic task is executed when an accelerator performs a memory de-allocation request, and this operation rises the overall heap's fragmentation percentage over a user-specified threshold. To find the fragmented memory regions, a linear search is performed, starting from the top of the heap, and checking the bit status of the free list bitmap array for every heap word. When an occupied word is found, the position of this memory block is stored, meaning that every non-occupied word that is located in lower memory addresses is fragmented. Once a fragmented region is found, then its content is shifted towards the top of the heap. The shifting process is performed by swapping the contents of adjacent memory words at an iterative process that halts when the top address is reached. At the same time the offset table is updated, monitoring the number of positions that every word has been moved within the heap. Similarly, the bits of the free list bitmap array are flipped in order to update the allocation status of the swapped words. In the worst-case scenario all the memory words except the one located in the top address are fragmented. In this case $n^2 + (n-1)^2 + (n-2)^2 + \cdots + (n-(n-1))^2$ iterations are performed for the algorithm's completion, where *n* is the size of the heap, hence the algorithm's complexity is $O(n^3)$. The Figure 35 below illustrates an example of the described compaction algorithm.

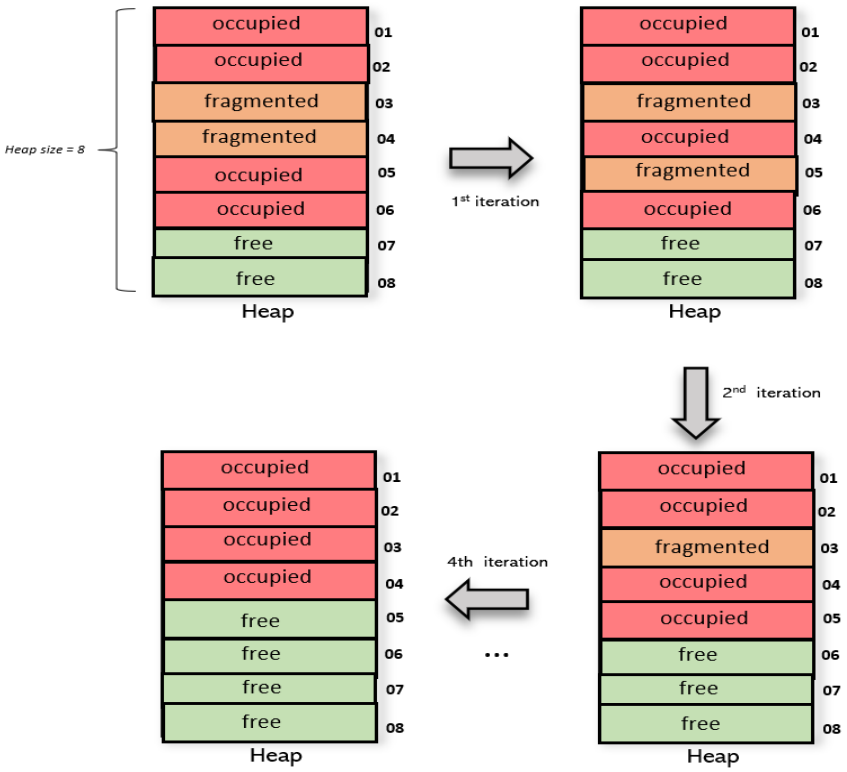

Figure 35: Compaction algorithm

Compaction Algorithm 2: The 2nd compaction algorithm similar to the 1st one, is invoked when the fragmentation percentage exceeds the pre-defined threshold. The purpose of this algorithm is to find the largest fragmented block and move only this region at the top of the heap. At first, a linear search is performed to find the position and the size of the largest "hole", then the found fragmented region is shifted towards the top of the heap while the offset table and the free list bitmap array are updated correspondingly. The complexity of the proposed 2nd compaction algorithm is $O(n)$.

3.4.6 Dynamic Memory Management API

The dynamic memory and heap management operations described in the sections above consist of a set of functions that are used from the designer during the accelerators' designing phase. Specifically, there are four functions that can be used from the accelerator's side:

- *void* MemAlloc(uint_t **bytes**, uint_t **heapID**, uint_t* **AllocHeapID**)*
- *void MemFree(void* **AllocAddress**, uint_t **bytes**, uint_t **AllocHeapID**)*
- *void* Update(void* **AllocAddress**, uint_t **bytes**, uint_t **AllocHeapID**)*
- *uint8_t CheckFlag(uint8_t **AllocHeapID**)*

1) ***MemAlloc***: It is used for performing allocation requests. The accelerator should specify the number of bytes that are required (***bytes***), the ID of the heap structure that would ideally serve the allocation request (***heapID)*** and a pointer that stores the ID of the

heap that will be used for memory allocation (***AllocHeapID***). In case that the user-requested heap has sufficient free memory space then the values of the ***heapID*** and ***AllocHeapID*** parameters are identical. The base address of the newly allocated region is returned to the accelerator.

2) ***MemFree***: It is used for de-allocating memory space from a specific heap. The accelerator should specify the base address of the allocated region (***AllocAddress***), the number of the allocated bytes that will be de-allocated (***bytes***) and the ID of the heap from which the specified memory region will be freed (***AllocHeapID***).

3) ***Update***: It is used for updating the addresses of the dynamically allocated objects in case that the defragmentation task was performed. The accelerator should specify the address of the allocated object (***AllocAddress***), the number of the dynamically allocated bytes (***bytes***) and the ID of the heap structure that was used for the allocation (***AllocHeapID***). At the backend, the ***Update*** function accesses the offset table of the corresponding heap and calculates the new addresses for the relocated heap words. It returns to the accelerator the base address of the relocated memory region.

4) ***CheckFlag***: It is used for checking whether the defragmentation task is performed and if all the accelerators that share the same heap have updated the addresses of their dynamically allocated objects. The ***CheckFlag*** takes input the ID of a specific heap and returns '1' if the content of the compactionFlag register is 1, otherwise it returns '0'. The content of the compactionFlag register should be monitored by the accelerators before performing memory read and write operations in order to avoid memory access errors. For that reason, a code transformation is required and the ***CheckFlag*** is used in a while-loop wrapper that stalls the accelerators' execution.

The code snippets below depict an example of using the described functions in HLS for utilizing the proposed methods.

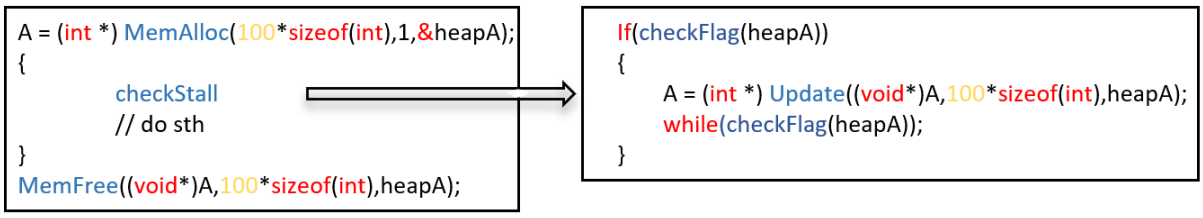

Figure 36: Code Transformation

The mechanisms described in the sections above are synthesized alongside with the accelerators, creating a uniform design that is implemented on the FPGA.

3.4.7 Evaluation

The proposed framework for Dynamic Memory and Heap management was evaluated on three axes. At first the DMM was evaluated for its capability to increase the number of

implemented accelerators on the FPGA. Secondly, the defragmentation mechanism was tested to showcase its capacity to improve the memory usage when multiple accelerators share the same heap. Finally, the experimental analysis of the heap allocator targets to exhibit its potential to minimize allocation failures when memory requests of multiple accelerators are distributed in many heaps.

DMM Evaluation:

The HLS Dynamic Memory Management (DMM) framework was evaluated to determine its performance and its capability to deliver more accelerators per chip area compared to the typical implementation of HLS accelerators when they use static allocation of local memory resources. Specifically, the synthesis results of three different accelerators are evaluated, as depicted on the table 4 below. The measurements that are presented in the following paragraphs were taken on an Alveo U200 data center acceleration card.

