http://www.everest-h2020.eu

dEsign enVironmEnt foR Extreme-Scale big data analyTics on heterogeneous platforms

D3.2 — Data management techniques: final version

The EVEREST project has received funding from the European Union's Horizon 2020 Research & Innovation programme under grant agreement No 957269

Project Summary Information

Project Title	dEsign enVironmEnt foR Extreme-Scale big data analyTics on heterogeneous platforms
Project Acronym	EVEREST
Project No.	957269
Start Date	01/10/2020
Project Duration	42 months
Project Website	http://www.everest-h2020.eu

Copyright

Num.	Partner Name	Short Name	Country
1 (Coord.)	IBM RESEARCH GMBH	IBM	CH
2	POLITECNICO DI MILANO	PDM	IT
3	UNIVERSITÀ DELLA SVIZZERA ITALIANA	USI	CH
4	TECHNISCHE UNIVERSITAET DRESDEN	TUD	DE
5	Centro Internazionale in Monitoraggio Ambientale - Fondazione CIMA	CIMA	IT
6	IT4Innovations, VSB – Technical University of Ostrava	IT4I	CZ
7	VIRTUAL OPEN SYSTEMS SAS	VOS	FR
8	DUFERCO ENERGIA SPA	DUF	IT
9	NUMTECH	NUM	FR
10	SYGIC AS	SYG	SK

Project Coordinator: Christoph Hagleitner – IBM Research – Zurich Research Laboratory

Scientific Coordinator: Christian Pilato – Politecnico di Milano

Disclaimer

Deliverable Information

Work-package	WP3
Deliverable No.	D3.2
Deliverable Title	Data management techniques: final version
Lead Beneficiary	USI
Type of Deliverable	Report
Dissemination Level	Public
Due Date	31/01/2024

Document Information

Delivery Date	26/03/2024
No. pages	46
Version | Status	1.0 | Final
Responsible Person	Francesco Regazzoni (USI)
Authors	Francesco Regazzoni, Tom Slooff, Subhadeep Banik, Alberto Ferrante (USI), Jan Martinovic, Jakub Beranek, Katerina Slaninova (IT4I), Karl Friebel, Jeronimo Castrillon (TUD), Serena Cruzel, Christian Pilato (PDM), Michele Paolino, Samuele Paone (VOS), Burkhard Johannes Ringlein (IBM)
Internal Reviewer	Christoph Hagleitner (IBM)

The list of authors reflects the major contributors to the activity described in the document. All EVEREST partners have agreed to the full publication of this document. The list of authors does not imply any claim of ownership on the Intellectual Properties described in this document.

Revision History

Date	Ver.	Author(s)	Summary of main changes
13.10.2023	0.1	Tom Slooff (USI)	Initial draft
3.12.2023	0.2	Tom Slooff (USI)	Updated Anomaly Detection
13.1.2024	0.3	Michele Paolino and Samuele Paone (VOS)	Updated Virtualization
9.02.2024	0.4	Burkhard Ringlein (IBM)	Updated Cloud FPGA communication
28.02.2024	0.5	Serena Cruzel (PDM)	Updated Custom Data Type
28.02.2024	0.6	Subhadeep Banik (USI)	Added Security Library
7.03.2024	0.6	Jan Martinovic and Jakub Beranek and Katerina Slaninova (IT4I)	Updated Data Management Architecture
15.03.2023	0.7	Francesco Regazzoni Alberto Ferrante (USI)	Assembly [1]
23.03.2023	0.8	Christoph Hagleitner (IBM)	Review
26.03.2023	1.0	Christoph Hagleitner (IBM)	Final

Quality Control

Approved by Internal Reviewer	March 25, 2024
Approved by WP Leader	(N.A.)
Approved by Scientific Coordinator	March 26, 2024
Approved by Project Coordinator	March 26, 2024

Table of Contents

1 Executive Summary

This deliverable reports the final version of the Data Management Techniques (DMTs) studied, developed, and adopted within the EVEREST project. It depicts the definition of techniques related to data layout, communication, and security. This deliverable is an updated version of Deliverable D3.1 “Data management techniques: initial version”, and extends this initial version of the DMTs by presenting the optimization, the changes, and the novel techniques that have been developed in the second part of the project. Since our goal is to provide a self contained document that does not require prior knowledge of the content of Deliverable D3.1 to be completely understood. While preparing this deliverable we followed an incremental approach. This means that we started the current deliverable from the text, the figures, and the tables that were already part of Deliverable D3.1. We updated content in this deliverable where needed, but left it unchanged where it was still valid. Next, we added new sections, figures and tables to describe the new results and methods developed in the second part of the project.

We followed the same structure of Deliverable D3.1, describing firstly the general position of the DMTs within the EVEREST data-lifetime explaining the difference between the data management techniques describe in this deliverable and the initial data management plan described in D1.3. Then we present at high level, the final version of the EVEREST data management architecture, which has been updated compared to Deliverable D3.1. Finally, we summarize the efforts done throughout the project discussing the DMTs developed for the FPGA memory architecture, for data allocation and storage, for the FPGA data processing, for virtualization, for custom data types and for providing data protection via anomaly detection and via a library of cryptographic primitives.

2 Introduction

This deliverable presents the final version of the DMTs, whose initial version was provided in deliverable D3.1. The goal of the EVEREST project is to establish the convergence between big data and HPC, focusing on a data-centric perspective and considering software tools that go beyond the border of a single hardware platform. In this deliverable, we focus on the DMTs that deal with the data needed for the execution of an EVEREST workflow.

The initial high level vision of the Data Management Architecture has been refined throughout the course of the project. The updated version of the architecture is depicted in Figure 1. We can see the categorization of the data management process into five tiers. The data management requirements of each tier are addressed by different tools and technologies suitable for the purpose.

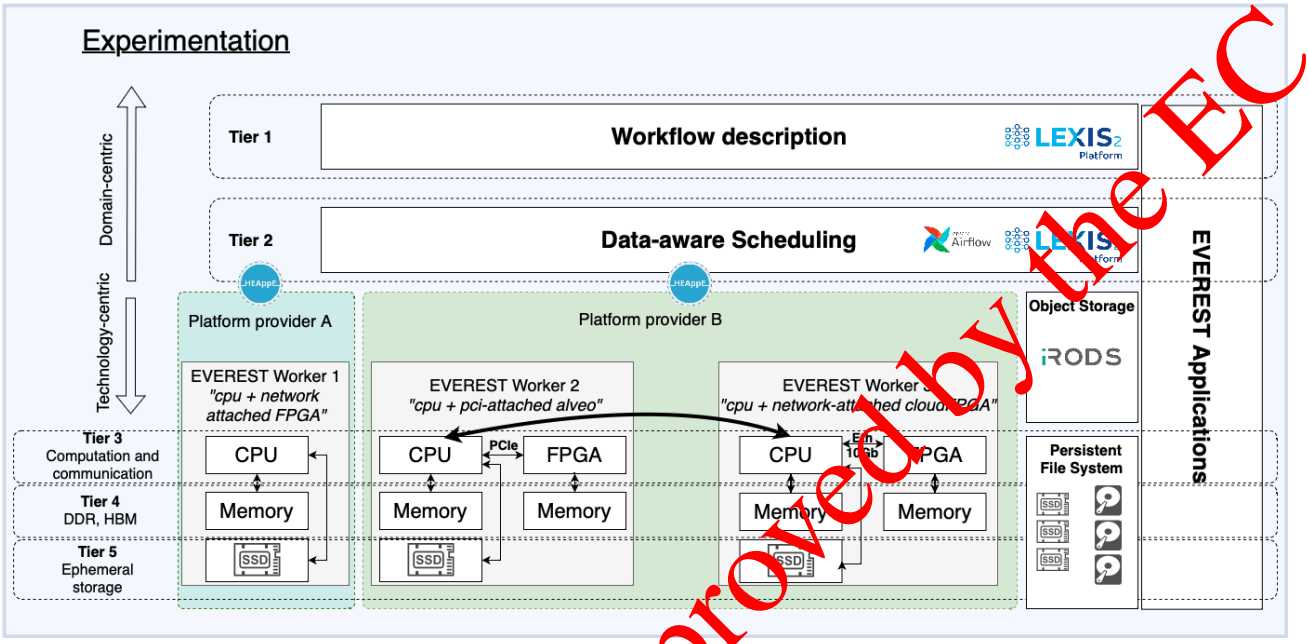


Figure 1 – The EVEREST Data Management Architecture (EDMA)

The first tier is used for the definition of the EVEREST high-level application complex workflows. Tasks in these workflows are already deployed HPC applications, containerized applications, data movement operations, etc. The workflows on this tier are represented by directed acyclic graphs (DAG) implemented in Python and orchestrated by the Apache Airflow [5] using custom operators implemented as a comprehensive library which is part of the LEXIS Platform [37, 38].

The LEXIS Platform implements the concept of a workflow, which executes a particular DAG. This execution accepts a set of input parameters, and the individual tasks are triggered according to their dependencies defined in the DAG file. In the second tier, Apache Airflow ensures the correct ordering of task executions, which triggers appropriate APIs for asynchronous data and HPC job management (using HEAppE middleware [36]) and implements a mechanism for observing state changes of the operations.

The third tier includes the data management within the tightly coupled processing units of the EVEREST platform, i.e., CPUs and FPGAs (PCIe-attached and network-attached). The fourth tier includes the data management at the boundaries of the processing elements and the external fast memory, i.e., Double Data Rate (DDR) and High Bandwidth Memory (HBM). The fifth tier includes the data management at an optional ephemeral SSD-based storage within the boundaries of an EVEREST node.

Persistent storage is provided by persistent file systems or by object stores of the LEXIS Platform Distributed Data Interface (see Section 3.1.2). As it can be seen, EVEREST applications span over all the tiers. This architecture offers to users of EVEREST the possibility to develop applications in a way that is transparent to the actual physical platform where the workflow will be executed.

In the remaining part of this section, we recall the whole EVEREST data-lifetime cycle, which is depicted in

Figure 2. The figure reports three main categories for the EVEREST Data Lifetime:

- Data gathering: The process of collecting data from various sources to process in the following stage.
- Experimentation: The main process of performing calculation on data to derive useful insights for the EVEREST applications.
- Data sharing: The process of offering the results of the previous stage to other interested parties, either in a confidential, open-access policy or as an input to the gathering stage in the form of a feedback loop (reintegration).

Using the categories identified in the figure, the activities presented in this deliverable are positioned mostly on the data during the experimentation phase.

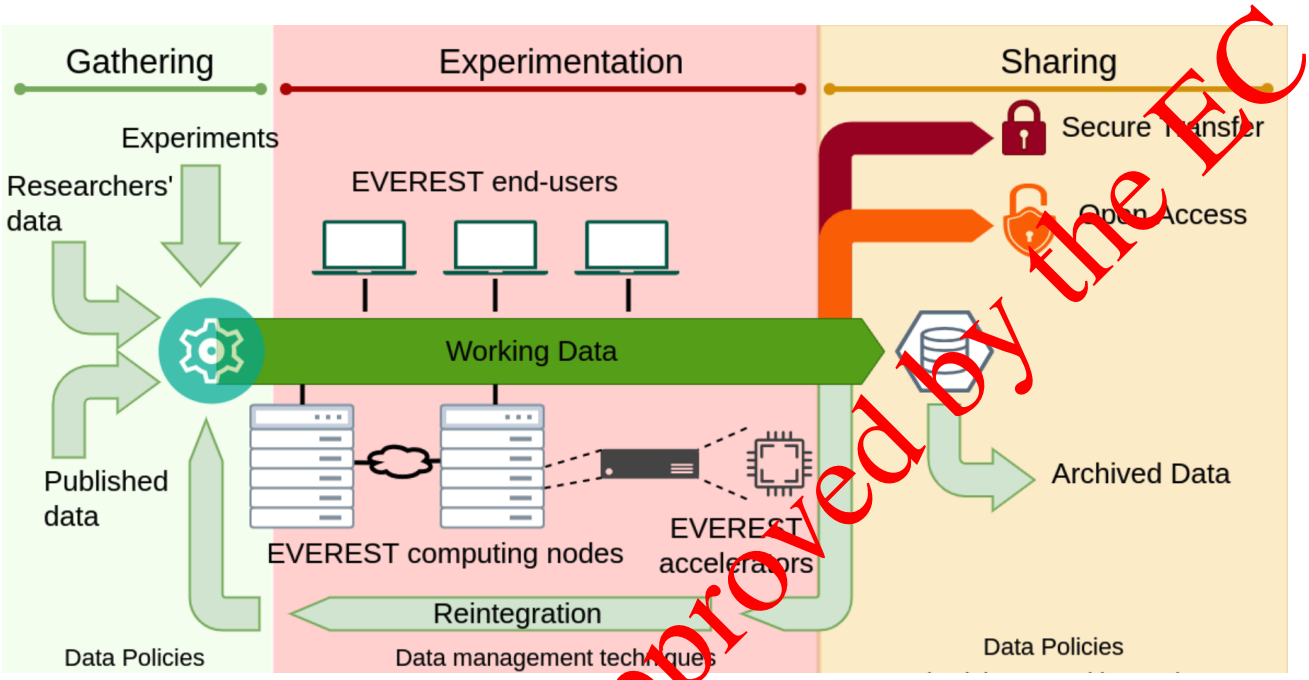


Figure 2 – The EVEREST data-lifetime, data gathering and sharing are mostly covered in deliverable D1.4, this deliverable D3.2 addresses the data management techniques used for experimentation.

3 Data Management Techniques

The DMTs reported in this section describe the methodologies and tools that have been developed and used during the whole EVEREST project. Several techniques envisioned, used, and developed in the first part of the project and described in Deliverable D3.1 have been maintained and confirmed also in the second part of the project, thus their description remain unchanged.

A key enabler for DMTs within the EVEREST project is the use of domain-specific programming abstractions. These programming abstractions convey information to the compilation and optimization flow about data-centric operations. To support the different use cases, we developed and leveraged abstractions for data structures (e.g., tensors in CFDlang and Machine Learning), for known operators (e.g., tensor contractions, stencils and convolutions), and for explicit typed data communication in the Ohua dataflow programming model. These data abstractions ensure a seamless integration in the high-level transformations and in the code generator. More concretely,

- Data structures: With index-free DSLs, the compiler has full control on the data layout and materialization of multi-dimensional arrays. This allows for high-level data partitioning, for advanced polyhedral analysis and scheduling, and for buffering optimizations. Via annotations, the user can specify custom data types, further reducing the memory footprint of data structures and allowing for trade-off exploration between memory bandwidth and area in the reconfigurable fabric.
- Known operators: With explicit syntax for operators, the compiler has rich information about memory access patterns. We thus capture high-level stencils, tensor operators and machine learning kernels. By abstractly specifying a stencil, the compiler can decide on the interplay between data allocation, buffering, re-computation and stencil scheduling. With known tensor operators, like contraction or tensor products, the compiler can decide on the most suitable implementation via algebraic transformations (e.g., sequence of transpositions followed by matrix-matrix multiplication). By capturing the structure of a deep neural network, the compilation flow can decide on how to implement ML engines in the reconfigurable hardware (e.g., streaming or batching).
- Dataflow: Typed dataflow channels, makes it easier to offload computation to accelerated kernels in FPGAs. This includes transparent data marshalling.