Table 4: Tested Accelerators

Histogram Accelerators	Determine frequency of RGB channels in a 640x480 pixels image
2D Matrix Multiplication Accelerators	Dense integer matrix multiplication 100x100
PCA[20] Accelerators	PCA on a matrix 250x250

The bar plots shown on Figure 37 below depict the number of accelerators that can be synthesized on the FPGA for different heap configurations. It can be seen that the highest gain in the accelerators' density is achieved when a single heap configuration is used. Specifically, a x6 increase on the number of implemented histogram accelerators is possible when a single heap structure is used in comparison with the static allocation scheme. In a 16-heap configuration the number of implemented accelerators is the lowest due to the extra memory resources that are required to synthesize each heap. For the matrix multiplication example case, the gain in the accelerators' density is between x2.4 and x2.6 depending on the implemented configuration, while for the PCA accelerators the same range is between x2.8 and x6.2.

[20] Principal Component Analysis

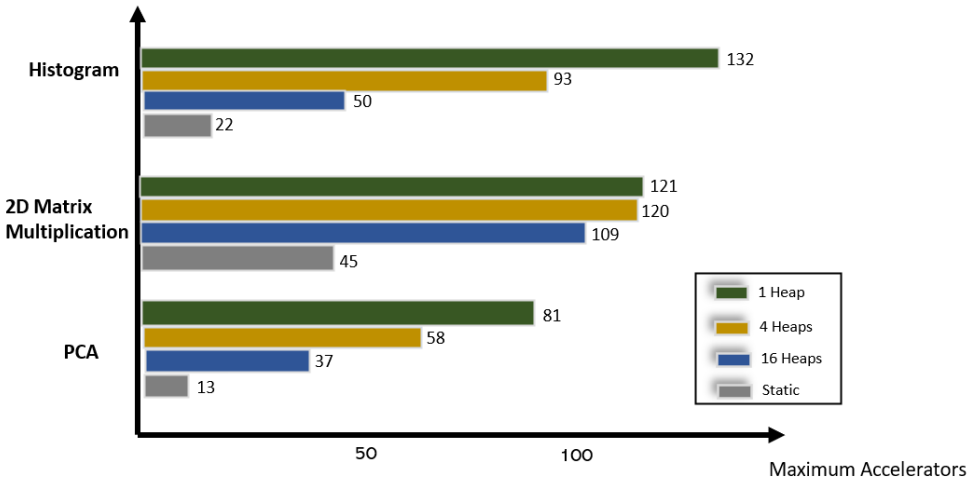

Figure 37: Increase on the accelerator density due to DMM

The Figure 38 depicts the latency of the examined accelerators when 16 accelerators of the three categories are synthesized on the FPGA for different implemented configurations (1 Heap, 4 Heaps, 16 Heaps). The y-axis shows the accelerators' normalized latency with reference the latency obtained from the static memory allocation. When multiple accelerators share one or 4 heaps there is a considerable latency overhead due to memory access conflicts that are caused by the shared memory space and the limited ports of the BRAMs. As the number of the implemented heaps rises then more accelerators can be executed in parallel and therefore the application's latency decreases. In the 16-heap configuration each one of the 16 synthesized accelerators is assigned to a unique heap, allowing all accelerators to be executed in parallel. However, the operations that DMM performs to allocate and de-allocate memory at runtime create a latency overhead that ranges from x1.2 to x2 compared to the static allocation setup.

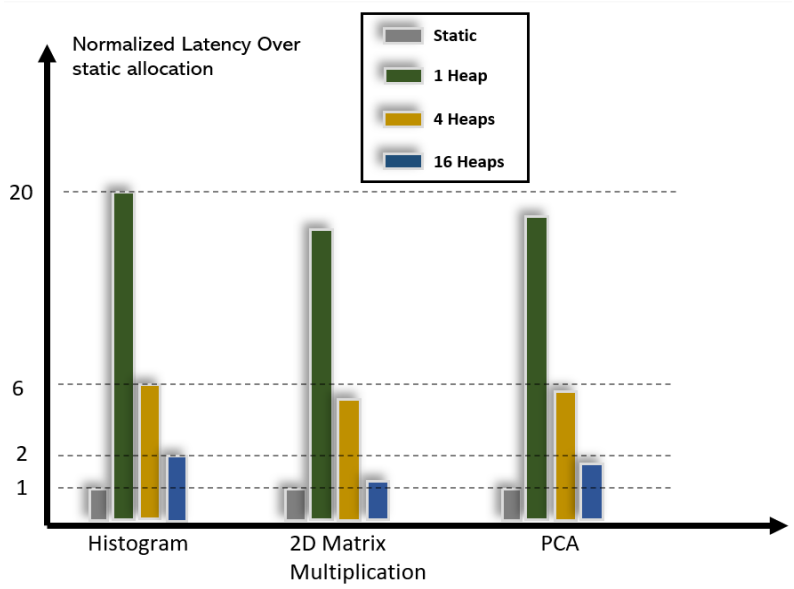

Figure 38: DMM accelerators' latency

Defragmentation Evaluation:

To showcase the effectiveness of the defragmentation mechanism in optimizing memory usage, the proposed methodology was evaluated on three execution scenarios as shown on table 5, where multiple memory allocation and de-allocation requests are performed on the same heap. The evaluation methodology is depicted on Figure 41. Specifically, a Monte-Carlo simulation is used to analyze the probability of allocation failure due to fragmentation when different memory patterns of varied sizes appear. Each memory pattern is transformed to a C++ code, based on which two HLS applications are generated. The first application uses the DMM mechanism without the component that performs defragmentation while the second application has both the DMM and the DHM defragmentation mechanisms activated. In the evaluation flow depicted on Figure 41, an initial memory pattern of requests for dynamic memory allocation and de-allocation that correspond to one of the three testing scenarios is created. Then, the pattern generator pseudo randomly re-assigns the order of the MemAlloc and MemFree commands with the constraint that a MemAlloc request precedes its corresponding MemFree. This process is repeated for 10000 iterations, meaning that 10000 memory patterns are evaluated per scenario. The Figure 39 below, illustrates an example memory pattern produced from the execution of 6 accelerators sharing the same heap. This example application execution is one input in the setup depicted on Figure 41.

Table 5: Evaluation Scenarios

Scenario 1	Memory Alloc / Free requests of sizes ranging from 10% to 40% of the heap size.
Scenario 2	Memory Alloc / Free requests of sizes that correspond to three matrix multiplication accelerators (40 x 40, 50 x 50, 80 x 80)
Scenario 3	Memory Alloc / Free requests of sizes that correspond to five FIR filter accelerators with input signal sizes (2000, 3000, 5000, 7000, 9000)

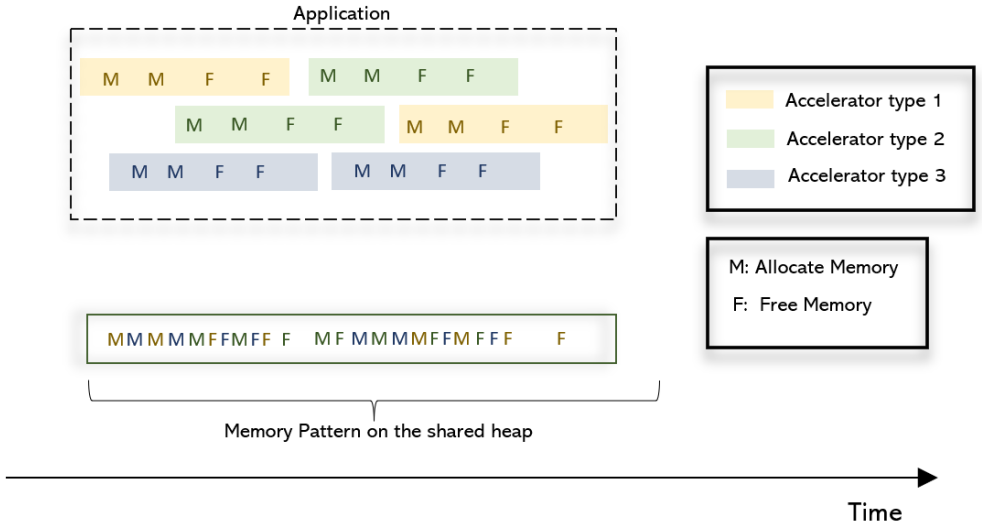

Figure 39: Example Application

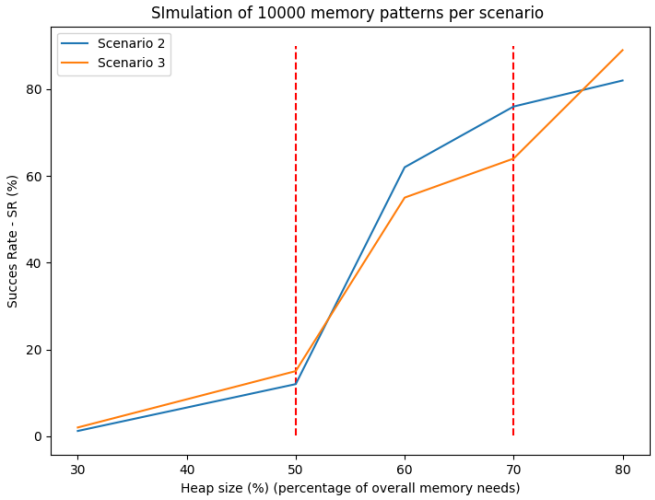

Figure 40: Successfully Completed Tests Vs Heap Size

Based on the Figure 40 below, the number of the successfully completed execution scenarios of the overall 10000 tested cases for both scenarios 2 and 3 plummets below 15% when the heap size is smaller than 50% of the accelerators' memory needs. This percentage reaches values up to 89% when the heap size increases. To exhibit the benefits of the defragmentation mechanism a heap size equal to 60% of the overall memory needs was selected for the evaluation experiments.

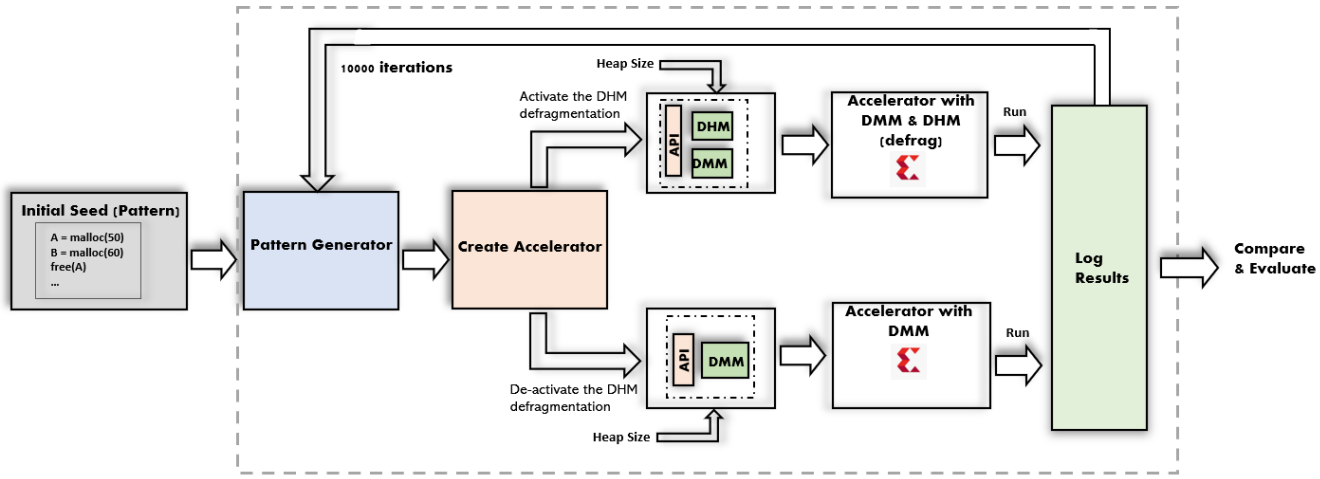

Figure 41: Defragmentation evaluation setup

Figure 42 shows the number of the successfully completed executions for the three evaluation cases. In the 1st scenario where the allocation sizes are normally distributed in a range from 10% to 40% of the heap size, 5211 out of the 10000 generated memory patterns are completed successfully when the defragmentation mechanism is activated. On the contrary, when the DHM defragmentation mechanism is not used, 4625 of the same 1000 patterns don't fail.

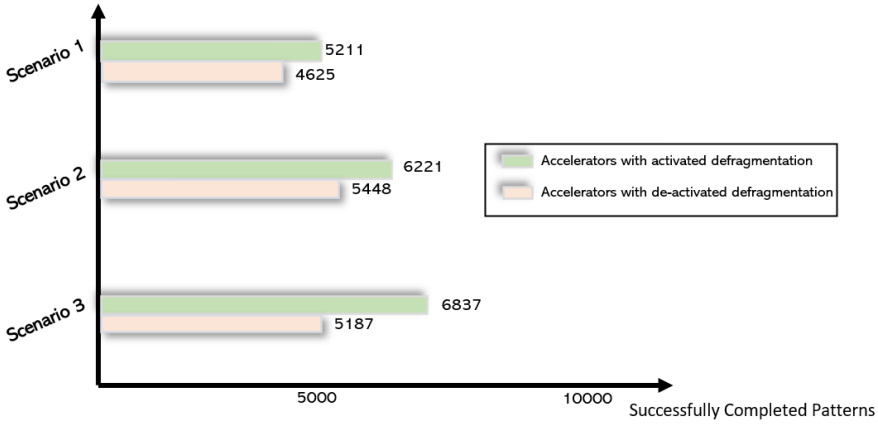

Figure 42: Successfully completed tests in the two execution flows

Similarly, in the 2nd scenario, the memory patterns that correspond to 9 matrix multiplication[21] accelerators are tested. In this case the success rate (**SR**) when the defragmentation mechanism is used is 62.21% while it is 54.4% when the memory compaction algorithm is de-activated, meaning that 7.4% of the total memory allocation failures are due to fragmentation. In the 3rd evaluation scenario, the memory patterns of 10 FIR filter accelerators are evaluated. Specifically, two accelerators from each one of the five versions of the tested FIR filters are instantiated and share the same heap. The difference in the **SR** between the two tested cases is 16.5%.

[21] three accelerators performing matrix multiplication on 40x40 matrices, three on 60x60 matrices and three on 80x80. Overall 9 accelerators.

On the Figure 43, the amount that the heap size can be decreased when the defragmentation mechanism is used is depicted. In the 2nd tested scenario a heap size reduction up to 12.5% is possible without dropping the probability of allocation failure. Similarly in the 3rd evaluation case, the heap size can be decreased up to 18.75% without a significant drop in the success rate.

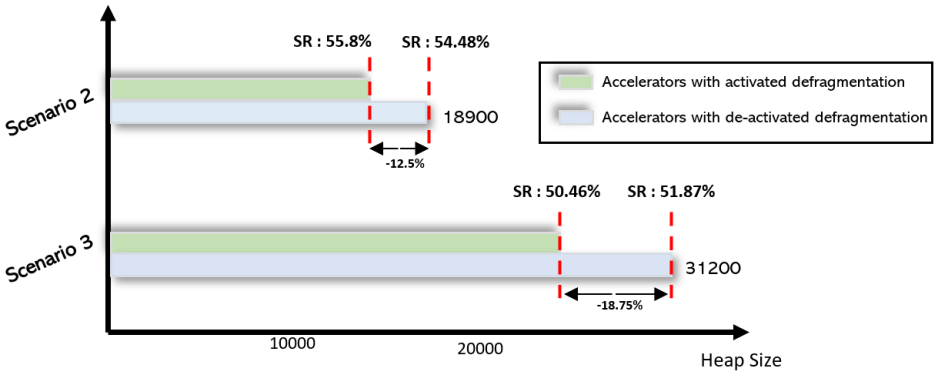

Figure 43: Achieved decrease on the heap size

Finally, the table 6 below shows the latency of the compaction algorithm[22] that was used for the defragmentation process. The presented measurements consider the worst-case latency for 4 different heap sizes. It is noted that the clock frequency for all 4 measurements is 250MHz.

Table 6: Compaction algorithm's worst-case latency

Heap Size	Latency
512	73 us
1024	0.14 ms
8192	1.17 ms
31200	5.47 ms

Heap Allocator Evaluation:

The evaluation scenarios of table 4 and the setup depicted on Figure 44 were used for testing the heap allocation mechanism. In this evaluation flow (Figure 44) an initial memory pattern that corresponds to one of the three scenarios is provided. Then the heap allocation generator pseudo-randomly assigns each allocation request to one of the available heaps and the pattern generator pseudo-randomly re-assigns the order of the memory allocation/de-allocation instructions. A Monte-Carlo simulation is performed for the two generated

[22] The 2nd compaction algorithm with O(n) complexity. The threshold was set to greater than zero.

applications similar to the evaluation process for the defragmentation mechanism. The described simulation is repeated for 10000 inputs.