These data-centric abstractions are accessible through DSLs and are represented within the EVEREST compiler with suitable intermediate representations (see Deliverable D4.2 and Deliverable D4.5). Compared to what we reported in Deliverable D3.1, extensions to the programming abstractions to provide better control on data abstraction are: (1) a complete Einstein Summation Notation for tensor expressions, and (2) a formal abstraction of dataflow with explicit control of accesses (read and write) to data channels. The former includes support for indirect accesses via subscripts of subscripts, which provide better control on data access patterns. The latter enable better type-safe analysis as enabler for the memory-related optimizations mentioned below.

3.1 Data Allocation and Storage

In EVEREST, we apply several memory-related optimizations to reduce the resource requirements (to possibly fit in more parallel computational units) or to facilitate optimizations of the computational part.

Figure 3 shows the hardware architecture common to the accelerators generated by the EVEREST SDK. In the FPGA side, a computational unit (CU)[1] is replicated one or more times based on the available logic resources and memory channels.

1

To optimize the use of on-chip memories **(1)**, we apply **memory sharing** to reduce the Block Random Access Memory (BRAM) requirements of each kernel generated with HLS. To this end, we exploit the information on the data and the interfaces computed by the compiler during liveness analysis. Based on this, it applies

[1] We use the Xilinx terminology of "Computational Unit" to refer to the largest unit of computation that can be generated by the compilation flow and possibly replicated into the final hardware architecture to parallelize data processing.

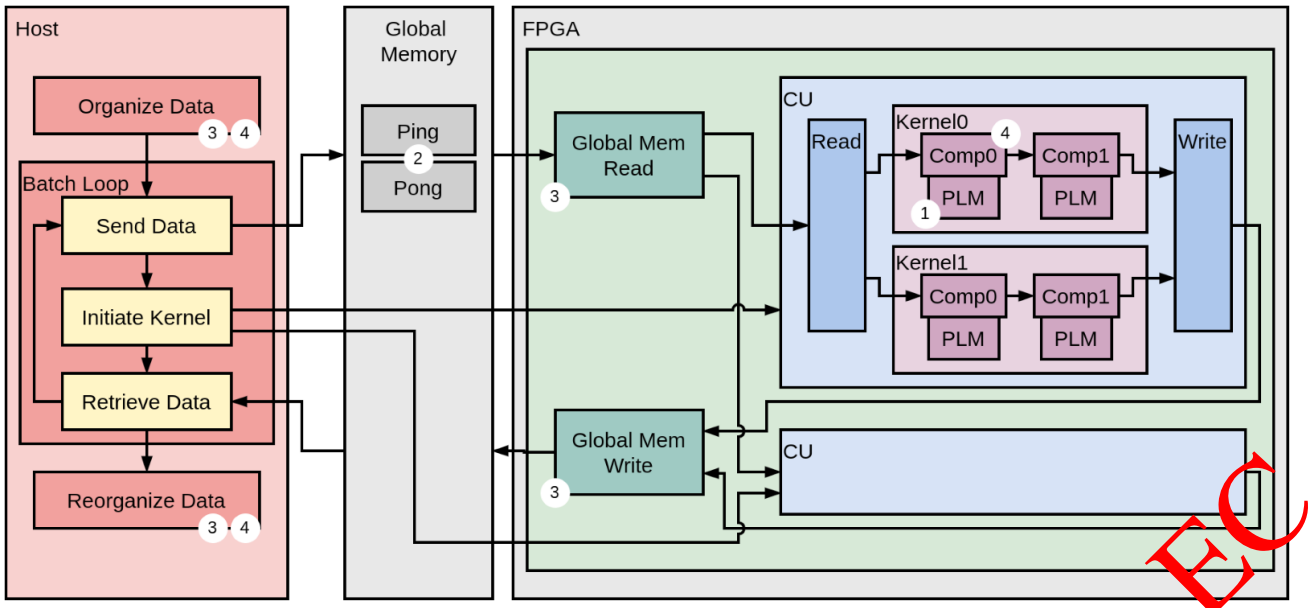


Figure 3 – FPGA memory architectures of the EVEREST accelerators.

sharing transformations based on a memory compatibility graph, which we can easily compute from the compiler for any given schedule. Our memory generation flow then uses this information to generate **zero-conflict memory architectures** while guaranteeing fixed latency of the memory accesses. It can also create multi-port, multi-bank architectures based on the requested HLS optimizations. In this way, the HLS tool can compute a more efficient scheduling of the computational part.

Our hardware generation flow aims at optimizing the data transfers around the kernel implementation produced by the compiler flow. In particular, EVEREST aims at generating hardware accelerators and the associated memory architectures like the one shown in Figure 3.

In this phase, we apply different optimizations on the data allocation to facilitate hardware execution, such as:

- Double buffering: To overlap the host-FPGA data transfers with the execution of the hardware module, we use **double buffering (2)** that requires the allocation of consecutive data chucks to different memory regions that can be transmitted independently.
- Memory layout reorganization: In the case of large bus lines (e.g., the 256-bit Advanced eXtensible Interface (AXI) interfaces of the PCIe-attached memory architectures) or custom data types with reduced bitwidths, EVEREST uses **bandwidth optimization (3)** methods to better exploit the available bandwidth and reduce the number of clock cycles for data transfers. To fully exploit the parallelism, we conceptually divide the bus into smaller and parallel lanes that can be accessed independently by the parallel kernels. Algorithms for better data layouts have been also proposed to maximize the bandwidth utilization. To obtain this layout, the host code data allocation must be modified to interleave the input for the multiple elements before sending it to the FPGA and de-interleave the output after receiving the results.
- Custom precision floating-point: Where full precision is not required power consumption and data-transfer/computation latency can benefit from the use of **smaller bitwidth floating-point types (4)**, while preserving results' precision.

All these optimizations affect the allocation of data in memory and so are implemented in the customization of the host functions (see Deliverable D4.4 for more details). These optimizations apply to the use of the on-chip memories and the data layout in off-chip memories.

For the memory aspects of a hardware accelerator, EVEREST proposes an approach based on a **memory template** that allows for the specialization of the components. The template comprises existing memory primitives, like caches, Direct Memory Access (DMA) engines, prefetchers, and private-local memories (multi-port on-chip memories with fixed-latency access). Based on given area constraints, only part of the data can

stay on chip, while the rest is stored in DRAM (either on the same device or in another that can be accessed through via inter-node data transfers). On-chip data are stored in different memories based on the application data structures but also the type of accesses that are expected. Irregular accesses are implemented with custom **latency-insensitive memory architectures** [46].

Data with regular accesses can be stored in fixed-latency **private local memories** (PLMs) and customized with multi-bank configurations to expose many ports to the accelerator logic. Data reuse buffers can remove unnecessary data transfers. Data accesses with a certain degree of locality can benefit from architectures featuring **caches** that are local or shared with the processor by means of a coherent protocol. We also feature a **DMA engine** to make the data transfers more efficient and we can introduce **prefetchers** to anticipate known data transfers to hide the communication latency. These IP blocks can be also special functions that can include security modules (e.g., encryption/decryption engines) or application-specific transformations (e.g., synthesizable matrix transpose for near-data computing).

This template is general enough to be reused across many kernels, but it can also be specialized based on the accelerator characteristics. For instance, we can vary the number of ports on a multi-bank memory based on the specific access patterns of the given application kernels. Also, components can be removed if they are unnecessary. For example, if the data resides entirely on-chip, the prefetcher can be removed or if there is only a single memory, the multi-channel controller can be simplified. On the contrary, for both network-attached and PCIe-attached FPGAs, the support for multiple channels is important to exploit the bandwidth and supply the parallel execution of the hardware resources with enough data.

3.1.1 Memory-related optimizations

In the following, we describe data optimizations that we include in EVEREST. Such optimizations can be extended to many big data applications that make large use of tensor operators.

On-Chip Memory Optimization. The tensor-based kernels can be often decomposed into a sequence of loops that are executed in sequence. Each loop produces a tensor. Intermediate tensors are used by the next loops. Each of these matrices requires on-chip resources (generally BRAM) to store the values. The number of available BRAMs can limit the number of FPGA kernels. However, once the matrix is not used anymore, the corresponding BRAM resources can be used by the same kernel to store new data. Using the liveness information generated by the compiler, we can reduce the number of on-chip resources required by each kernel. Indeed, arrays with disjoint lifetimes can use the same physical memory banks. Reducing the kernel's BRAM requirements can increase the total number of kernels that we can instantiate. However, sharing opportunities can operate only inside each subkernel. So, the effects of this optimization may be limited.

Host-FPGA Double Buffering. In a naive implementation, the host code transfers the data required for computation into the FPGA. The Compute Units (CU) are then called upon to execute on each of these elements and generate the corresponding output results. The host transfers these outputs back from HBM to its main memory. Each CU interfaces with one PC and we can instantiate up to 32 CUs (each with one kernel) to operate in parallel. However, all communication and execution for a single CUs are serialized. Since the host-HBM communication as much expensive as the computational part, this significantly affects the overall performance. To overlap the host-HBM data transfers with the CU execution, we use double buffering where each computational unit interfaces with two channels. When the total host transfer time for input and output of one batch is less than the total CU execution time for the same batch, the host transfer time is entirely hidden and the CUs are actively executing at all times.

Bandwidth Optimization. The data elements of an application do not usually require more than 64 bits, even less in case of custom data types. However, modern FPGA architectures feature wider busses so using only part of the bus line to transfer the data leads to underutilize the bandwidth. It is possible to "pack" more data elements to significantly reduce the number of clock cycles for data transfers. However, to do so, the CPU code must efficiently prepare the data in the FPGA memory or into the network packets and the accelerator needs to efficiently manage the multiple parallel data to avoid serialization when writing them into the buffers. To fully exploit the parallelism, we conceptually divide the bus into multiple lanes and replicate the innermost kernel as many times as needed within a CU, allowing each kernel to access one of the lanes. This way,

read/write modules still require the same number of cycles, but the accelerator can start the computation of more "hardware threads" in parallel. An additional optimization uses custom data layouts with scheduling principles for maximizing the bandwidth utilization [49].

Dataflow Optimization. Each single execution of the CU reads data from the FPGA memory, execute the kernel operator(s) on them, and write back the results. When the kernel can be decomposed into multiple operations, we can decompose the hardware module accordingly into elementary blocks. Such blocks can be implemented as subfunctions in the kernel that communicate via AXI Stream in a dataflow model. These hardware modules will thus execute in a pipeline, significantly improving the throughput. The number of elementary blocks in each subfunction is a tradeoff between latency and resource requirements. Indeed, having more blocks in the same subfunction increase resource sharing opportunities but also increases the latency of the pipeline stages, reducing the throughput. This optimization improves the performance but also increases the resource usage, potentially limiting the total number of CUs that can be instantiated.

Custom floating-point types. Given the nature of the application data, it is often possible to customize the bitwidth of the data without significant error degradation. From the memory viewpoint, data types with reduced bitwidth require less on-chip memory resources, reducing the number of BRAM units that can be used. Also, reducing the precision of the data allows for creating more "lanes" and thus enabling more parallel computation.

3.1.2 Storage of Data at Cluster Level

Data on the cluster level at IT4I reference infrastructure are handled by the Distributed Data Interface (DDI) service which is part of the LEXIS Platform. The service provides asynchronous data transfers between geographically distant data sources such as federation of iRODS zones. The service uses remote worker processes implemented in Celery framework in Python. These worker processes use native client libraries to transfer the data, in case of the clusters it uses native SSH-based protocols for data transfers like sftp or rsync. It uses userspace level of access to the cluster and does not require any change in the cluster configuration. The credentials and preparation of a work directory are handled by the HEAppE middleware, which the worker process calls.

The DDI service provides REST API for data management, which resembles traditional object storage extended with rich metadata index. The API provides a set of user endpoints for direct data transfers using HTTPS based chunked upload (TUS protocol) and direct downloads. Another set of enpoints is provided to manage the asynchronous data transfers between external data sources and HPC clusters. The LEXIS Platform also offers a Python library Py4Lexis [83] which offers Python API as well as interactive terminal-based interface for all DDI features, including direct access to iRODS zones connected to the LEXIS Platform for large data transfers.

The DDI capabilities are nicely illustrated by their integration in the LEXIS workflows. The user first uses the DDI to upload their input data and metadata to a particular iRODS zone and then executes a workflow, where the uploaded input dataset is specified as paramater. The LEXIS workflow orchestrator then triggers data movement through the DDI, which issues a set of tasks for the worker process, which in turn pulls the data from the remote iRODS zone to a temporary staging area and uses HPC cluster native protocol to stage the data to the cluster. The orchestrator then triggers an HPC job submission through the HEAppE middleware. Once the job finishes, the orchestrator then triggers a DDI operation which transfers the output data produced by the HPC job and stores them as a dataset in a selected iRODS zone along with a set of metadata. These metadata also contain apart from user specified values also provenance metadata about the workflow and its execution used to produce the output dataset.

3.2 Data Processing and Communication

The scope of this subsection is to cover the activities related mainly with the data processing and communication. In this task we define how the different components interact with the memories and communicate with each other. Concerning data processing, this task will analyze the data access patterns, and the alignment of data accesses with the width of the memory, taking into consideration the different memories of the EVEREST

heterogeneous platform. Memory access are also optimized for virtualized environments by enhancing the performance of transactions between the virtual machines and the hardware accelerators, aiming at minimizing data processing latencies and increasing the guests-hardware bandwidth.

We start the description of those activities by firstly providing information about the virtualization technology of PCIe-attached FPGAs and afterwards on the DMT for the FPGA-based EVEREST heterogeneous platform.

3.2.1 VMs guests-hosts PCIe communication extensions

In the Deliverable D3.1. we highlighted the "VMs guests-hosts data transfer optimization" in section 3.2.1, mainly targeting SoC-attached FPGAs. In this section, we are now detailing the extensions developed focusing PCIe FPGAs. More information about such extensions and in general the EVEREST virtualization framework and its components (e.g., ESFM, QDMA drivers, etc) can be found in Deliverable D5.5.

In particular, the developed extension aims at finding a way to notify the host when an FPGA kernel is in use; in some way, such information manages the FPGA accelerators lifecycle. The use case we imagined for the Everest project involves having multiple VMs, each with one VF (FPGA kernel) attached. Such VF can be detached anytime by the Everest virtualization framework in the host. From the VMs point of view, this event is aleatory, hence not predictable. The asynchronous detaching of a VF can bring to a critical inconsistent state that brings to a kernel panic in the VM and eventually a crash in the host. The correct execution of this use case depends on when the user detaches the device (VF) and if this is actually in use by the VM (computation ongoing).

The VF driver in the VM takes care of the requests done by the application using the FPGA; after that, the VF driver forwards these requests to the Physical Function (PF) driver which physically issues them on the QDMA. The PF driver handles the requests of multiple VFs; this means that a request must contain information about which VF is about to use which kernel.

The changes made to the PF driver are used to capture this information and forward it to the host. To accomplish this, it was used the *sysfs*, a common mechanism used by drivers to get information to host user space; for that reason, due to the changes done to the PF driver, it creates a new file in the *sysfs* that a host user space program can read to know which kernel is in use and by which VM.

In conclusion, as shown in figure 4, the PF driver intercepts the requests coming from the VFs and modifies the sysfs file accordingly to which VF is issuing the request.