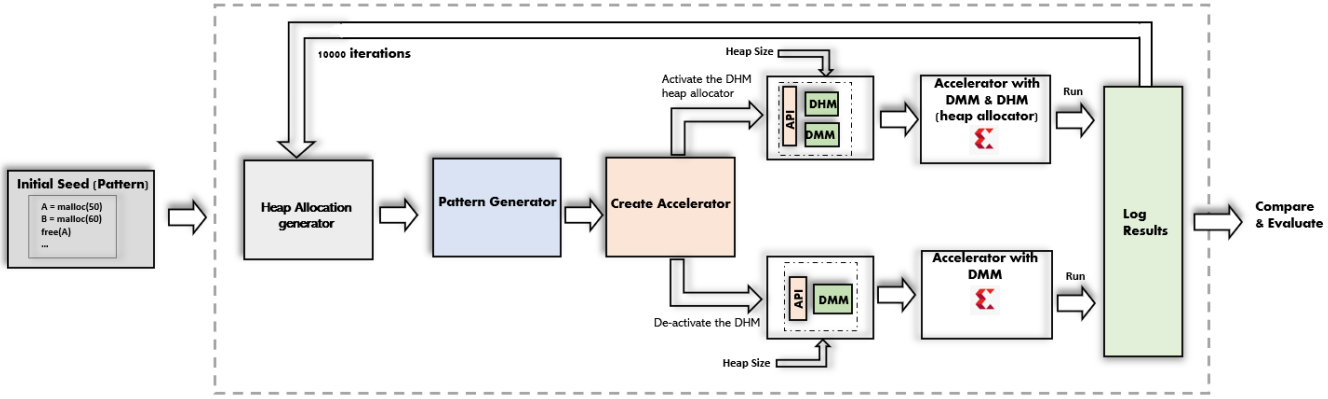

Figure 44: Evaluation setup for heap allocator

The Figures 45 and 46 below show the results of the aforementioned simulation when 2 and 3 heaps are used respectively. A decrease of up to 39% at the memory allocation failures can be achieved when the heap allocator is used, while the lowest gain appears on the 1st scenario with the possibility of allocation failure being decreased by 18%. In the 3-heap configuration the number of the successfully completed simulations rises for all scenarios because the accelerators' memory needs are shared among all 3 instantiated heaps and hence the possibility of allocation failure due to lack of available free memory resources is low.

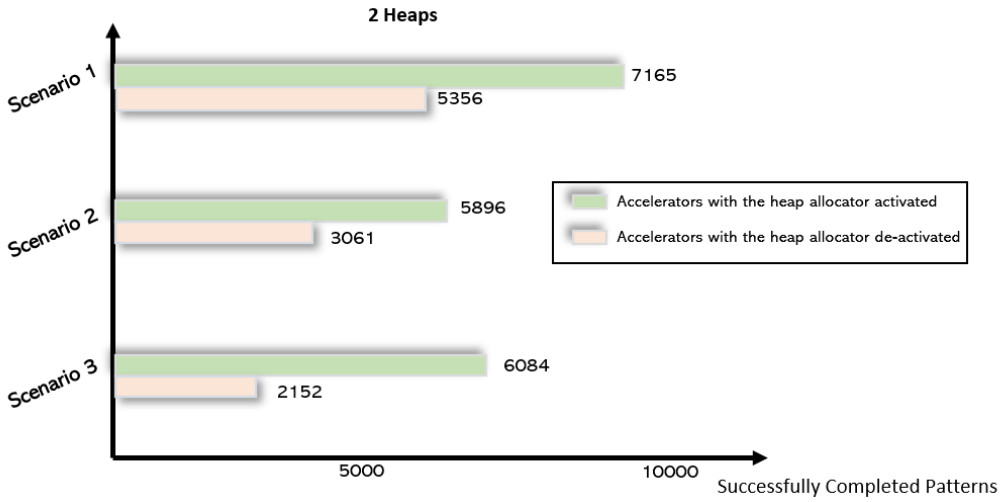

Figure 45: Reduction on allocation failures when 2 heaps are shared

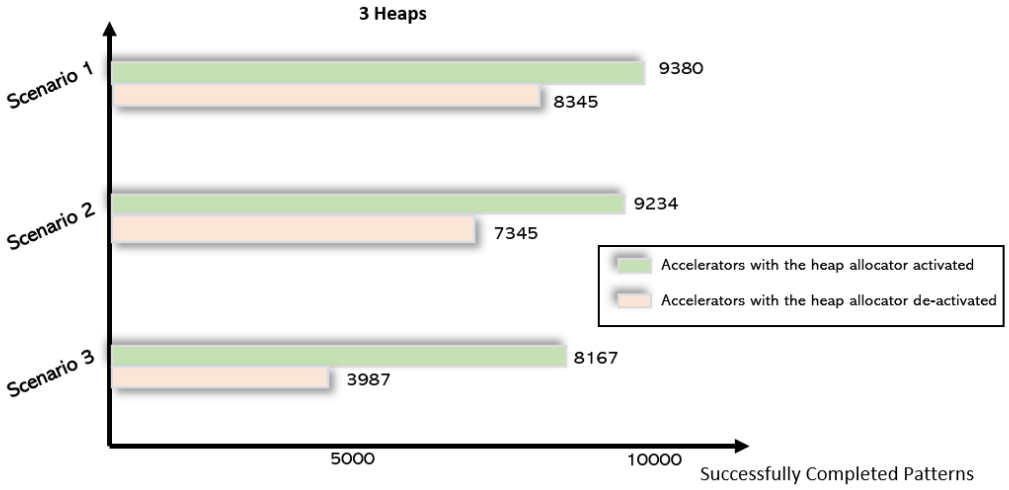

Figure 46: Reduction on allocation failures when 3 heaps are shared

4 Abstractions for HW acceleration of short-lived tasks

4.1 Overview

Hardware acceleration offers a high degree of computational throughput in a very small power envelope for a wide range of application domains. Many applications, e.g. Machine Learning, Computer Vision, HPC, rely on hardware accelerators, such as GPUs, FPGAs, NPUs, etc. to increase the amount of data they can process and at the same time reduce their energy footprint compared to traditional CPU-only systems.

At the same time, we have seen in the past decades 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 organization of the application owner, in order to reduce costs related with the ownership, operation, maintenance and deployment of software.

This paradigm changed dramatically the way in which we develop, package and deploy applications. Migrating application execution to public cloud infrastructures means that our code will run side-by-side with code of other tenants of the platform. In order to tackle issues related mainly with 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, is responsible for monitoring and restricting the user application from accessing resources they do not own. However, neither of those systems are able to control the access to hardware acceleration devices in the same granularity or the same isolation guarantees as they can with other types of 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 important 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 efficiently virtualize them in an abstract and efficient way.

In SERRANO we introduce vAccel, a framework that enables virtualized workloads to access hardware accelerators securely and efficiently. vAccel is addressing this situation in two ways. Firstly, it enables developing 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 the part of the application that actually 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, in order to delegate accelerable code in vAccel agent on the host.

In the following sections, we describe the interface we have designed and implemented to enable workloads to access various hardware accelerators securely and efficiently.

4.2 Hardware acceleration for Serverless

Hardware acceleration virtualization is an open problem. Typical ways to expose hardware accelerator capabilities inside a virtualized guest involve (i) device pass-through, (ii) para-virtualization and (iii) API remoting[23].

Device pass-through allows direct access to the physical device whereas it achieves performance identical to running on bare metal. However, problems arise with sharing a single hardware accelerator across multiple guests executing on the same host.

Another line of research focuses on implementing para-virtual drivers for GPUs[24] [25] or even FPGAs[26]. Para-virtual drivers are already very popular for other devices like NICs and storage devices. Para-virtual devices abstract a wide range of physical devices in a uniform way exposing a single API inside the guest in the form of specialized kernel modules and guest libraries. This requires that (i) we can efficiently abstract and unify a widely-diverse set of hardware and acceleration framework APIs and (ii) the guest components continuously develop to reflect the evolution of these APIs. Both these requirements are difficult to achieve in an efficient and scalable way hindering any effort for hardware acceleration para-virtual interfaces.

Finally, API remoting provides mechanisms to intercept calls to hardware accelerator drivers inside the guest and offload them to a host with direct access to the physical device. A number of frameworks have been developed which provide such primitives. An earlier body of work [27] [28] targeted traditional graphics applications, building on top of frameworks such as OpenGL whereas more recent research[29] [30] implements solutions for General Purpose GPU (GPGPU) applications building on top of CUDA. These solutions are typically hypervisor-agnostic

[23] C.-H. Hong, I. Spence and D. Nikolopoulos, "GPU Virtualization and Scheduling Methods: A Comprehensive Survey.," *ACM Comput. Surv.,* vol. 3, no. 50, p. 37, 2017.

[24] Y. Suzuki, S. Kato, H. Yamada and K. Kono, "GPUvm: GPU Virtualization at the Hypervisor," *IEEE Transactions on Computers,* vol. 65, no. 9, 2016.

[25] D. Vasilas, S. Gerangelos and N. Koziris, "VGVM: Efficient GPU capabilities in virtual machines," in *International Conference on High Performance Computing & Simulation*, 2016.

[26] W. Wang, M. Bolic and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2013.

[27] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan and E. d. Lara, "VMM-independent graphics acceleration," in *3rd international conference on Virtual execution environments*, 2007.

[28] J. Hansen, "Blink: Advanced display multiplexing for virtualized applications.," 2007.

[29] J. Duato, A. J. Pena, F. Silla, R. Mayo and E. S. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," International Conference on High Performance Computing & Simulation, 2010.

[30] L. Shi, H. Chen, J. Sun and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines.," *IEEE Transactions on Computers,* vol. 61, no. 6, 2012.

however they tend to suffer from overhead due to network latencies when sending very fine-grain acceleration requests and at the same time they are hardware-specific, e.g. CUDA.

AvA[31] identifies these shortcomings of current API remoting approaches and suggests a framework for automatically generating hardware-specific and agnostic components through user-provided descriptions of the acceleration API they are consuming. The API remoting communication is mediated through a hypervisor-controlled communication channel. Their approach is not hardware specific and minimizes transport-related overheads, however it relies on the programmer to define the semantics of the communication in a process which requires iterations for defining the remoting API. Moreover, since the communication channel is mediated through the hypervisor the whole stack is hypervisor specific and requires effort to migrate the solution to other systems. Finally, the agent of the remoting stack is API specific, hence tightly coupled with the specific user-application being executed in the VM guest, which is impractical for cases where acceleration is provided as service.

vAccel differentiates itself from other API remoting approaches by clearly defining the acceleration API at the granularity of pre-defined functions it supports. The approach reduces flexibility in terms of what is accelerated with vAccel, however it enables a number of desirable properties:

- The granularity of the request offloading is sufficiently coarse-grained avoiding transport overheads.
- The vAccel stack is hypervisor agnostic and relies on the standard vsock para-virtual interface as its transport layer. All major hypervisors already support vsock, so the vAccel framework can be deployed to a wide variety of systems without extra effort.
- vAccel minimizes the effort required by developers in order to deploy code in the given infrastructure.
- vAccel supports a well-known API, allowing the implementation of the server side agent to be unique and straightforward. It can be deployed in a simple way as a service from the operator of the platform.

4.3 vAccel

vAccel is a framework designed to enable portable and secure hardware acceleration in multi-tenant environments. The core idea of vAccel is decoupling the user application from hardware-specific code. The vAccel runtime exposes to user applications functions that can be accelerated, however the actual hardware-specific code implementing these functions for a particular hardware accelerator device is provided in the form of plugins which are loaded at runtime. As a result, vAccel applications can migrate from one host to another, without need for code modifications, or re-compilation. At the same time, the vAccel modular design eliminates user code running on shared accelerators. Only code included in a vAccel plugin executes on the hardware accelerator, decreasing the effective attack surface.

Moreover, vAccel employs an API remoting approach for exposing hardware acceleration inside virtual machines. vAccel uses a *transport* plugin in order to dispatch user API calls to the

[31] H. Yu, A. M. Peters, A. Akshintala and C. J. Rossbach, "AvA: Accelerated Virtualization of Accelerators," in *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020

host over a VirtIO Sockets (vsock) device. On the host side, a vAccel Agent application listens for acceleration requests, executes them on the available hardware devices and responds with the computation result.

vAccel Runtime

Figure 47 depicts vAccel runtime (vAccelRT), the core component of the vAccel framework. vAccelRT is a thin dynamic library which on the front-end exposes to the application an API consisting of a set of “accelerable??” functions, while at the back-end it manages a set of plugins providing hardware implementations for the vAccel API.

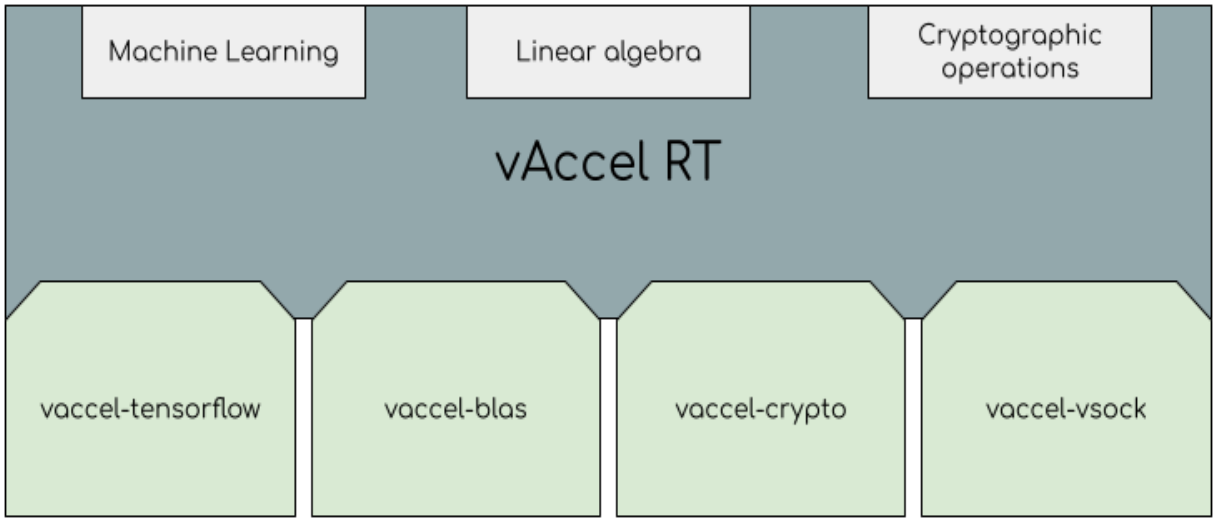

Figure 47: vAccel Runtime System

In vAccel the hardware accelerator specific implementation of the API lives inside vAccel plugins. A vAccel plugin is a dynamic library which implements a subset of the front-end API for a particular hardware accelerator. When a vAccel application is being started, vAccelRT loads the available plugins on the platform inside the address space of the application. Subsequently, it is responsible for dispatching vAccel API calls made by the user application to the corresponding implementation of one of the available vAccel plugins.