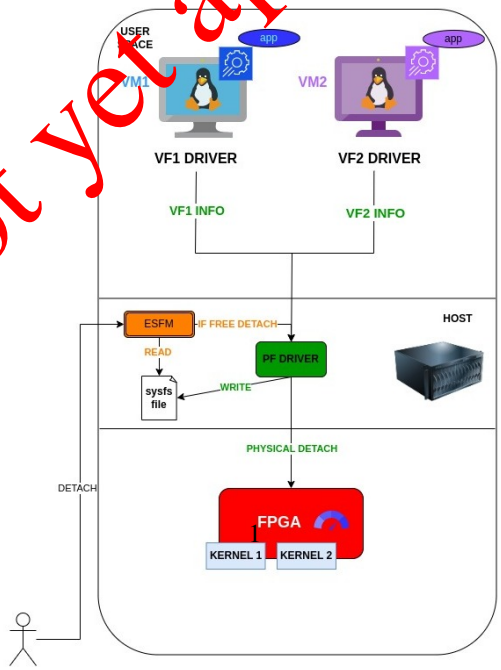


Figure 4 – Diagram showing how the QDMA driver extension to enable the graceful detach feature.

3.2.2 Data management techniques for the cloudFPGA platform

One of the foundational tasks of the workload processing on a heterogeneous system, like the EVEREST platform, is the data movement in and out of the discrete computing resources. In the case of cloudFPGA, as one of the computing nodes of the EVEREST platform, the primary medium to transfer data is the network. In the deliverable D3.1 we explained the data management techniques for the IBM cloudFPGA platform in detail. The continuous evaluation of the requirements for EVEREST as well as repeated performance tests confirmed that the architecture presented in Figure 5 and Table 3 initially reported in Deliverable D3.1 still fulfill the requirements of the project, thus it was not changed in the second part of the project. With the purpose of making this deliverable a self contained document, the remaining part of this subsection, summarizes the description of the data management techniques for the cloudFPGA platform that were extensively presented in Deliverable D3.1, reporting here from the same deliverable also all the needed architectural figures, block diagrams, and tables relevant for the explanation.

IBM has developed a TCP/UDP offload engine on the FPGA logic. The FPGA developer is provided with the option to move network data to an AXI memory map (MM) interface or to an AXI4-Stream interface inside the FPGA. The data can be processed directly at line-rate or they can be buffered at the DRAM of the cloudFPGA module.

The cFDK offers two Shells, the Kale and the Themisto. Both shells provide access to the "MEM" subsystem in order to interact with two physical DDR4 memory modules. A block diagram of MEM is depicted in Fig. 5. The memory channel #0 (MC0) is dedicated to the network transport stack (NTS) of the Shell. The user's application has full access to the 8 GB of memory channel #1 (MC1).

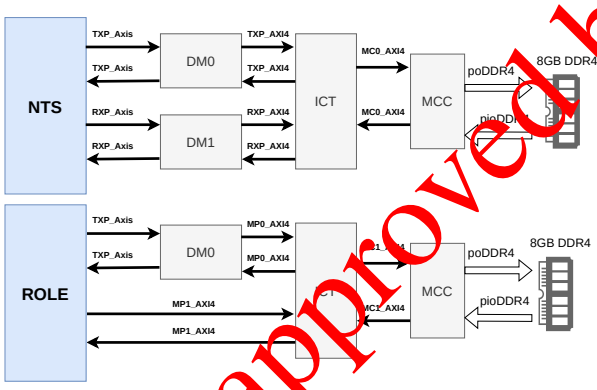


Figure 5 – Overview of the cloudFPGA memory subsystem ("MEM").

Table 3 lists the sub-components of MEM and provides a link to their documentation as well as their architecture body.

Entity	Description	Architecture
MEM	Memory Sub-System	memSubSys
MEM/MC0	Memory Channel 0	memChan_DualPort
MEM/MC1	Memory Channel 1	memChan_DualPort_Hybrid
MEM/MC[0,1]/DM[0,1]	AXI Data Mover	PG022
MEM/MC[0,1]/ICT	AXI Interconnect	PG059
MEM/MC[0,1]/MCC	UltraScale Architecture-Based	PG150

Table 3 – The sub-components of cloudFPGA MEM module.

Simultaneous support for AXI4 memory-mapped and streaming interfaces for cF DRAM. In EVEREST we choose to provide both popular interfaces for Xilinx FPGAs, i.e. an AXI4-full and an AXI4 stream. This is achieved by implementing, as part of the Themisto Shell and the cFDK, an AXI DataMover and an AXI Interconnect module to interface the ROLE to the physical DDR4 module. The AXI DataMover is a key interconnect infrastructure IP that enables high throughput transfer of data between the AXI4 memory-mapped and AXI4-Stream domains.

Network to DRAM buffering. Instead of having the accelerator to handle the network data directly, two

soft-modules, namely N2MS and N2MM, can process the network data stream and store it in the DRAM, using either an AXI streaming interface or a memory interface respectively. As shown in Fig. 6, the network stream is accumulated to a local memory until a payload of 4KB is reached. A 4KB burst write follows. From that moment, a notification signal is raised to inform the accelerator that there are data into the memory so that the processing can start. While the accelerator is processing the data, the N2MS and N2MM engines are working independently. This allows the accelerator to utilize the entire bandwidth of the DRAM module over the AXI interface. If the accelerator is programmed in C++/HLS, the #pragma dataflow directive can be employed to allow for an initiation interval (II) of 1. This allows the accelerator's pipeline to be utilized at every clock cycle.

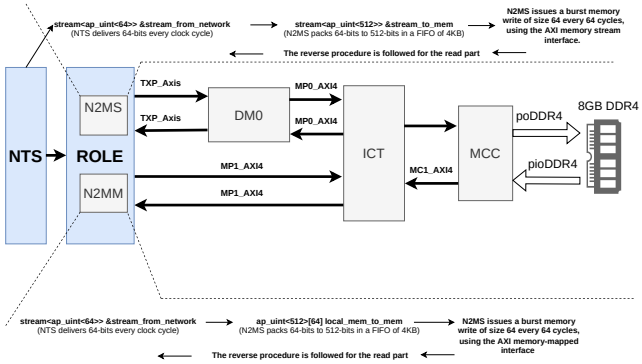


Figure 6 – A cF data management technique to optimize accelerator throughput.

Network payload encapsulation. Another DMT deals with the encapsulation of special data over the TCP/UDP payload. For such cases in cF we provide two ways of handling such data:

- Through an AXI Lite memory-mapped channel provided by Themisto Shell.
- Through encapsulation over the TCP/UDP payload. The user has the freedom to encapsulate a custom data header of arbitrary length (less than the configured MTU) into the UDP/TCP payload.

3.2.3 Data management techniques for Xilinx Alveo accelerators

Each Alveo card of EVEREST combines three essential things: a powerful FPGA for acceleration, high-bandwidth DDR4 memory banks, and connectivity to a host server via a high-bandwidth PCIe Gen3x16 link. Alveo designs are split into a shell and role conceptual model, similar to the other EVEREST platform. The shell contains all the static functionality. Your design fills the role portion of the model with custom logic implementing your specific algorithm(s). You can see this topology reflected in Fig. 7.

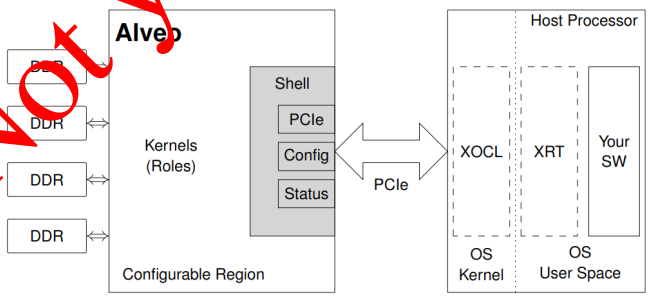


Figure 7 – Conceptual Topology of EVEREST Alveo-based platform

The Alveo FPGA is further subdivided into multiple superlogic regions (SLRs). Alveo cards have multiple on-card DDR4 memories, and in the context of OpenCL are collectively referred to as the device global memory. This memory has a very high bandwidth, and the EVEREST kernels can easily saturate it if desired. There is, however, a latency hit for either reading from or writing to this memory.

The PCIe lane, on the other hand, has a decently high bandwidth but not nearly as high as the DDR memory on the Alveo card itself. In addition, the latency hit involved with transferring data over PCIe is quite high. From

the perspective of designing an EVEREST acceleration architecture, the important points with respect to DMT are:

- Moving data across PCIe is expensive. For larger data transfers, bandwidth can easily become a system bottleneck.
- Bandwidth and latency between the DDR4 and the FPGA is significantly better than over PCIe, but touching external memory is still expensive in terms of overall system performance.
- Within the FPGA fabric, streaming from one operation to the next is effectively free.

For the Alveo cards, the software library that fundamentally interacts with the Alveo hardware is the Xilinx Runtime (XRT). While XRT consists of many components, its primary role can be boiled down to three simple things:

- Programming the Alveo card kernels and managing the life cycle of the hardware
- Allocating memory and migrating that memory between the host CPU and the card
- Managing the operation of the hardware: sequencing execution of kernels, setting kernel arguments, etc.

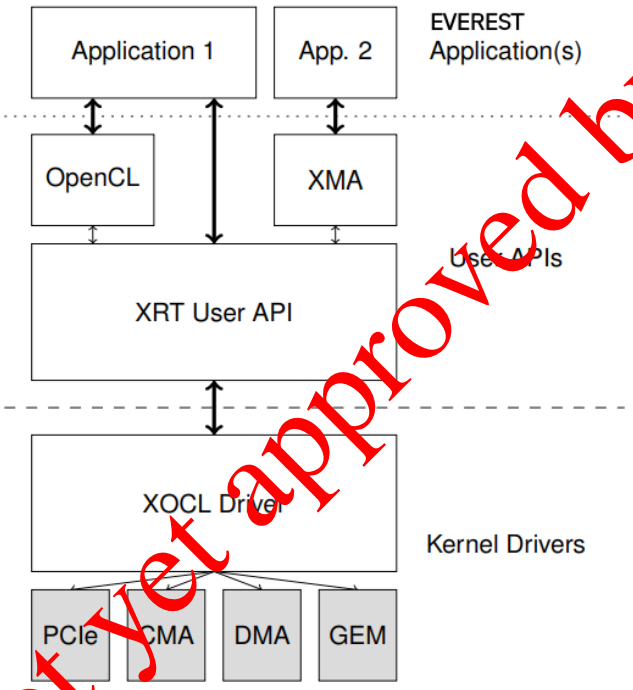


Figure 8 – XRT Software Stack

The XRT is a low-level API, which might be used for advanced use by direct interaction. In addition, there are available higher-level APIs, such as OpenCL, the Xilinx Media Accelerator (XMA) framework, or others. Figure 9 shows a top-level view of the available APIs. In EVEREST we are utilizing both the high-level OpenCL and the low-level XRT APIs.

The memory of the EVEREST Alveo-based platform has six attributes. Given a pointer to a data buffer, that data pointer may be virtual or physical. The memory to which it points may be paged or physically contiguous. And, finally, from the standpoint of the processor that memory may be cacheable or non-cacheable.

An allocated memory space on the host side results in virtual page addresses of 4KiB. Moving those pages to Alveo's memory will result in resolving virtual page addresses to physical memory addresses. The next step is the assembling of these physical addresses into a scatter gather list to enqueue to a DMA engine with scatter-gather capability, which would then copy those pages one-by-one to their destination. The reverse work is followed for moving a buffer from Alveo to a virtual, paged address rang on host memory.

Fig. 9 depicts a simplified view of the system, while the virtual to physical mapping is reported in 10. For performance reasons, within the Alveo card, accelerators operate only on physical memory addresses and data is always stored contiguously.

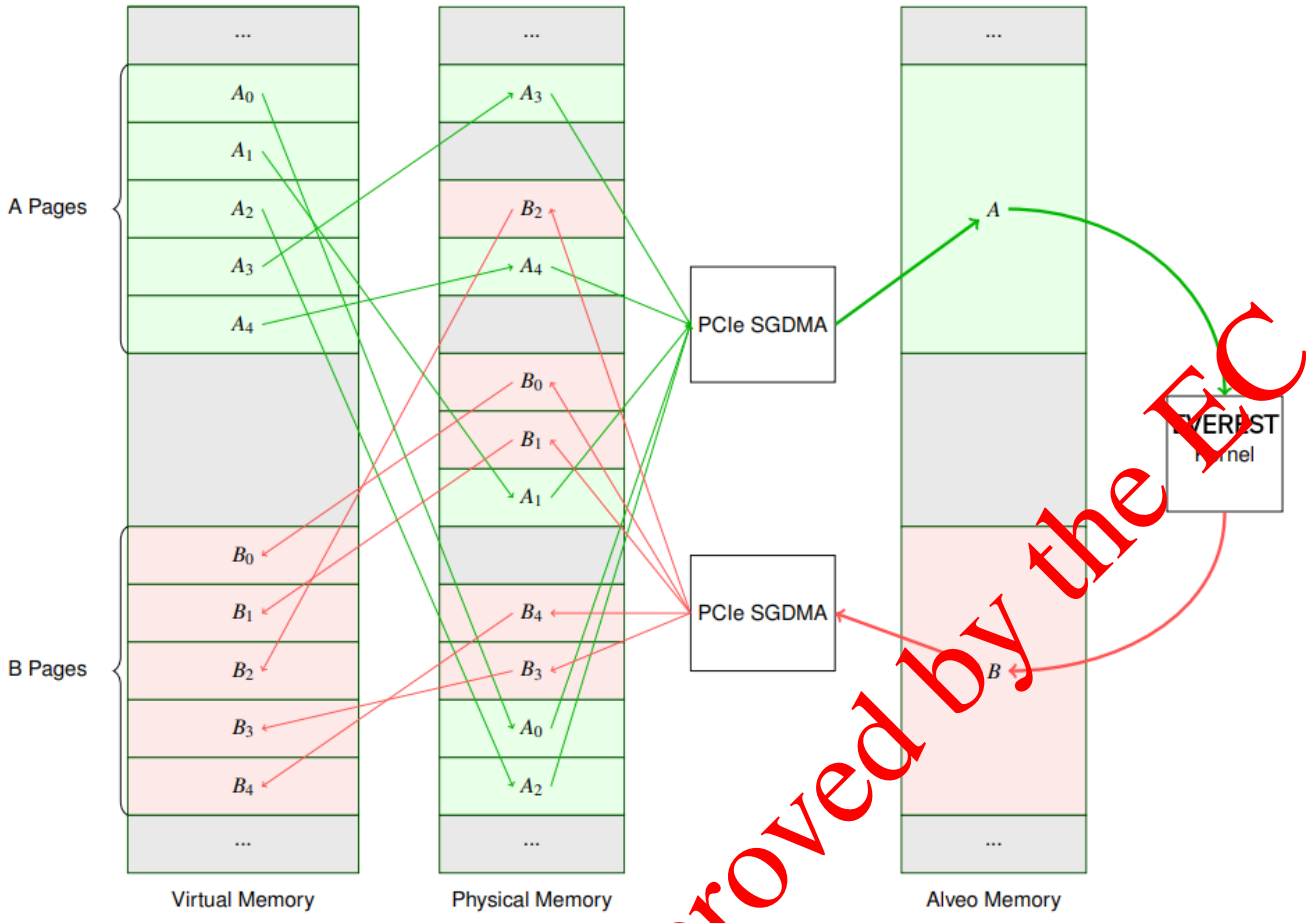


Figure 9 – Virtual Memory transfers to/from Alveo

Building and managing those scatter gather lists and managing the page tables can become time consuming even for a fast host processor. It's much easier to build a scatter gather list, though, if all of the pages are contiguous in physical memory. Modern operating systems provide memory allocators for this purpose. In EVEREST we exploit this DMT to decrease the complex logic of advanced scatter-gather DMA lists. This results in lower FPGA resource utilization and lower latency in host-FPGA memory operations.