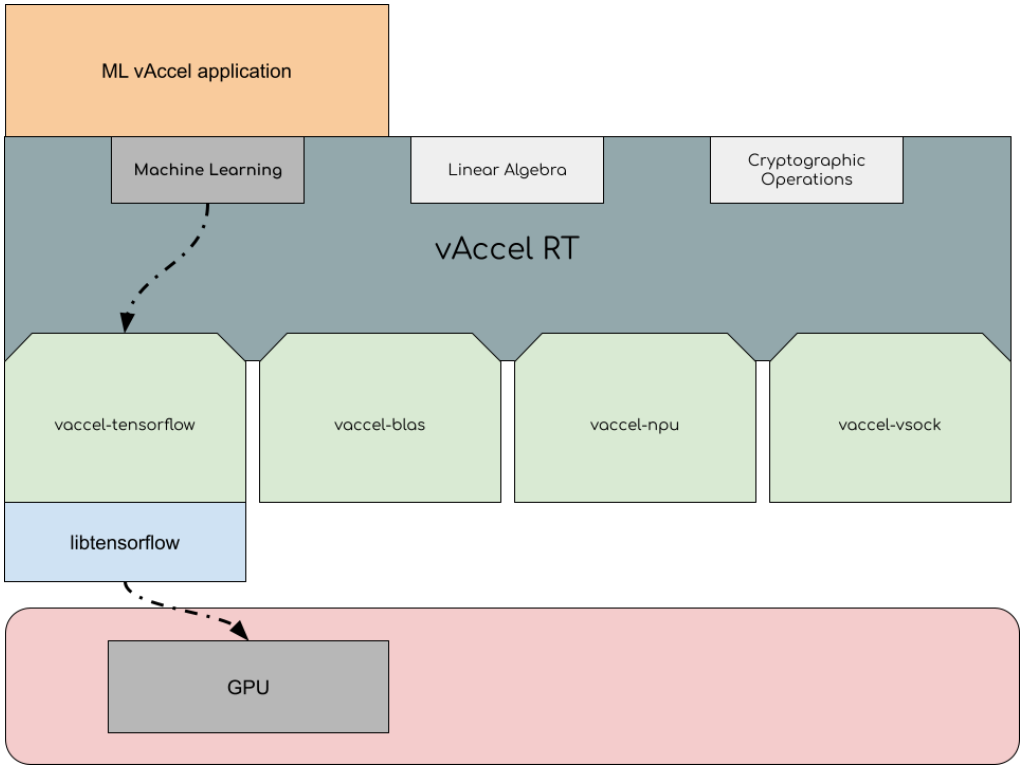

Figure 48: A vAccel application using the ML API of vAccel. vAccelRT dispatches the ML API calls to the TensorFlow plugin which performs the operation on GPU

For example, Figure 48 depicts a vAccel application that consumes the Machine Learning API of vAccel. When a call to the vAccel API is made by the application, vAccelRT searches for any of the available plugins that implement the relevant call. If one is found, the call is being dispatched to it (in this case a TensorFlow-based plugin) which, in turn, executes the operation on the corresponding hardware accelerator device.

Separating user- from hardware-specific code is the key concept of vAccel design that renders vAccel portable across hosts with varying hardware accelerator devices. It is the vAccel plugins that provide the hardware implementation of vAccel functions. Consequently, migrating a vAccel application from one host to another is straightforward; we only need vAccel plugins implementing the required subset of the vAccel API on each one of these hosts.

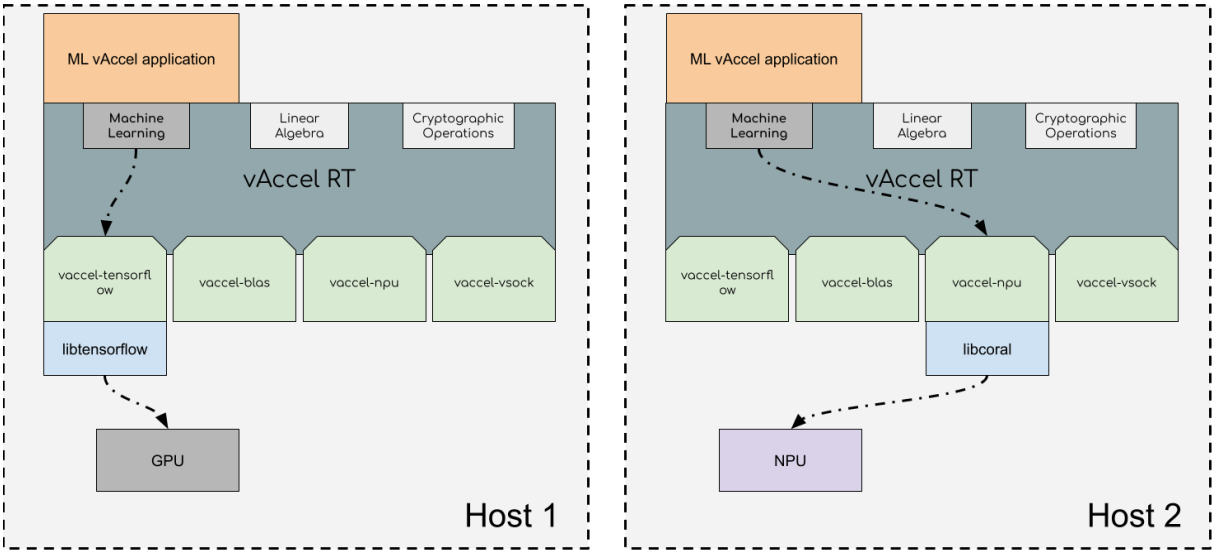

Figure 49: The same ML vAccel application deployed on two hosts, one featuring a GPU and the one an NPU. Execution on each one of the hosts is supported on suitable vAccel plugins

Figure 49 represents this concept. We are deploying the same ML vAccel application on two different hosts. Host 1 features a GPU whereas Host 2 an NPU device. In the first case, vAccelRT is dispatching the ML API calls to the TensorFlow plugin which uses libtensorflow to perform the operations on GPU. On the other hand, on Host 2 where we do not have a GPU, vAccelRT is redirecting API calls to the vaccel-npu plugin which uses libcoral to perform the operations on the underlying hardware. In both cases, the vAccel application and vAccelRT are the exact same binaries. The differentiation lies

Virtualization

The vAccel framework exposes hardware acceleration inside virtual machines in a hypervisor-agnostic fashion by employing an API remoting approach. The natural granularity for vAccel to intercept code and offload it to the host is that of vAccel API calls. This differentiates vAccel from other API remoting frameworks such as rCUDA which on one hand operate at the level of the accelerator hardware driver, i.e. at a finer-grain level (which renders it more difficult to hide latency overhead) and on the other hand they are hardware-device specific, e.g. NVIDIA.

Vhost communication channel

The vAccel API remoting transport channel for an application executing inside a VM is *vsock*. vsock is a paravirtual interface for socket-like communication between a guest VM and its host operating system. On the guest side the channel is exposed as a para-virtual device under */dev/vsock*, whereas on the host side, the hypervisor exposes a UNIX socket. A vsock address consists of two parts, a 32 bit Context Identifier (CID) address and a 32 bit port number where ports below 1024 are privileged. Well known CID and port values are shown in Table 7.

Table 7: Context Identifiers and port values for vsock addresses

Name	Value	Description
VMADDR_CID_ANY	-1U	Any address for binding
VMADDR_CID_HYPERVISOR	0	Reserved for services built inside the hypervisor
VMADDR_CID_LOCAL	1	Local communication (loopback)
VMADDR_CID_HOST	2	Address of the host
VMADDR_PORT_ANY	-1U	Any port number for binding

gRPC

vAccel builds the serialization of acceleration requests on top of the gRPC framework. gRPC is a remote procedure call framework designed to enable efficient services communication across data centres or within microservices. It uses HTTP/2 at its transport layer and Protocol Buffers (protobuf) as its service description language while it provides features for authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, cancellation and timeouts as well as telemetry. Finally, it allows generating bindings for various high-level programming languages and tools for creating boiler-plate code both for client and server implementations.

vAccel API has been translated into protobuf definitions and client and server implementations have been created based on these definitions. Currently, we support the synchronous flavour of gRPC API services while we are evaluating the potential performance benefits of adopting the asynchronous semantics as well.

Client (guest) implementation

The client side of the vAccel API remoting mechanism is built on top of the client definition of the vAccel gRPC service. The core functionality is built within a vAccel plugin which implements the whole of the vAccel API. Its responsibility is to serialize vAccel API calls in gRPC calls and send them over the vsock-end of the communication channel.

Figure 50 shows this succession of events: (i) a vAccel application calls one of the vAccel supported functions, (ii) vAccelRT dispatches the call to the vsock plugin, (iii) the vsock plugin translates the function call into a gRPC service request and finally (iv) the serialized message is transmitted over the vsock socket to the host.

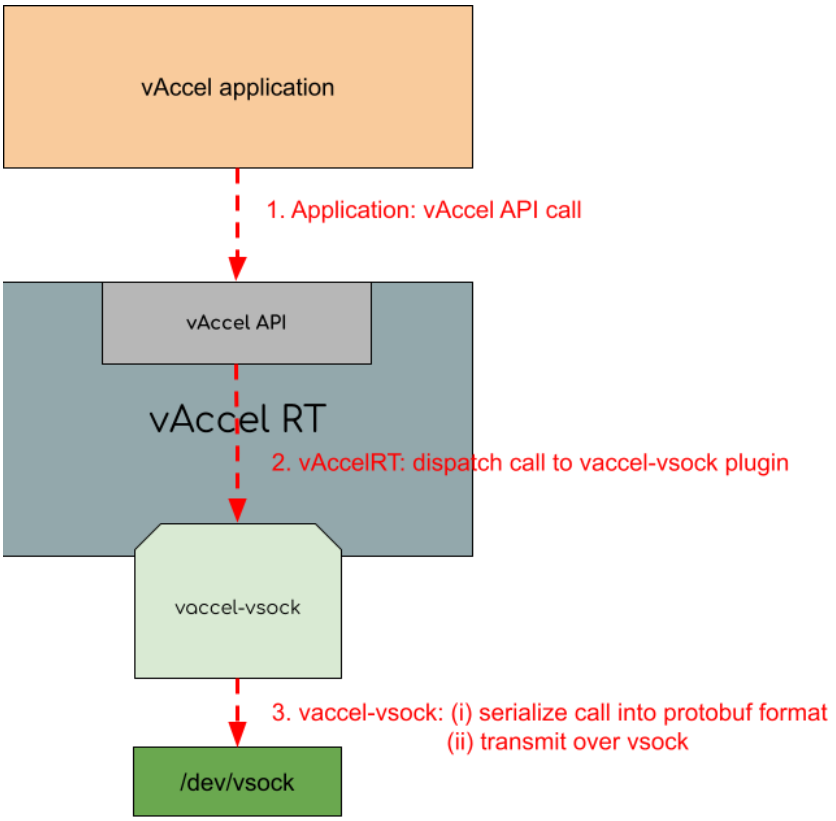

Figure 50: Steps into offloading a vAccel call over a vsock connection

Server (host) implementation

In order to handle acceleration requests originating from inside a guest VM, vAccel introduces a vAccel *Agent* application. The Agent is, essentially, a server binding on the UNIX end of the vsock channel which the hypervisor exposes as a named socket on the host file-system. It is implemented on top of the server definition of the vAccel gRPC service and it is essentially a vAccel application, meaning that it links against vAccelRT.

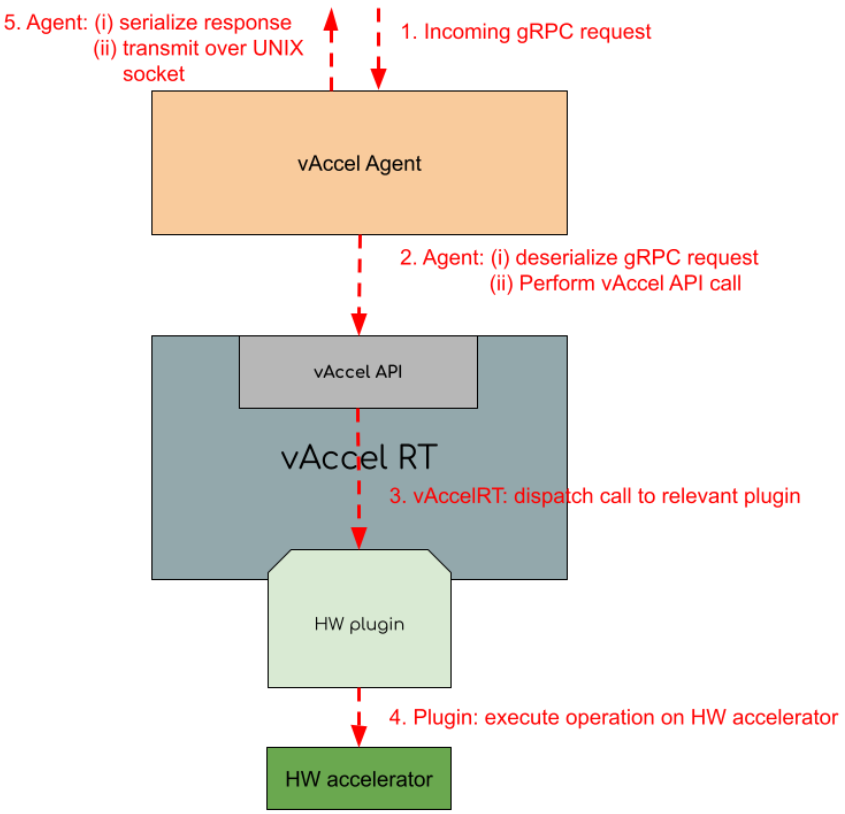

Figure 51: Steps taken by the vAccel agent while handling an acceleration request from a guest application

Figure 51 represents the series of steps taken by the Agent when handling an acceleration request from a guest VM: (i) an acceleration request arrives over the UNIX socket, (ii) the Agent deserializes this request, retrieves the type of acceleration function called from the guest application with its arguments and it calls vAccelRT, (iii) vAccelRT looks for a proper plugin that can handle the type of acceleration requests and dispatches the call and (iv) the hardware plugin performs the operation on the hardware accelerator device. Finally, the Agent serializes the response in gRPC format and transmits it over the UNIX socket.

4.4 Serverless Framework

In SERRANO, we build on OpenFaaS 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.

In this section we briefly describe the functionality of OpenFaaS and how we enhance it to support secure hardware acceleration for functions.

An OpenFaaS deployment consists of the following components:

OpenFaas Gateway: This is the only user-facing component of OpenFaaS. Users can interact with the Gateway via the faas-cli, the provided UI or HTTP requests. The API Gateway provides

an external route into the deployed functions and collects metrics through Prometheus. The built-in UI can be used to deploy and invoke functions. The gateway scales functions according to demand by altering the service replica count in the Kubernetes API. It is also responsible for receiving custom alerts generated by AlertManager on the /system/alert endpoint.

OpenFaaS Provider: The faas-provider provides a CRUD (Create, Read, Update, Delete) API to manage functions, the API to invoke functions via a proxy and the auto-scaling of functions. Optionally, it can also provide the CRUD API for secrets and log streaming capabilities. It is an abstract interface that can be implemented in various ways. Essentially, it provides a compatible interface for the OpenFaaS Gateway to interact with the underlying infrastructure. The most prominent providers are faas-netes, suitable for a Kubernetes cluster and faasd, suitable for a single node deployment.

OpenFaas Function Watchdog: The function watchdog is a lightweight HTTP server with the knowledge on how to execute the actual function's business logic. It is the init process of the function's container and exposes the function via HTTP requests. It also provides Prometheus instrumentation, a health check mechanism, rate limiting and metrics. There are two different watchdog implementations available, the classic watchdog and the of-watchdog. Additional details about these watchdog types is provided below.

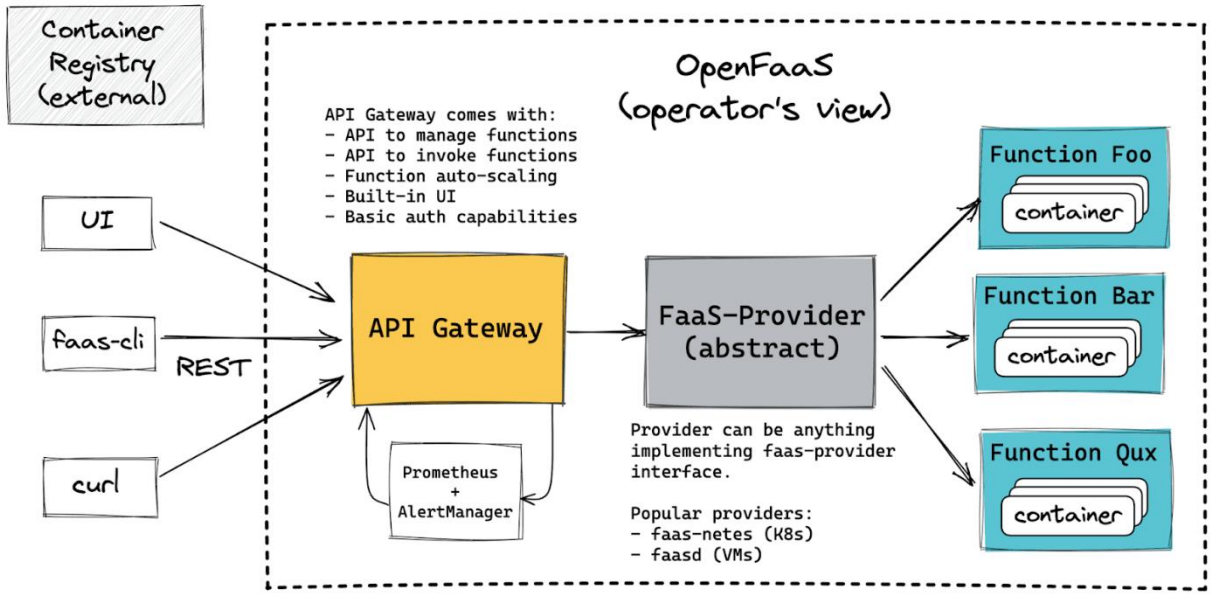

Figure 52: A logical diagram of the OpenFaas architecture

Typically, OpenFaaS is deployed on high-level orchestrators such as Kubernetes using the faas-netes Provider. Installation is usually done using a Helm chart, which bundles all the necessary components that need to be deployed.