3.2.4 HLS data management techniques

In this subsection we update the description of the data management techniques done at the HLS level. We concentrated on loop pipelining and custom data types. Our description start from the text, figures, and tables previously reported in Deliveralble D3.1, which have been update where needed, most notably in the addition of new results on the HLS optimization, obtained in the second part of the project and in the description of the cluster level communication.

Considering the data processing at the level of the FPGA accelerator leads to the analysis of data access patterns, and of the alignment of data accesses with the width of the memory. This allows defining where the HLS directives characterizing the memory access pattern should be inserted together with unrolling directives and loop transformations to ensure alignment and improve performance. Another example is provided by transformations like loop pipelining when applied at a higher level (e.g., MLIR), as they can take into account latency-insensitive memory accesses and increase the instruction-level parallelism. In the following sections of the deliverable we elaborate on some of the techniques that we exploit in EVEREST.

3.2.4.1 Loop pipelining

Loop pipelining aims at overlapping the execution of multiple loop iterations. This technique has been successfully used in compiler infrastructures for decades [42], and it generally consists of two steps: loop scheduling and code generation. Depending on the available computation and memory resources, and if inter-iteration data dependencies allow it, a pipelined loop can issue the execution of a new iteration at every clock cycle.

The proposed approach aims to leverage high-level code optimizations to provide a hardware-oriented input description to High-Level Synthesis. Fig. 10 shows the main steps and tools involved: the input code contains a loop to be pipelined, so the code is first passed to a scheduler to obtain a loop iteration schedule. Then, we implement code transformations that work on the input code and use the schedule to produce the pipelined loop. The resulting code is finally translated and processed by the HLS tool to generate an accelerator description in Verilog/VHDL.

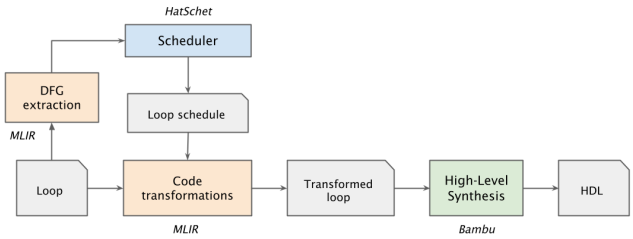


Figure 10 – Overview of the optimization flow for synthesis-oriented loop pipelining starting from MLIR description.

As previously mentioned, loop pipelining requires a scheduling phase and a code generation phase; we will introduce here a simple example that will be useful to illustrate these steps more in detail. Let us consider a for loop that reads values from an array, multiplies them with a constant, and writes them into another array. A single iteration of this simple loop contains three operations: load, multiply and store. Figure 11 represents the data flow graph of one iteration; clearly, the three operations depend on each other and cannot be parallelized.

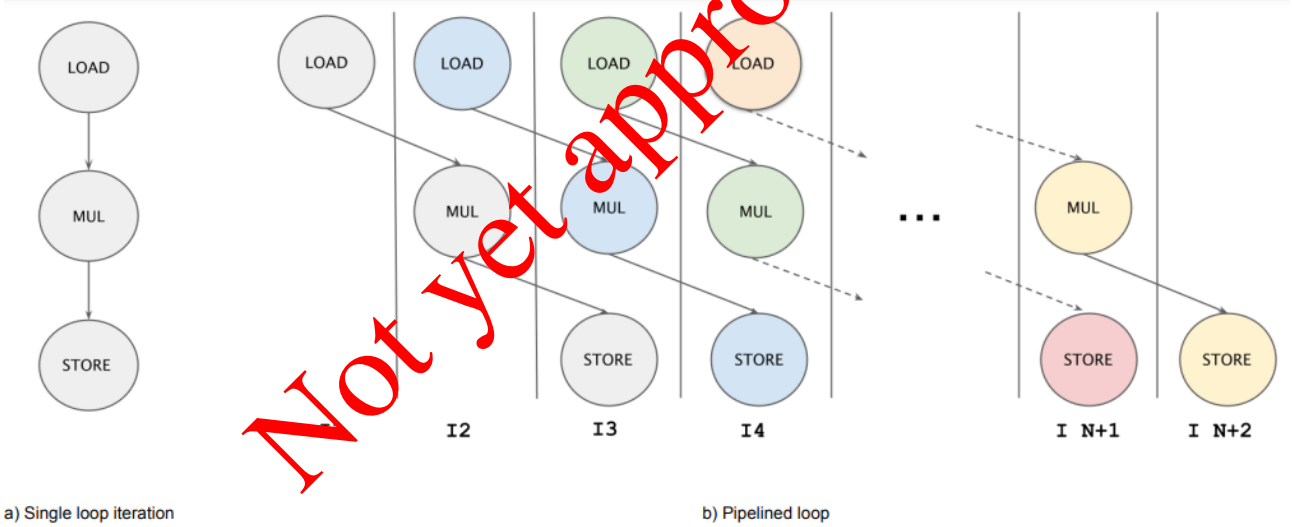


Figure 11 – Creation of a pipelined loop.

Loop pipelining allows scheduling operations from different original iterations together: as these operations would not depend on each other, they could be executed in parallel without constraints. The result is shown in Figure 11b), where each column represents one iteration of the new loop, and operations originating from the same original iteration use the same color. By overlapping original iterations, loop pipelining eliminates the parallelization constraints: all operations within the same iteration are independent now, so they can be executed in parallel. Incomplete iterations at the beginning form a loop prologue; the last few iterations are also incomplete, and they form a loop epilogue. The new loop is built of the complete iterations between prologue and epilogue. In the example shown in Figure 11b), iterations I1 and I2 belong to the loop prologue, I N+1 and

I N+2 represent the epilogue, while the actual new loop starts from I3. If we assume that all functional units execute in one clock cycle, the achieved II in this simple example is equal to 1, as shown in Fig. 12.

CYCLE	LOAD0	STORE0	MUL0	
0	LOAD			PROLOGUE
1	LOAD		MUL	
2	LOAD	STORE	MUL	NEW LOOP ITERATIONS
3	LOAD	STORE	MUL	
	...	...	...	
N+1		STORE	MUL	EPILOGUE
N+2		STORE		

Figure 12 – Pipelined loop schedule.

Within the proposed flow, scheduling is performed by HatSchet, and code generation is implemented as a set of transformations in MLIR; the pipelined loop is then passed to Bambu to obtain an HDL implementation. It represents an alternative to other loop pipelining approaches that delegate scheduling and pipelining to the HLS tool itself. Bringing loop pipelining (and possibly other optimizations) outside the scope of the HLS tool has significant advantages: for example, the developer is more in control of the applied techniques, as their effects are visible in the transformed IR. Moreover, applying transformations on a specialized, higher-level abstraction increases flexibility, portability, and requires less time than implementing and exploring different techniques within the HLS tool. Furthermore, MLIR is built to allow easy integration between different optimizations: this means that loop pipelining may be combined with other techniques to create inputs to the HLS tool that are more appropriate to generate efficient hardware accelerators. The results of the proposed approach for loop pipelining applied to Polybench kernels are presented in Table 4.

Benchmark	Baseline (cycles)	Pipelined (cycles)	Speedup
2mm	214.914	76.482	x2.81
3mm	304.576	117.428	x2.59
atax	41.911	16.869	x2.48
bicg	41.887	8.749	x4.79
doitgen	130.742	69.222	x1.89
gemm	244.622	83.002	x2.95
gemver	90.122	25.845	x3.49
mvt	43.362	16.722	x2.59
syr2k	227.082	70.910	x3.21
syrk	153.182	57.490	x2.66
trmm	74.362	37.392	x1.99

Table 4 – Performance of selected Polybench kernels: baseline and MLIR loop pipelining.

The modularity and flexibility provided by MLIR allow to introduce optimizations, as we did with affine loop pipelining, and to experiment with existing ones, to generate optimized IRs for HLS. The affine dialect provides a growing set of loop-oriented transformations as compiler passes, which can easily be enabled or disabled. Even if some of them are also available as backend HLS optimizations triggered by pragmas, applying them at the MLIR level allows to decouple loop optimizations (which do not necessarily require hardware-related considerations) from the backend HLS tool, and thus enhance portability.

Loop pipelining can provide performance benefits on its own, but it can also be coupled with different optimizations to explore different design points with different performance/area trade-offs. We explored a few different options on the gemm kernel with the Bambu backend: Table 5 shows that it can be beneficial to increase the number of iterations in the pipelined loop through loop permutation, which reduces the number

of cycles with a minimal increase in resource utilization. If we increase the size of the loop body through unrolling, instead, we obtain an even faster design at the cost of significant area consumption. The same exploration of design points would require manual modifications on the code when done at the C/C++ level; for typical HLS optimizations such as loop unrolling, this can be as simple as adding a pragma, but it can require significant code rewriting for other transformations (including loop permutation). In an MLIR-based design flow, optimizations can be exposed as compiler passes and compiler options that are easier to enable/disable in a design space exploration phase.

Optimizations	Cycles	DSPs	LUTs	Slices	Registers	Frequency	Speedup	Slices overhead
none	157122	10	1678	724	1397	102.27 MHz	baseline	baseline
loop pipelining	82362	20	3024	1303	3576	101.48 MHz	1.91x	1.80x
loop permutation + pipelining	81182	20	3006	1306	3413	100.94 MHz	1.93x	1.80x
loop unrolling + pipelining	17642	100	21380	8075	18671	90.39 MHz	8.91x	11.15x

Table 5 – Effect of affine optimizations on gemm (double, mini) synthesized by Bambu.

3.2.4.2 Custom data types

Most of the HPC applications deployed on cloud servers deal with complex computation flows operating on floating-point data. Floating-point data types are commonly provided in two formats only, single- and double-precision, and therefore, if the computation does not fully exploit the available range, floating-point calculations result in wasted precision and power consumption. It is not worth addressing this waste when targeting CPU execution, since arithmetic units in modern general-purpose processors are highly optimized to handle single- and double-precision data types, and even if software-based implementations of smaller precision floating-point types are available (e.g., through the MPFR library), they usually bring no improvement to the computation time nor the power consumption.

When designing a hardware accelerator, on the other hand, it is possible to generate ad-hoc functional units able to deal with custom data types. In this case, an application able to exploit computation on floating-point data with smaller bitwidth is able to benefit from this technique in many aspects: when targeting FPGAs, it may result in significant improvement in computation latency and resource usage, which can lead to faster and more power-efficient accelerator designs. Furthermore, with a smaller bitwidth memories can be restructured resulting in a smaller memory footprint and faster memory accesses during the computation flow.

Automated generation of custom floating-point functional units is available within the EVEREST SDK [30], and high-level optimization phases are able to exploit it if the considered application, or part of it, can benefit from this technique. At the end of the optimization flow, custom precision floating-point types are implemented and integrated into the accelerator design by the PandA-Bambu HLS tool (details are in Deliverable D4.2).

In software, floating-point formats are mainly based on the IEEE standard, and their precision is classified from half to quadruple. Most of the processors available provide a hardware implementation of single- and double-precision functional units. Only recently they have started to add hardware support to half-precision (see FP16 or BFLOAT16 formats), mainly because of their use in artificial intelligence applications. In an FPGA accelerator, however, designers have more freedom to choose different formats, for example by relaxing the number of bits required for the mantissa and exponent. This is a parameter that can be considered during the EVEREST code-variant generation to generate alternatives that allow the run-time system to trade off accuracy against resource consumption on the FPGA fabric. This has been achieved by integrating support for parametric floating-point formats (with a variable number of bits of exponent and mantissa) into the higher-level compiler infrastructure and into the HLS engine that generates the hardware accelerators.

To allow optimization of floating-point data during the HLS process, we relaxed the constraints on the number of bits of the mantissa and exponent of the IEEE 754 standard. We do not consider our approach, called TrueFloat, to be a novel data type, since it is based on existing standards and uses their representation and format. The novelty of TrueFloat lies in the fact that it allows the HLS tool to optimize floating-point operations before producing the Verilog: this is not possible in current approaches based on predefined RTL cores, where the implementation of the floating-point operations is taken from a library without optimization at the HLS level. Comparisons on basic floating-point operators such as adder, multiplier, and divider have been

Table 6 – Custom precision floating-point adder.

		TrueFloat				FloPoCo				Template HLS			
Target	Spec	Slices	LUTs	Cycles	Frequency	Slices	LUTs	Cycles	Frequency	Slices	LUTs	Cycles	Frequency
Zynq7000 100MHz	e9m38	193	584	6	104.21	163	534	5	104.76	192	519	9	101.90
	e8m23	122	340	5	101.06	103	291	5	104.49	125	300	8	109.16
	e5m10	57	157	4	103.38	52	160	5	109.86	71	159	8	158.81
	e3m4	28	93	4	113.01	23	70	4	117.20	43	93	7	164.96
Virtex7 200MHz	e9m38	185	590	6	205.42	193	545	8	208.89	219	508	10	268.96
	e8m23	123	363	5	216.54	118	337	7	223.76	133	338	9	223.31
	e5m10	54	161	4	215.10	48	161	6	239.12	63	143	8	259.13
	e3m4	26	81	4	235.34	29	76	6	306.74	38	93	8	321.75
Virtex7 400MHz	e9m38	233	640	11	417.71	207	566	14	360.23	253	638	17	413.22
	e8m23	141	390	8	394.16	126	254	12	370.23	168	399	17	445.43
	e5m10	75	164	6	419.11	61	167	10	450.24	98	197	14	489.72
	e3m4	38	97	6	408.83	41	94	9	414.07	55	124	13	524.93

carried out against two other research tools: FloPoCo [25] and Template HLS [50]. Both these implementations exploit a custom floating-point representation slightly different from the IEEE 754 standard to achieve a simpler exception handling without impacting precision, at the cost of two additional bits: thus, a double-precision floating-point which requires 64 bits to be represented in the IEEE 754 format would require 66 bits to be represented in the FloPoCo format used in [25] and [50]. An indirect comparison with commercial floating-point cores (e.g., from Xilinx, Altera) is possible since FloPoCo has been already compared with such cores as presented in [28]. Table 6, Table 7, and Table 8 show results of the synthesis for the three floating-point operators for each one of the described implementations. Two different Xilinx FPGA boards have been used, a Zynq7000 (xc7z020clg484-1) and Virtex7 (xc7vx485tffg1761-2), and three different target frequencies (100MHz, 200MHz, 400MHz) have been selected to achieve a fair comparison also considering the flexibility of each solution. TrueFloat operators are synthesized into Verilog code using Bambu HLS, while Verilog operators from the Template HLS library are generated using Xilinx Vitis HLS 2021.2. All accelerators are synthesized using Xilinx Vivado 2021.2 and results are reported after implementation. Five floating-point formats are explored by the benchmark:

- **e9m38** 9-bits exponent, 38-bits mantissa
- **e8m23** 8-bits exponent, 23-bits mantissa (IEEE 754 single precision)
- **e5m10** 5-bits exponent, 10-bits mantissa (IEEE 754 half precision)
- **e3m4** 3-bits exponent, 4-bits mantissa

The proposed approach is quite consistent in delivering a design close to the target frequency while still being competitive with respect to FloPoCo [25] and Template HLS [50] both in terms of latency and resource usage. Results for the TrueFloat floating-point addition (Table 6) are quite similar to FloPoCo ones while our approach is able to achieve a better latency with higher target frequencies. The same stands for floating-point multiplication (Table 7): the TrueFloat multiplication core employs a Karatsuba multiplier as its core multiplying unit, which proves to be quite resilient to different frequencies requirements, while FloPoCo and Template HLS exploit a bit heap to perform the same task. The use of a bit heap seems to be better in terms of resource footprint, but is not suitable for clock-frequencies optimization and thus resulting in slower designs. Finally, similar considerations may be extended to the floating-point division unit (Table 8): the TrueFloat implementation relies on a loop to perform the long division, thus resulting in a non-pipelined core. Conversely, FloPoCo and Template HLS exploit an unrolled version of the base-4 long division algorithm, which is suitable for pipelining, but yields much higher impact on resource usage.