OpenFaaS, by default, uses two separate Kubernetes namespaces. The openfaas namespace is used by the components that comprise the OpenFaaS platform (the control plane), while openfaas-fn is used by the deployed functions.

The Pods required to drive a serverless function execution using OpenFaaS are the following:

- Alertmanager
- Basic Auth Plugin
- Gateway
- NATS
- Prometheus
- Queue worker

When deployed on Kubernetes using the faas-netes provider, OpenFaaS makes use of the primitives that Kubernetes provides, both for the platform components and the deployed functions. All requests via faas-cli, UI or HTTP requests are routed by the Gateway service to the Gateway Pod(s). Similarly, all function invocations are routed to the appropriate Pods and handled by the watchdog.

Inside the container, the watchdog starts up an HTTP server listening on port 8080 for incoming requests (function invocations). There are two different implementations of the watchdog that provide different functionality depending on the functions' needs.

The **classic watchdog** reads the request's headers, body and method, spawns a new process that contains the actual function, writes the body to the stdin of the function process and makes the headers and the body available to the process as environment variables. It then waits until the spawned process exits or times out, reads the process' stdout and stderr and sends the read bytes back to the caller in an HTTP response. This mode is very useful, because it offloads the need to implement an HTTP server from users. It also allows any program that uses stdio streams for the I/O handling to be deployed as an OpenFaaS function.

The functionality of the classic watchdog is depicted in Figure 53.

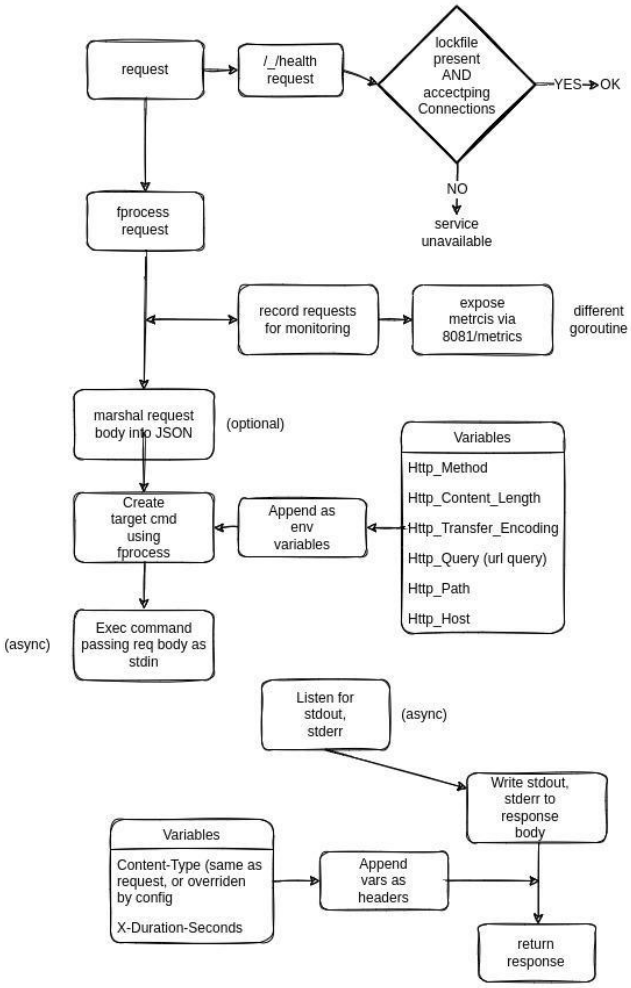

Figure 53: Fwatchdog logical diagram

The **of-watchdog** uses a totally different approach. The watchdog starts up an HTTP server listening on port 8080, but only spawns the function's process once, expecting it to become a long-running HTTP server that sits behind the watchdog. When the function gets invoked, the watchdog sends an HTTP request to the function's server and serves the response back to the caller. In other words, the of-watchdog acts as a Reverse Proxy for the actual function server.

This mode is useful when spawning a new process for each invocation is not ideal due to performance issues. It can also be used to deploy more complex services, stateful functions etc.

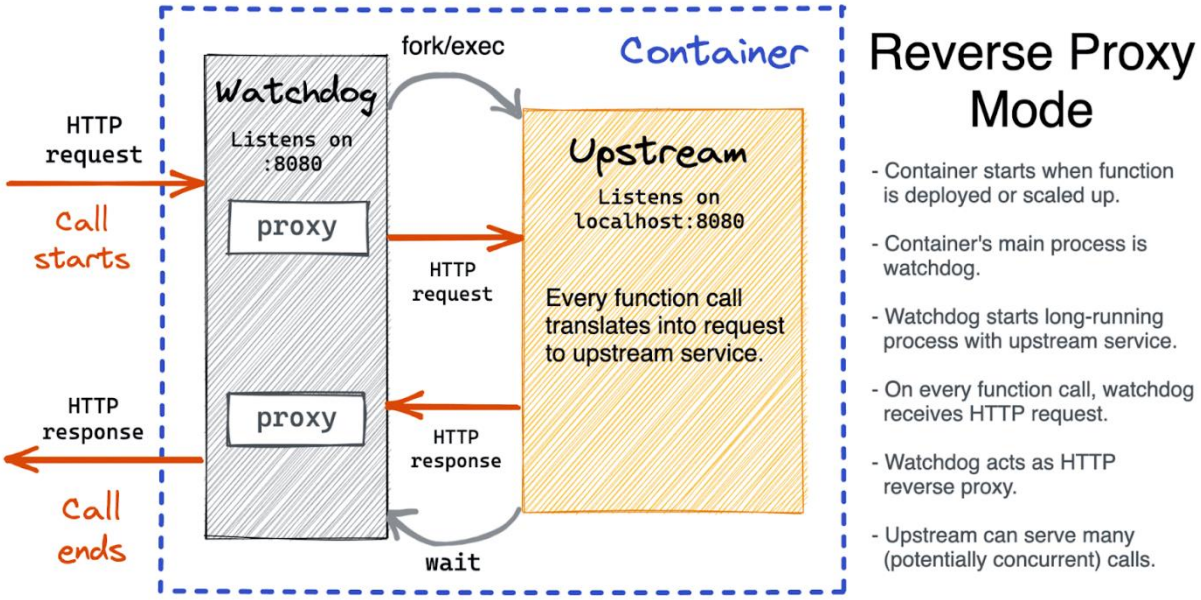

Figure 54: of-watchdog functionality

5 Conclusion

Developing and execution application on infrastructure from the edge to the cloud has fundamentally changed due to the ever-increasing demands for high performance computations and low power designs. Even though compute systems' programmability has seen significant improvement over the last few years, achieving close-to-peak performance and energy efficiency remains a complex and challenging task even for the experts. Fine-tuning programs on both the edge and the cloud remains a
non-trivial and demanding task that has not been fully solved yet, neither by the academia nor by the industry.

In this deliverable, we present the SERRANO's contribution on automatic frameworks for seamlessly integration of heterogeneous workload-aware performance improvement. We describe our work for automatic optimization tools for both FPGA and GPU accelerators, runtime memory management for FPGA sharing and abstractions for HW acceleration of short-lived tasks. Our tools can achieve speedups of up to x2.3 in terms of performance in comparison with native approaches.