The aforementioned benchmark setup does not include EVEREST platforms like Kintex and Alveo boards, which are evaluated separately in Table 9.

1

Floating-point cores for the Posit representation have not been considered so far, since Posit can not be considered as a simple drop-in replacement for standard IEEE754 floating-point data types, as discussed in [24]. There are many differences in number accuracy throughout the range of represented values between IEEE754 and Posit encoding, operators' behavior may differ and an accurate analysis of the application may be required before applying such a radical transformation. Furthermore, as shown in [29], performance and

Table 7 – Custom precision floating-point multiplier.

		TrueFloat					FloPoCo					Template HLS				
Target	Spec	Slices	LUTs	DSPs	Cycles	Frequency	Slices	LUTs	DSPs	Cycles	Frequency	Slices	LUTs	DSPs	Cycles	Frequency
Zynq7000 100MHz	e9m38	133	331	8	6	111.55	80	216	6	3	74.96	77	153	5	4	105.29
	e8m23	46	92	2	4	118.35	44	68	2	3	84.37	41	47	2	3	122.25
	e5m10	27	50	1	4	126.05	15	27	1	3	108.71	24	38	1	6	185.22
	e3m4	23	57	0	4	105.51	14	38	0	3	122.83	16	41	0	3	145.12
Virtex7 200MHz	e9m38	144	333	8	7	232.99	99	256	6	4	163.15	88	156	5	5	223.11
	e8m23	48	91	2	5	220.60	47	69	2	4	177.39	38	44	2	4	221.68
	e5m10	29	47	1	4	230.14	23	28	1	3	197.39	27	38	1	6	397.30
	e3m4	21	59	0	4	245.63	13	37	0	3	255.29	17	41	0	3	278.94
Virtex7 400MHz	e9m38	224	400	8	15	411.18	144	284	6	6	255.29	84	106	5	10	250.13
	e8m23	75	126	2	11	498.50	44	79	2	5	301.11	55	58	2	10	523.56
	e5m10	31	62	1	5	396.98	31	40	1	4	252.20	34	39	1	9	556.48
	e3m4	26	66	0	4	407.66	23	42	0	4	369.95	28	46	0	6	456.41

Table 8 – Custom precision floating-point divider.

		TrueFloat				FloPoCo				Template HLS			
Target	Spec	Slices	LUTs	Cycles	Frequency	Slices	LUTs	Cycles	Frequency	Slices	LUTs	Cycles	Frequency
Zynq7000 100MHz	e9m38	191	566	26	127.48	478	1603	14	73.96	347	2188	28	127.19
	e8m23	123	360	18	149.85	235	660	9	79.3	304	780	16	131.22
	e5m10	62	172	12	159.79	77	205	6	102.10	91	258	8	116.90
	e3m4	34	104	8	171.73	36	91	4	129.0	43	109	7	174.00
Virtex7 200MHz	e9m38	195	564	26	253.16	504	1453	16	168.86	678	2012	24	222.22
	e8m23	119	359	18	277.39	221	626	11	154.96	357	860	19	254.26
	e5m10	68	180	12	238.37	91	234	7	215.56	105	274	10	251.07
	e3m4	32	99	8	302.38	39	103	5	231.64	37	106	7	313.87
Virtex7 400MHz	e9m38	311	669	49	340.71	640	1562	31	213.49	1250	2876	58	404.04
	e8m23	190	411	19	344.35	253	645	20	236.12	544	1197	37	417.19
	e5m10	75	187	12	404.69	80	195	11	252.33	174	367	22	473.93
	e3m4	48	131	9	438.21	58	140	8	418.23	72	144	13	462.11

Table 9 – TrueFloat custom precision operators on the EVEREST target boards.

		Adder				Multiplier					Divider			
Target	Spec	Slices	LUTs	Cycles	Frequency	Slices	LUTs	DSPs	Cycles	Frequency	Slices	LUTs	Cycles	Frequency
Kintex UltraScale 250MHz	e9m38	113	535	5	266.31	85	365	8	6	262.88	134	679	27	282.97
	e8m23	160	827	5	247.52	90	32	5	2	298.42	88	441	18	301.57
	e5m10	44	216	4	286.86	17	62	1	4	288.10	46	204	12	292.48
	e3m4	24	99	4	295.51	20	106	0	3	326.58	25	118	9	335.57
Alveo U55C 250MHz	e9m38	93	603	5	262.19	78	323	8	5	270.86	126	696	26	327.65
	e8m23	58	331	4	271.37	22	89	4	2	292.65	87	440	18	340.25
	e5m10	39	195	4	278.09	14	50	1	4	328.62	49	204	12	378.07
	e3m4	15	89	4	315.76	25	106	0	3	441.70	23	117	9	475.06
Alveo U280 250MHz	e9m38	102	583	5	250.44	76	322	8	5	278.32	123	696	26	313.68
	e8m23	61	334	4	273.22	25	89	2	4	278.32	86	440	18	342.82
	e5m10	41	194	4	275.86	14	50	1	4	328.62	47	204	12	357.78
	e3m4	19	91	4	312.40	25	106	0	3	441.70	23	117	9	475.06

resource utilization of Posit cores are still not comparable to those of standard floating-point operators. Citing from the conclusions of [24]: "Posit-to-posit operators are shown to be significantly more expensive, both in terms of resources and delay, than IEEE operators for the same input width. For instance, addition and multiplication on 32-bit standard Posit require about 50% more hardware and about 50% more delay than standard-compliant of binary32 floats. This overhead should be put in balance with the increased accuracy sometimes offered by posits. On the example of 32-bit formats, posits offer up to 3 extra bits of accuracy (an 11% improvement) in a limited domain of exponents, while degrading the accuracy outside of this domain due to tapered precision."

The choice of the value format affects the accuracy of the EVEREST application use cases, and trading accuracy with performance and resources consumption is an important design decision during the process of accelerating the application on FPGA. (See REQ5, REQ7, and REQ9 in D2.1.) In general, custom floating-point and fixed-point formats positively impact data-intensive scientific computing use cases whenever numeric stability is not a concern.

As an example, an inverse Helmholtz transform kernel has been synthesized exploiting different floating-point formats. In this case, the same floating-point data type has been applied to the whole application, since it is composed by a single kernel, but mixed data formats are supported as well. The default data type has been modified with respect to both exponent and mantissa bitwidth: the most precise format used is the double-precision IEEE 754 standard format with 11-bit exponent and 52-bit mantissa, while the least precise format features 7-bit exponent and 23-bit mantissa. The presented kernel has been synthesized on a Xilinx Virtex 7 board (xc7vx690t-ffg1930) with a target frequency of 200MHz; the whole design is pipelined with an initiation interval of one clock cycle. Results on overall clock cycles and resource usage (both slices and DSPs) for the different data types are shown in Figures 13, 14 and 15, respectively. It is clear that both latency and resource usage are linearly dependent with respect to the selected data type precision: lower precision data type requires lower resource usage and lower overall latency for the computation. The increase in resource usage observed with lower exponent bitwidths is due to the addition of conversion logic from the default input format (standard double precision) to the internal lower precision data type.

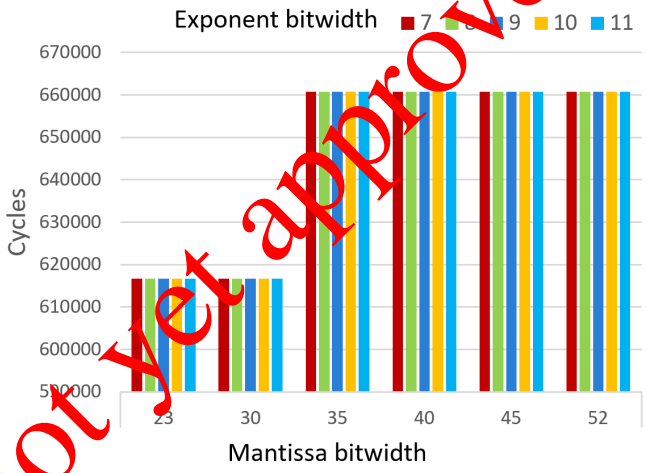


Figure 13 – Number of clock cycles for inverse Helmholtz transform kernel with respect to different floating-point formats.

Custom floating-point and fixed-point formats are relevant also for the traffic modeling application when considering the Probability Time-Dependent Routing (PTDR) algorithm. The most computing-demanding part is based on Monte Carlo simulation. This part is not affected by numeric instability when we approximate the values that describe the car positions and their distance. A design space exploration, where different floating-point precisions are considered, is reported in Table 10; all accelerators are generated targeting an Alveo U55C FPGA and a clock period of 3ns.

1

Another component of the traffic modeling application, i.e., the projection component in the map matching algorithm, has been considered for data types optimization. In this case, however, the analysis of the baseline C++ implementation revealed not only that double-precision trigonometric computation was superfluous, but that the same results could be obtained using simpler fixed-point arithmetic, rendering the application of True-Float types unnecessary. Table 11 quantifies the improvement in terms of performance and area consumption

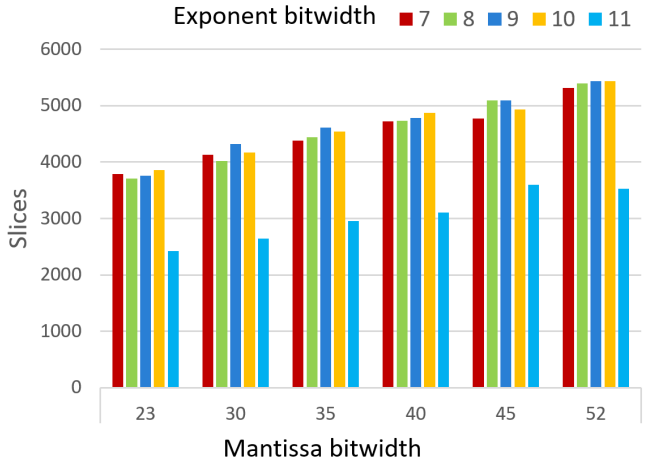


Figure 14 – Number of slices for inverse Helmholtz transform kernel with respect to different floating-point formats.

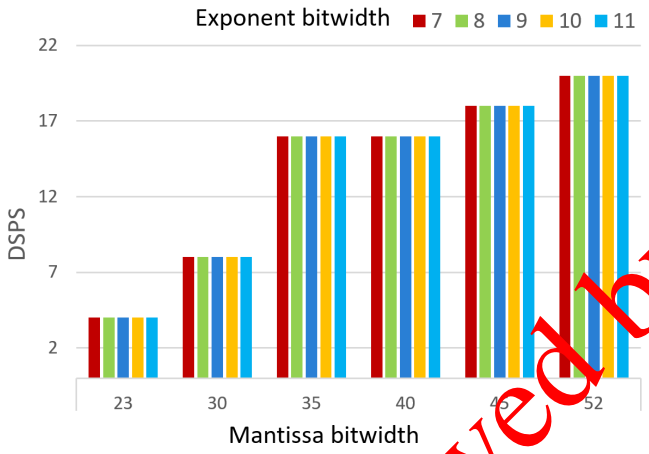


Figure 15 – Number of DSPs for inverse Helmholtz transform kernel with respect to different floating-point formats.

when moving from the baseline implementation to the optimized one based on fixed-point arithmetic; the target is again an Alveo U55C FPGA with a clock period of 3ns.

Finally, we have also examined applying precision reduction techniques to the radiation kernel from the WRF use cases. The implementation details and preliminary exploration results are described in [30]. These experiments show that a naive reduction in precision is feasible, but a technique that considers the structure of the underlying constant data is much more effective (scaled variants). Combining the statistical data about the value distributions with the physical limits of the atmospheric profiles, we arrive at a specialization of the data types for the kernel interface shown in [30].

From these constraints on the inputs of the RRTMGP kernels, we can propagate the value range and accuracy limits to the intermediaries and result values using `base2`. However, due to the structure of the kernel, and the limits to its parallelization imposed by the way it is called from WRF, this transformation does not yield sufficient benefits. In particular, on the FPGA target, the fixed-point kernel achieves a higher throughput while not relying on DSPs, at the expense of a substantial amount of LUTs and FFs. Unfortunately, the overall throughput of the kernel remains limited by the random-access lookup into the spectral constants table, and thus no reduction in II is observed.

Table 10 – Exploration of custom floating-point formats for the PTDR application.

Format	Slices	LUTs	DSPs	Cycles	Frequency
original (fp64)	1396	6962	28	9218302	322.58 MHz
e9m38	1437	6592	26	8570104	323.41 MHz
e11m25	1131	5558	20	6276459	325.41 MHz
e5m10	879	4068	19	5017615	335.68 MHz

Table 11 – Performance and area consumption improvement in the fixed-point map matching algorithm.

Kernel	Slices	LUTs	DSPs	Cycles
baseline	16748	79498	336	31737467
optimized	4445	19280	96	27412500
improvement	73,4%	75,7%	71,4%	13,6%

```
// 0 <= k * 2^77 <= 1e4 +/- 8 ULP [1]
#prescale_ld = 77 : i64
!SpectralConst = !base2.ui16_48
// 0 <= T <= 512 +/- 1e-3 [K]
!Temperature = !base2.ui9_10
// 0 <= p <= 131072 +/- 1e-1 [Pa]
!Pressure = !base2.ui17_4
// 0 <= r < 1 +/- 1e-8 [1]
!MixingFraction = !base2.ui0_27
```

Figure 16 – RRTMGP kernel interface MLIR data types

Comparison with existing methods. Existing formats for floating-point numbers include Posit, which is a new alternative to represent real numbers for computers. The Posit number system has demonstrated a higher accuracy over standard floating-point arithmetic for many scientific applications. However, when it comes to implementing accelerators for these applications, the design methods and the costs are still unclear. In EVEREST, we explored the generation of Posit-based accelerators with Bambu [47]. To add support for Posit arithmetic to the HLS flow, we designed an RTL library of Posit operators based on FloPoCo, and integrated it within Bambu. The design flow of the tool was extended to handle such an additional library without the need of modifying the C/C++ source code. From the point of view of the programmer, the use of Posit arithmetic for the computation of real numbers should be as transparent as selecting between single or double-precision floating-point.

We performed HLS of several benchmarks with Bambu targeting a Xilinx Artix-7 (XC7A100T-1CSG324C) FPGA device. In particular, for the HLS with Bambu we included the options `-no-iob` (so primary ports from the IOB are disconnected, and large arrays can be instantiated in the target device) and `-experimental-setup=VVD` (which provides similar settings for RTL synthesis as the commercial solution Vivado HLS). Under this approach, all objects and internal variables that need to be stored in memory are allocated on BRAMs rather than on external memory.

To select a suitable target frequency for the HLS, we conducted detailed tests for individual arithmetic operators targeting different maximum clock frequencies, which allow us to obtain more details in this regard. Xilinx Vivado 2021.2 was used to perform the logic synthesis for the comparison of hardware resources.

To generate floating-point logic for the accelerators, the option `-flopoco=float` was used, so the floating-point FUs are the ones provided by FloPoCo. However, such units are non-compliant with the IEEE 754 standard: although the memory format is in IEEE 754 format, subnormals are flushed to zero to save resources. This could produce inaccurate results in applications that make use of such small-magnitude data. Also, exceptions are handled in a much simpler way as required by the standard, and just a single rounding mode is implemented (round to nearest, ties to even), rather than the five rounding rules defined in the standard. Therefore, it should be kept in mind that a fully IEEE 754-compliant implementation would incur a much higher overhead than the current one. On the other hand, we extended Bambu with the option `-flopoco=posit` to allocate posit FUs in the final accelerator.

Posit adders require about $1.5\times$ hardware resources (LUTs and FFs) than the corresponding float units, while this overhead is between $2\times$ and $6\times$ for the rest of the operators [47]. Nonetheless, the amount of resources required by Posit32 is always fewer than by Float64 units. Regarding the frequency, all the functional units except the Float64 multiplier satisfy the timing target conditions up to 150 MHz. For a target frequency of 200 MHz a few operators violate the timing constraint, and none of them reach 300 MHz. Therefore, 150 MHz is a clear candidate as the target frequency for the HLS of complex applications. Finally, it must be noted how the iterative algorithm used for division and square root has a direct impact on the latency of such units as the

target frequency increases, especially for the Posit64 format.

HLS results in [47] show that, independently of the target frequency, the 32 and 64-bit floating-point multipliers require 2 and 9 DSPs, respectively, and the corresponding posit multipliers make use of 2 and 12 DSPs, respectively. Also, the design of the floating-point division includes a table for fast computation, which requires 7 and 14 extra BRAMs when synthesizing the 32 and 64-bit designs, respectively.

We believe that these resource requirements are not suitable for the accelerators that we need to create in EVEREST because they would limit the number of accelerators that can run in parallel.

3.2.5 Cluster level communication

The Evkit distributed library allows EVEREST use-cases to distribute their workload over a set of nodes in a cluster. The nodes have to be reachable using the TCP/IP protocol. Communication between the nodes is performed using the ZeroMQ message broker, which allows load balancing computational requests from a single Evkit client to a set of Evkit workers (each running on a separate node). The load balancing is performed in a round-robin fashion. Evkit also allows broadcasting management messages to all connected workers, which is leveraged for synchronizing global shared state. This is described in more detail in Deliverable D5.5.

In terms of data management, Evkit workers primarily exchange data with an Evkit client through their ZeroMQ connection based on top of the TCP/IP protocol. They are also able to load input data files from a (typically network-based) filesystem.

3.3 Data Protection

At the infrastructure level, we provide two types of support to ensure data protection handled and computed in EVEREST. The first aims to detect if something anomalous happened, by analyzing the data themselves. The second type of support is the classical protection offered by using appropriated cryptographic primitives and functions. Here, we rely on these primitives and the associated protocols to ensure that data is not modified or accessed by a non-authorized party. In EVEREST, we use both approaches. In the first phase of the project, we concentrated mostly on the first approach, the one based on anomaly detection. In the second part of the project we followed two main directions. On the one side, we continued and completed the approach based on anomaly detection, and, starting from the initial results obtained in the first part of the project, we built a library of anomaly detection components. On the other, we built a library of classical cryptographic primitives that can be instantiated during the creation of a workflow.

3.3.1 Anomaly Detection

In Deliverable D3.1, Hierarchical Temporal Memory (HTM) has been compared with other anomaly detection techniques and its suitability for EVEREST use case and the the achieved preliminary results drove us to the selection of HTM as the anomaly detection technique to be used within the EVEREST environment. In this section, we summarize from Deliverable D3.1 the motivations behind this choice, then we describe the ADlib, that has been developed in the second part of the project. ADlib includes various models for carry out anomaly detection and a support for the selection of model selection.

Figure 17 shows how anomaly detection fits into the goals of data protection. Anomaly detection mainly handles the integrity of the working data. It can be applied to any of the working data, as well as node metrics, in order to monitor the nodes.

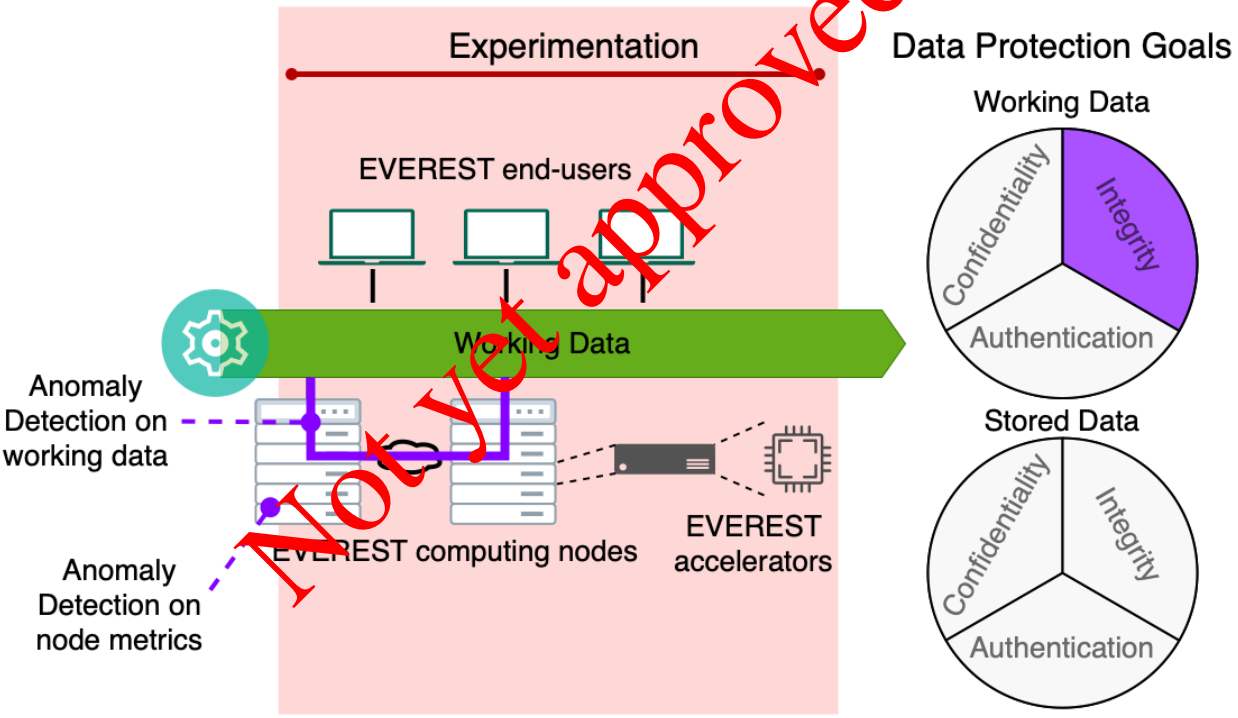


Figure 17 – Anomaly Detection as part of the Data Protection goals.

Anomaly detection within EVEREST is performed on time-series data, which are both, univariate and multivariate. All anomaly detection needs to be done in an unsupervised setting and on both streamed data (which, in our case, require real-time detection) and batched data. Furthermore, to not incur an excessive overhead, the selected technique should be computationally efficient. Ideally, within EVEREST we want to get as close as possible to autonomous anomaly detection. This means that we need a technique which works with any type of data, as well as any combination of different data types. Furthermore, the technique should not need

much tuning to produce good results. HTM is a technique used for prediction and anomaly detection on time-series data [33, 3]. It has several properties which make it highly suitable for EVEREST. HTM is an inherently temporal algorithm. Additionally, the algorithm handles patterns in the data changing over time. This is called concept drift, and it is a challenging problem when using time-series data. HTM is an online and continuous algorithm, which is especially important when the data is streamed and requires real-time detection. Being online and continuous means firstly that the input can be handled by the algorithm immediately on its arrival, there is no need to wait for further input. Secondly, being continuous means there is no need to store previous values of the time-series to learn. This circumvents common techniques which may hamper the computational efficiency of algorithms, for instance rolling windows or batches of data. Researchers also found that HTM is not sensitive in its hyperparameters, which makes it suitable for EVEREST as this means less tuning is required to obtain good performance. Returning to the requirement of handling a large variety of data, the HTM uses encoders to transfer any input data into binary vectors, which it can work with. The general pipeline of the HTM algorithm can be seen in Figure 18.

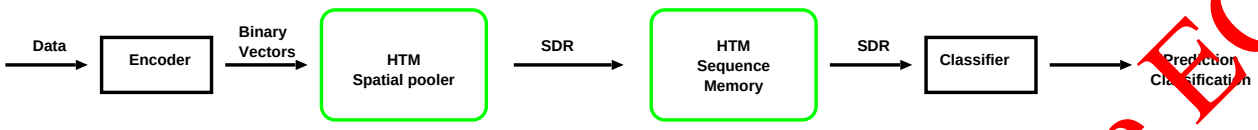


Figure 18 – The general pipeline of the HTM algorithm. Image Source: [21]

These encoders are highly suitable for EVEREST because they allow the HTM to work with any kind of input data so long as an encoder exists which can transform it into binary vectors while maintaining semantic information. Maintaining semantic information in this context means that semantically similar input should have overlapping 1 bits in their binary vector. The spatial pooler is then able to use this data. The spatial pooler learns to be sensitive to patterns in the input space through Hebbian learning, and in general encodes the binary vectors into sparse distributed representation. This is essentially a binary vector where only a small percentage of bits (e.g. 2%) are 1. The sparsity gives some benefits in robustness as well as memory / computational efficiency. The sequence memory then learns to recognize temporal patterns. It is also the sequence memory where prediction and anomaly detection occur.

ADLib

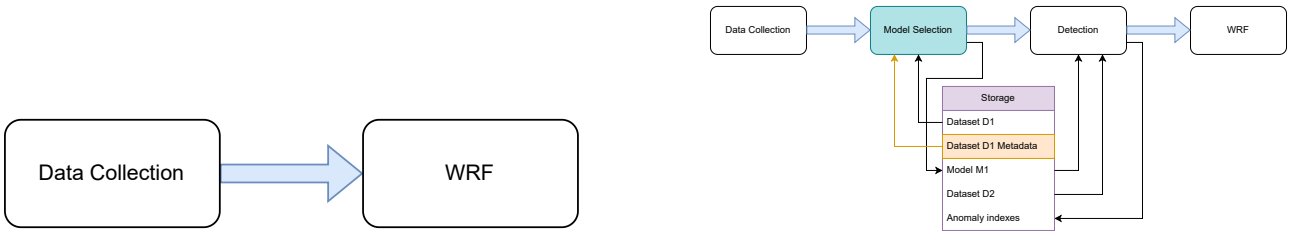
Starting with the chosen technique of HTM, a library has been developed towards the goal of automated anomaly detection. As previously described, HTM was chosen as it was a suitable baseline for all use-cases. The library has since been extended with other models which work differently, i.e. the Autoencoder working in a batched data setting.

The Autoencoder was included as it is more suitable for high-dimensional data compared to the HTM, and it is possible to accelerate this technique using existing libraries. For example, the Autoencoder is the preferred choice for the Duferco use-case as this has higher dimensionality data to process. The choice of technique between the Autoencoder and the HTM is handled automatically during the model selection stage, which will be explained in the following text.

The library comprises of two stages, which are provided to the user. The model selection stage and the detection stage. During model selection, the library considers a given input dataset and aims to find the most suitable model for this particular data. This model is subsequently stored, and will be used during the detection stage. In the detection stage, (new) input data is parsed and processed by the model. The output of the detection stage is a file providing the indexes of datapoints which are considered anomalous in the input data. This standardized file can then be kept for general statistics, or used for actions such as replacing the data, removing the data, or even halting the workflow.

Figure 19 shows an example of how anomaly detection can be introduced to a workflow, and which files are handled. The model selection is coloured to indicate that it is executed only once. It does not need to be explicitly removed from the workflow, there is simply a flag which will make this stage exit immediately.

The library is capable of parsing the most common data formats, and at least all data formats provided by use-case partners. This parsing is done automatically based on file extension. However, some data formats require additional input to allow parsing. For these formats, there is a simple standardized metadata file which



(a) Snapshot of example workflow

(b) Snapshot of workflow with anomaly detection introduced

Figure 19 – A high-level overview of the ADLib stages

the user must provide once during model selection. This metadata file is explained in the documentation of the library such that a user can easily create one. This metadata file is stored along with the model in a single file, and therefore does not need to be provided again in the detection stage.

For the model selection, the library uses the open-source optimisation framework of Optuna, which allows the automated finding of the optimal hyperparameters, as well as the best model. The library uses the optimisation framework for a given amount of time, before outputting the best found model at that point.

In the detection stage, this model is loaded in and used to process the given input datasets. As previously mentioned, HTM is an online continuous learning technique, which is a great quality for this setting. For other techniques, such as the autoencoder, the batch size is determined during model selection stage. In the detection stage, this technique is also continuously trained. This is not ideal as the data may obviously contain anomalous data, but it is required to handle concept drift, and will provide better long-term performance than leaving the model unchanged.

The library is implemented as a python library, and as such can be easily executed provided the required external libraries are installed. Furthermore, a docker container containing the required libraries is available, allowing the execution of the library within a container as well. Within EVEREST, both approaches are utilised.

3.4 Cryptographic Libraries for Data Protection

Efficient cryptographic primitives are needed to help protect communication to, from and within the FPGA. In EVEREST, we developed a number of cryptographic primitives providing encryption, authenticated encryption and authenticated encryption with associated data, and we provide them in form of a library of hardware components that can be instantiated and used to secure cloud FPGAs. The library includes the necessary RTL source files for all selected the encryption (block and stream ciphers) and authenticated encryption primitives we identified being relevant for the EVEREST SDK. The RTL is implemented in the VHDL hardware coding language and can be ported in to any single FPGA device or cluster easily. In the remaining part of this section, we introduce the cryptographic primitives that we implemented and we report the performance obtained synthesizing them on the target FPGAs.

Block ciphers and stream ciphers cater to the cryptographic operation of encryption. Whereas encryption deals with message confidentiality, it does not address message integrity, i.e. the situation when an adversary can tamper with the ciphertext, which would potentially result in the receiver getting an incorrect plaintext. A message authentication code (MAC) [45], often called tag, is a short piece of information used to authenticate a message, confirming that the message came from the stated sender (its authenticity) and that it has not been changed. The MAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. Authenticated Encryption (AE) [45] or Authenticated Encryption with Associated Data (AEAD) [45] is a form of encryption which simultaneously provides confidentiality, integrity, and authenticity assurances of the data.

The large body of research addressing efficient implementation of cryptographic algorithms on standalone FPGAs served us as based for developing our library. The starting point for the library development, was thus to study and understand the implementation strategies for various cryptographic algorithms on FPGA. As a typical FPGA device can accommodate a huge amount of logic gates, the metric we target is mostly throughput.

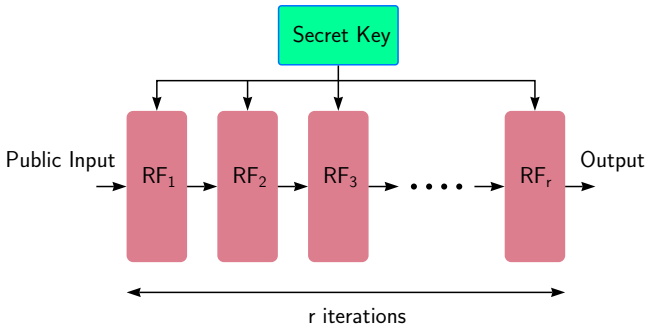


Figure 20 – Commonly, block or stream cipher consists of repeated application of a publicly known round function.

We started focusing on the AES algorithm. The AES-128 block cipher [22] is the the de-facto encryption standard worldwide, having been recommended by the United States' National Institute of Standards and Technology, in 2001. Since the design of AES-128 was finalized, many block ciphers with lightweight properties have been proposed. Among them, PRESENT [17] is well-studied with respect to its security and implementation. The cipher has been standardized in ISO/IEC 29192 "Lightweight Cryptography" process. While the above ciphers have mostly targeted optimization of hardware area, there have been other block ciphers aimed at optimizing other lightweight design metrics. The block cipher Prince [18], was designed for low latency (defined as the total delay incurred in computing an operation) based applications like memory encryption, while Midori [7] had been designed targeting energy optimization.

3.5 Architectures

Both block and stream ciphers consists of similar transformations which are applied repetitively on some public and private input to produce the output stream, see Figure 20. In the case of block ciphers specifically, the public input is the plaintext, the private input is the secret key and the output is obviously the encrypted plaintext also referred to as the ciphertext. As a result we can classify the flavours of block cipher implementation on hardware in the following four categories:

- **Round based circuits**: These are designs in which each round function is executed in one clock cycle. The architecture includes the circuitry required to execute the function, proceeded by a register on which the intermediate outputs of the round function computation are written on to. If the specification of the block or stream cipher requires R executions of the round function, then a round based implementation would require exactly R clock cycles to execute the encryption operation.
- **Multiple round-unrolled circuits**: this architecture extends the previous: instead of one it uses $r < R$ round function units connected serially. Since architecture computes r round function operations sequentially, they require only $\lceil \frac{R}{r} \rceil$ clock cycles. Because of the higher hardware footprint, such circuits consume more power but take less number of clock cycles to execute the encryption operation. Some circuits, e.g. for $r = 2$, are known to be energy-optimal for some specific block ciphers, especially on ASIC [8].
- **Fully round-unrolled circuits**: This takes unrolling to the extreme level, i.e. $r = R$ so that only a single clock cycle is required to execute encryption.
- **Serialized circuits**: The idea in these circuits is to reduce hardware footprint by employing increasingly lower number of logic gates, i.e. a fraction of the entire round function circuit. As a result the circuit takes multiple clock cycles to compute one round function. For example, the AES-128 circuit in [41] has only one S-box circuit whereas the AES round function requires 16 S-boxes. The circuit takes 160 cycles just to execute a single round function.

1

3.5.1 Cryptographic Primitives: Block Ciphers

In this subsection we compare the performances of 5 different block ciphers when implemented on FPGA platforms. We have considered block ciphers with both algebraically simple and complicated round functions.

We include in our exploration the AES algorithm, mostly as reference point, and a number of lightweight algorithms, that appear to be more suitable for providing encryption at a finer granularity without incurring in a high performance overhead. In the target platform for the EVEREST SDK, like cloud FPGAs, where hardware area is not as critical as in extremely constrained devices, one of the parameters that is the primary target of optimization is throughput. For lightweight ciphers, the architecture most suited for this goal is generally the fully unrolled one, so our designs are implemented following that design strategy. In the following part of this section we highlight the characteristics of the ciphers evaluated.

- **AES 128:** The Advanced Encryption Standard [22] has a simple Substitution Permutation Network (SPN) type round function which supports 128-bit plaintexts and 128-bit, 192-bit or 256-bit keys. The non linear layer of the AES algorithm consists of 16 applications of an S-box function in $\{0,1\}^8 \rightarrow \{0,1\}^8$. The permutation layer consists of ShiftRows and MixColumnoperations. Since the block cipher state can be interpreted as a 4×4 array of bytes, the ShiftRows operation simultaneously rotatesthe i-th row of the state by i bytes. The MixColumn operationmultiplies each column of the state by an MDS matrix over$GF(2^8)$. This is followed by an AddRoundkey operation where a 128-bit RoundKey, derived at each round from the initial secret key, is Xored to the state.
- **Present:** Present [17] is a 64-bit block cipher which has an SPN type round function. It has been adopted as a standard in ISO/IEC 29192-2. The cipher specifications allow for both 80-bit and 128-bit Key (we focused only focus on the 80-bit). The only non-linear component in the round function is the 4-bit S-box (i.e. over $\{0,1\}^4 \rightarrow \{0,1\}^4$), which is applied in parallel to each of the sixteen nibbles of the 64-bit state after the RoundKey addition. The state bits are then rearranged by a permutation layer.
- **Prince:** Prince [18] is a 64-bit block cipher with an SPN type round function. It allows for a 128-bit key but does not use any KeyScheduling logic. Prince is based on the FX construction: The 128-bit Key is divided into the most and least significant 8-byte blocks k_0, k_1 and a key k' is computed from them by a simple rotate and add operation. k_0 and k' are used as whitening keys and k_1 is used as the RoundKey in every round. The cipher uses three types of Round functions: Forward, Middle and Inverse. The Forward round consists of a SubBytes, MixColumn and addition of a Round constant and RoundKey. The Middle round consists of a SubBytes, MixColumn and Inverse SubBytes layer. The Inverse rounds are structurally and functionally the opposite of the Forward round. The main reason the cipher was designed was to minimize the latency of the encryption, making it very suitable for memory encryption.
- **Midori:** Midori [7] is an SPN based block cipher designed specifically to be energy efficient, and it is still the block cipher solution that consumes the least amoung of energy. The specifications support both 64-bit and 128-bit plaintexts and 128-bit keys. Since the 64-bit version has been the subject of Invariant Subspace attacks [32], while the 128-bit version is still secure, we consider only Midori-128.
- **Gift-128:** Gift [11] was proposed as a redesign of the popular PRESENT block cipher. The idea was to design a cipher that would be efficient on both hardware and software platforms and yet offer a high degree of security. The cipher is of SPN type and like PRESENT uses a bit permutation as the linear layer, so as to minimize the hardware footprint. The design supports both 64 and 128-bit plaintexts, and we focus on the 128-bit version here.

Table 12 reports the experimental results obtained using xc7a200t Xilinx device from the Artix7 family and the Alveo U280. The design was synthesized, mapped, placed and routed using the Xilinx Vivado design suite 2021.2.

3.5.2 Cryptographic Primitives: Stream Ciphers

A Stream cipher is an algorithm that takes a Secret Key K as input, usually a binary string of around $80-256$ bits, and applies a number of transformations to produce a long pseudorandom sequence known as the keystream. This sequence is usually XORed with each bit of the plaintext to produce the encrypted ciphertext. So, if $P = p_0, p_1, p_2, \ldots$ represents the bits, bytes, or words of the plaintext, and $\kappa = k_0, k_1, k_2, \ldots$ represents the

Table 12 – The synthesis of block ciphers.

Device	Design	# LUTs	# Slices/CLBs	Latency (ns)	$f^{\dagger}_{max}$ (MHz)	TP_{max} (Gbps)
Artix 7	AES-128	11977	3387	129.341	7.7	0.92
	Present	2222	801	66.222	15.1	0.90
	Prince	1263	505	45.631	21.9	1.31
	Midori-128	3840	1454	62.939	15.9	1.89
	Gift-128	3836	1303	66.193	15.1	1.80
Alveo U280	AES-128	17091	3049	43.333	23.1	2.75
	Present	2092	291	25.248	39.6	2.36
	Prince	1320	216	20.097	49.8	2.97
	Midori-128	4976	858	35.026	28.6	3.40
	Gift-128	2560	458	29.888	33.5	3.99

† : Note that f_{max} is generated from the Post-PAR simulation.

Table 13 – The eStream Portfolio.

Profie 1 (HW)	Profile 2 (SW)
Grain v1 [35]	Salsa20 [15]
MICKEY 2.0 [6]	Sosemanuk [13]
Trivium [19]	HC128 [52]
	Rabbit [16]

keystream bits, bytes, or words produced by the Stream Cipher using the Secret Key K, then the encryption rule is given by:

$$c_i = p_i \oplus k_i \ , \ \forall i,$$

where $C = c_1, c_2, \ldots$ represents the ciphertext bits, bytes or words. Since the Secret Key is already known to the receiver, he can compute the keystream bits $k_0, k_1, \ldots$ at his end, which are then used to decrypt the ciphertext as follows:

$$p_i = c_i \oplus k_i \ , \ \forall i.$$

Stream ciphers can be viewed as approximating the action of a proven unbreakable cipher, the one-time pad (OTP). Stream ciphers have been largely studied during the eSTREAM project [1], which had the goal to design new stream ciphers suitable for widespread adoption. The call for primitives was first issued in November 2004. The project was completed in April 2008. The project was divided into separate phases and the project goal was to find algorithms suitable for different application profiles.

The eSTREAM portfolio ciphers fall into two profiles. Profile 1 stream ciphers are particularly suitable for hardware applications with restricted resources such as limited storage, gate count, or power consumption. Profile 2 contains stream ciphers more suitable for software applications with high throughput requirements. The portfolio currently contains the algorithms reported in Table 13. Among them, we concentrated on the following cihphers since they have been widely studied in literature:

- **Trivium:** Trivium[23] is a stream cipher designed for the eSTREAM project by DeCannière and Preneel and is currently an ISO standard under ISO/IEC 29192-3:2012. Trvium has an internal state of 288 bits which is divided into 3 registers of sizes 93, 84 and 111 bits respectively, see Figure 21. The stream cipher uses an 80-bit key and 80-bit Initialization Vector (IV) which is used to initialize the state. The setup is updated for 1024 iterations using a very simple to implement update function shown partially in Figure 21.

- **Grain 128:** The Grain family of stream ciphers. It consists of three ciphers: Grain v1, Grain 128, Grain 128a. In this chapter we focus on Grain 128 [34] which offers 128 bit security. Like the other members of the Grain family, Grain 128 has a connected register structure as shown in Figure 22. Grain-128 consists of a 128-bit Linear-Feedback Shift Register (LFSR) and a 128-bit Nonlinear-Feedback Shift Register (NFSR), and uses an 128-bit key K. Given that $L_t = [l_t, l_{t+1}, \ldots, l_{t+127}]$ is the LFSR state at the t-th clock

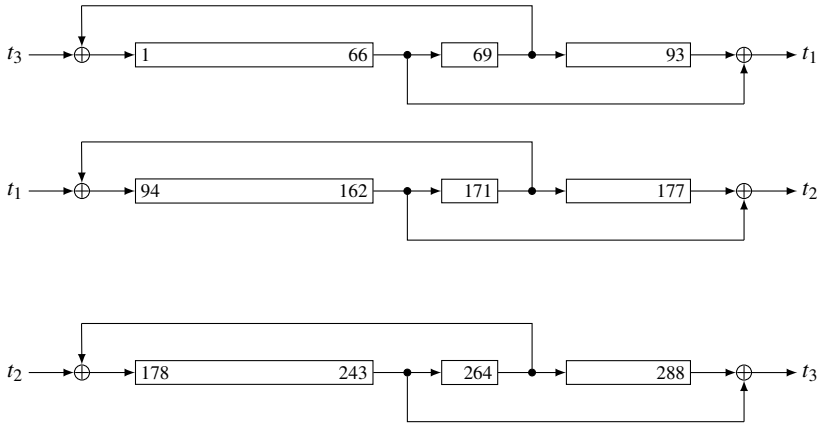


Figure 21 – Structure of Trivium. The AND gates $s_{91} \cdot s_{92}$, $s_{175} \cdot s_{176}$, $s_{286} \cdot s_{287}$ are added to the leftmost XOR gates before the 2nd, 3rd and 1st registers respectively and have been omitted for ease of depiction. The keystream bit produced every clock cycle is given as $z = t_1 + t_2 + t_3$.

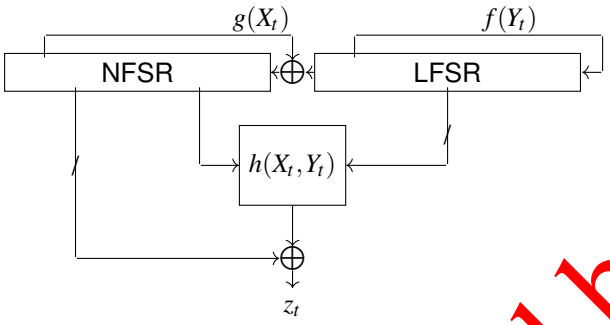


Figure 22 – Structure of Stream Cipher in Grain amily

interval, Grain-128's LFSR is defined by the update function f given by:

$$f(Y_t) = l_{t+96} + l_{t+81} + l_{t+70} + l_{t+38} + l_{t+7} + l_t.$$

The NFSR state is updated as $n_{t+128} = l_t + g(\cdot)$ for NFSR update function g, which is given by:

$$g(X_t) = n_{t+96} + n_{t+91} + n_{t+56} + n_{t+26} + n_t + n_{t+3}n_{t+67} + n_{t+11}n_{t+13} +$$
$$n_{t+17}n_{t+18} + n_{t+27}n_{t+59} + n_{t+40}n_{t+48} + n_{t+61}n_{t+65} + n_{t+68}n_{t+84}.$$

The output function is of the form

$$z_t = h'(X_t, Y_t) = \bigoplus_{a \in A} n_{t+a} + h(s_0, \ldots, s_8) + l_{93},$$

where $A = \{2, 15, 36, 45, 64, 73, 89\}$, $h(s_0, \ldots, s_8) = s_0s_1 + s_2s_3 + s_4s_5 + s_6s_7 + s_0s_4s_8$, and $(s_0, \ldots, s_8) = (n_{t+12}, l_{t+8}, l_{t+13}, l_{t+20}, n_{t+95}, l_{t+42}, l_{t+60}, l_{t+79}, l_{t+95})$. The cipher is initialized with a 128-bit key and a 96-bit IV. 256 clocks of initialization are executed before entering the keystream phase.

Also in this case, our target platform was the xc7a200t Xilinx device from the Artix7 family. The results are reported in Table 14 Since stream ciphers, once initialized, continuously produces keystream every clock cycle, we can experiment with different number of unrolled rounds r for each of the stream ciphers. The higher the value of r, the higher is the device utilization, but it also ensures higher throughput.

1

3.5.3 Cryptographic Primitives: Authenticated Encryption

Authenticated Encryption (AE) or Authenticated Encryption with Associated Data (AEAD) is a form of encryption which simultaneously provides confidentiality, integrity, and authenticity assurances on the data [45]. The

Table 14 – The synthesis reports for Stream Ciphers.

Device	Design	r	# LUTs	# Slices/CLBs	#FFs	Latency (ns)	$f^{\dagger}_{max}$ (MHz)	TP_{max} (Gbps)
Artix 7	Trivium	36	364	110	297	2.656	376.5	12.62
		72	462	168	296	2.740	365.0	24.47
		144	786	344	295	3.687	271.2	36.37
		288	1481	597	291	6.398	156.3	41.92
	Grain-128	32	551	245	260	4.109	243.4	7.25
		64	986	418	260	5.817	171.9	10.25
		128	1815	782	310	8.820	113.4	13.52
		256	4216	1550	401	15.731	63.6	15.52
Alveo U280	Trivium	36	258	43	297	6.121	163.4	5.48
		72	401	67	296	5.659	176.7	11.85
		144	746	121	295	7.996	125.1	16.77
		288	1275	184	294	10.470	95.5	25.62
	Grain-128	32	471	79	260	5.730	174.5	5.20
		64	983	149	259	7.715	129.6	7.73
		128	1747	253	258	14.843	67.4	8.03
		256	4178	649	258	18.669	53.6	12.77

† : Note that f_{max} is generated from the Post-PAR simulation.

need for AE emerged from the observation that securely combining a confidentiality mode with an authentication mode could be error prone and difficult. This was confirmed by a number of practical attacks introduced into protocols and applications by incorrect implementation, or lack, of authentication (including SSL/TLS) [14, 51, 4]. A typical programming interface for AE mode implementation would provide the following functions: a) Encryption that takes as input plaintext, key, and optionally a header that will not be encrypted, but will be covered by integrity protection. It produces as output a ciphertext and authentication tag (MAC), and b) Decryption that takes as input ciphertext, key, authentication tag, and optionally a header and outputs a plaintext, or an error if the authentication tag does not match the supplied ciphertext or header.

The diffusion of low-resource devices and their security requirements spurred the NIST Lightweight Cryptography competition [2], that started in 2018 and completed 5 years after announcing ASCON 128[26] as the winner. Since the selection process have been concludes only recently, we report a comparison between ASCON 128, the selected algorithm, and two schemes that moved to the second round of competition. We selected GIFT-COFB and ROMULUS as comparison since they are bootstrapped either directly via lightweight block ciphers or variants of them. More precisely, they are directly instantiated with the Gift block cipher [11] or Skinny block cipher [12]. ASCON 128, GIFT-COFB and ROMULUS are further compared with AES-GCM. These fours schemes are likely to be very important for the foreseeable future. We summarize their characteristics in the next part of this section.

- **AES-GCM** Galois/Counter Mode (GCM) [27] is an authenticated encryption algorithm designed to provide both data authenticity (integrity) and confidentiality. GCM is defined for block ciphers with a block size of 128 bits. Galois Message Authentication Code (GMAC) is an authentication-only variant of the GCM which can form an incremental message authentication code. GCM is proven secure in the concrete security model. It is secure when it is used with a block cipher that is indistinguishable from a random permutation; however, security depends on choosing a unique initialization vector for every encryption performed with the same key (see stream cipher attack). For any given key and initialization vector combination, GCM is limited to encrypting $2^{39} - 256$ bits of plain text (64 GiB).

  GCM combines the well-known counter mode of encryption with the new Galois mode of authentication [27]. The key-feature is the ease of parallel computation of the Galois field multiplication used for authentication. This feature permits higher throughput than encryption algorithms, like CBC, which use chaining modes. T

  The authentication tag is constructed by feeding blocks of data into the GHASH function and encrypting

the result. This GHASH function is defined by:

$$GHASH(H,A,C) = X_{m+n+1},$$

where $H = E_K(0^{128})$ is the Hash Key, a string of 128 zero bits encrypted using the block cipher, A is data which is only authenticated (not encrypted), C is the ciphertext, m is the number of 128-bit blocks in A (rounded up), n is the number of 128-bit blocks in C (rounded up), and the variable X_i for $i = 0,\ldots,m+n+1$ is defined below.

First, the authenticated text and the cipher text are separately zero-padded to multiples of 128 bits and combined into a single message S_i:

$$S_i = \begin{cases} A_i & \text{for } i = 1,\ldots,m-1 \\ A_m^* \parallel 0^{128-v} & \text{for } i = m \\ C_{i-m} & \text{for } i = m+1,\ldots,m+n-1 \\ C_n^* \parallel 0^{128-u} & \text{for } i = m+n \\ \text{len}(A) \parallel \text{len}(C) & \text{for } i = m+n+1 \end{cases}$$

where $len(A)$ and $len(C)$ are the 64-bit representations of the bit lengths of A and C, respectively, $v = len(A) \bmod 128$ is the bit length of the final block of A, $u = len(C) \bmod 128$ is the bit length of the final block of C, and $\parallel$ denotes concatenation of bit strings. Then X_i is defined as:

$$X_i = \sum_{j=1}^{i} S_j \cdot H^{i-j+1} = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus S_i) \cdot H & \text{otherwise} \end{cases}$$

The second form is an efficient iterative algorithm (each X_i depends on X_{i-1}) produced by applying Horner's method to the first. Only the final X_{m+n+1} remains an output.

The most critical operation in GCM is multiplication in the finite field $GF(2^{128})$. The multiplier uses the irreducible polynomial $p(x) = x^{128}+x^7+x^2+x+1$ to compute $C = AB \bmod p(x)$. In [48], several implementation options for such a multiplier are proposed, including bit-parallel, digit-serial and hybrid multipliers. Bit-parallel multipliers use multiplication by x as the fundamental circuit of computation and replicate it 128 times for the complete operation. Digit serial multipliers take this idea forward by making multiplication by x^m as the basic unit. Hybrid multipliers redefine the original finite field $GF(2^k)$ as $GF((2^m)^n)$ where $k = mn$. Arithmetic calculations can then be performed using circuits in the subfield $GF(2^m)$ and combining them in the extension field in the extension field $GF((2^m)^n)$.

All the above architectures take more than one clock cycles to compute the result of multiplication. Since our core encryption algorithm will operate in a single clock cycle, we propose an architecture that will compute the multiplication also in a single cycle. Let $A(x), B(x)$ be two polynomials of degree $2k-1$. The Karatsuba method of multiplying them requires the following. We first split both the polynomials into 2 degree $k-1$ polynomials as follows:

$$A(x) = a_L(x) + x^k a_H(x),\ B(x) = b_L(x) + x^k b_H(x),$$

The multiplication operation requires the following logic operations over k-bit polynomials:

1. Compute $S = (a_L \oplus a_H) \cdot (b_L \oplus b_H)$.
2. Compute $L = a_L \cdot b_L$ and $H = a_H \cdot b_H$.
3. Compute $M = S \oplus L \oplus H$.

It is easy to see that $A(x) \cdot B(x) = x^{2k}H \oplus x^k M \oplus L$. Thus the original $2k$ bit multiplier requires 3 k-bit multipliers plus some gates performing linear operations. Thus one can recursively define multiplication over 128-bit polynomials as multiplication over 64-bit polynomials which in turn can be defined as multiplications over 32-bit polynomials and so on. The base case is defining multiplication over 2-bit (i.e. degree 1) polynomials. This can be constructed easily by defining a look up table $\{0,1\}^4 \to \{0,1\}^4$, i.e. that takes

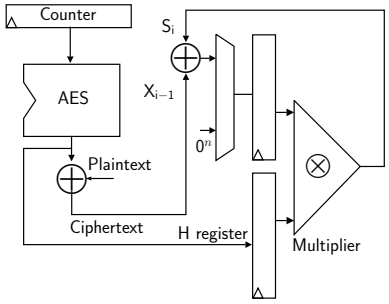


Figure 23 – AES-GCM circuit.

Table 15 – The synthesis reports for AES-128 GCM.

Device	Design		# LUTs	# FFs	# Slices	Latency	$f^{\dagger}_{max}$	TP_{max}
	S-box	Mixcolumns				(ns)	(MHz)	(Gbps)
Artix 7	Small	Tiny	22955	308	8775	168.092	5.95	0.71
	Tradeoff	Tiny	29687	301	10942	178.925	5.59	0.67
	LUT	Tiny	14626	304	5034	73.893	13.53	1.61
	Small	Fast	23794	300	9831	163.204	6.13	0.73
	Tradeoff	Fast	23945	302	9694	166.131	6.02	0.72
	LUT	Fast	14624	302	4788	74.203	13.48	1.61
	T-Table		20204	300	6614	87.922	11.37	1.36
Alveo U280	Small	Tiny	22922	294	3808	43.193	23.15	2.76
	Tradeoff	Tiny	23207	294	3913	40.895	24.45	2.92
	LUT	Tiny	14005	294	2473	32.365	30.90	3.68
	Small	Fast	22921	294	3906	43.174	23.19	2.77
	Tradeoff	Fast	23214	294	3990	41.203	24.27	2.89
	LUT	Fast	14027	294	2447	33.283	30.04	3.58
	T-Table		21335	294	3774	44.187	22.63	2.70

† : Note that f_{max} is generated from the Post-PAR simulation.

the 4 bit coefficients of the two 2-bit polynomials and produces the 4 bit coefficients of the product. The result of the above logic circuit is a polynomial of degree 254, i.e. 255 bit-coefficients. We now need to perform the modulo $p(x)$ operation to reduce it to 128 coefficients. However this is a purely linear operation and needs only a few XOR gates depending on the structure of $p(x)$.

We experiment with a number of different architectures for the components of the AES block cipher. We experiment with four different architectures of the S-box, i.e. Small, Tradeoff, and LUT. These three were proposed in [44]: "Small" refers to the smallest S-box circuit existing in literature, whereas "Tradeof" is the circuit that provides a balance between latency and circuit area."LUT" refers to a simple look-up-table style of implementation which the synthesizer optimizes. We further explore two different styles: Fast and Tiny of implementation of the Mixcolumns circuit. "Tiny" refers to the smallest implementation of the circuit (92 gates) proposed in [43]. "Fast" refers to the 103 gate implementation in [10] which, despite larger, it is instead characterized by the shortest gate depth. We further compare these implementations with T-Table based implementations which are known to be extremely fast on FPGA platforms [31].

The AES-GCM circuit is depicted in Figure 23. In the 1st clock cycle the hash key $H = E_K(0)$ is computed and stored in the Hash register. Thereafter every 128-bit block of plaintext and associated data is processed in one block to produce ciphertext. Simultaneously, the MAC is computed using a Horner-like computation using the H and an auxiliary register using the single cycle finite field multiplier. Thus processing n blocks of data takes only $n+1$ clock cycles.

The following are the results obtained after the designs were synthesized, mapped, placed and routed on the xc7a200t Xilinx device from the Artix7 family and on the Alveo U280 platform.

The results show that for table based architectures, the total throughput of around 4 Gbps can be reached on the Alveo U280 platform. It is well known that the GCM algorithm may be further parallelized by a factor of k by using a proportionally multiple amount of circuit resources. This allows for reaching speeds well above 100 Gbps depending on the type of application.

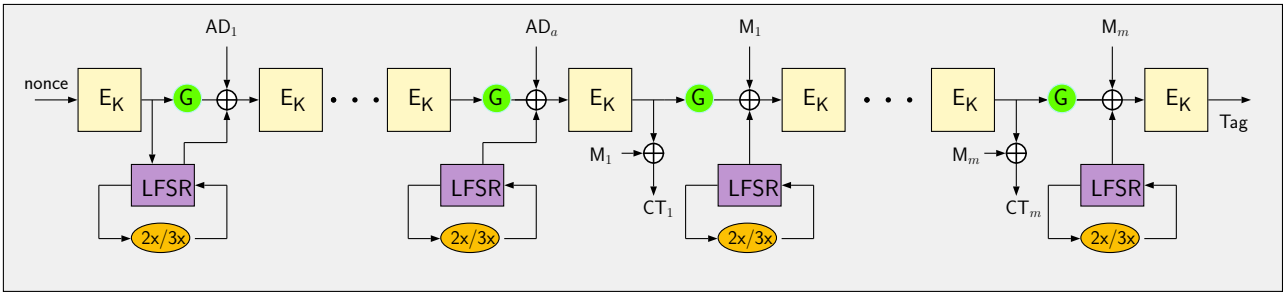


Figure 24 – GIFT-COFB mode of operation.

- **GIFT-COFB** GIFT-COFB [9] is a lightweight AEAD candidate and a submission to the recently closed NIST lightweight cryptography standardization process. The algorithm reached the final round of the competition. The construction processes 128-bit blocks with a key and nonce of the same size and has a small register footprint, only requiring a single additional 64-bit register. Besides the block cipher, the mode of operation deploys a bit permutation and a finite field multiplication with different constants. Note that unlike GCM, multiplication in GIFT-COFB occurs by only a few constant field elements. As such this circuit is completely linear and can be efficiently implemented in hardware using simple XOR gates.

  Mathematically, GIFT-COFB is a block-cipher-based authenticated encryption mode that incorporates GIFT-128 as the underlying block cipher with an 128-bit key and state. The construction adheres to the *COmbined FeedBack* (COFB) mode of operation [20] which provides a processing rate of 1, i.e., a single block cipher invocation per input data block. The mode only adds an additional 64-bit LFSR state L (initialized as the first 64 Most Significant Bits (MSBs) of $E_K(Nonce)$) to the existing block cipher registers and thus ranks among the most lightweight AEAD algorithms in the literature.

  In this mode, encryption interspersed by 3 operations: execute the operation G on the state, Update the LFSR L, Add plaintext/associated data to state as shown in Figure 24. The G operation is given by $G(X_0, X_1) = (X_1, X_0 \lll 1)$, where X_0, X_1 are the upper and lower 64 bit blocks of a a 128 bit word. The register L is initialized with the first 64 MSBs of $E_K(Nonce)$ and updated by finite field multiplication over $GF(2^{64})$ by the constant $2^x 3^y$ where

$$x = \begin{cases} 1 & \text{if } |A| \bmod r = 0 \text{ and } A \neq \varepsilon, \\ 2 & \text{otherwise.} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } |M| \bmod n = 0 \text{ and } M \neq \varepsilon, \\ 2 & \text{otherwise.} \end{cases}$$

  The GIFT-128 Block cipher [11] has 40 rounds in which each round consists of a substitution layer composed of 4-bit S-boxes. It uses a bit permutation over 128-bits as the linear layer. Initially designed keeping in mind software efficiency, it is more or less efficient in both software and hardware platforms.

  The block diagram of the implementation is depicted in Figure 25. In the 1st clock cycle the LFSR L is updated with the top half of $E_K(Nonce)$. Thereafter every 128-bit block of plaintext/associated data is processed in one block to produce ciphertext. Simultaneously, the LFSR is updated using finite field computations. After all the plaintext and associated data (AD) have been processed, the mode uses one additional encryption call to produce the MAC. Thus processing n blocks of data takes only $n+2$ clock cycles.

- **ROMULUS**

  ROMULUS is an AEAD scheme designed by Iwata et al. [40], and uses the SKINNY family of block ciphers. In this work, we provide implementations for Romulus-N1. This the primary candidate of the family that employs SKINNY-128-384 tweakable block cipher.

  Romulus-N1 makes $1/2$ block cipher call per associated data block, and 1 block cipher call per message block. It admits 128-bit key, 128-bit nonce, variable-length message chopped into 128-bit blocks, and

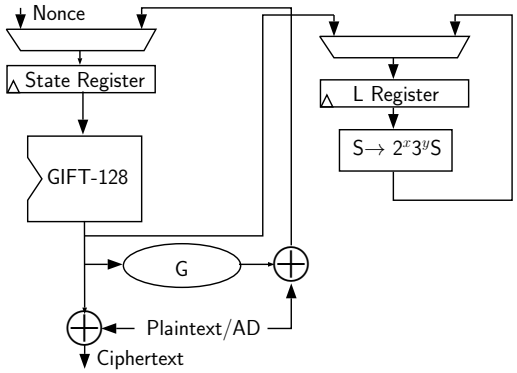


Figure 25 – GIFT-COFB Circuit.

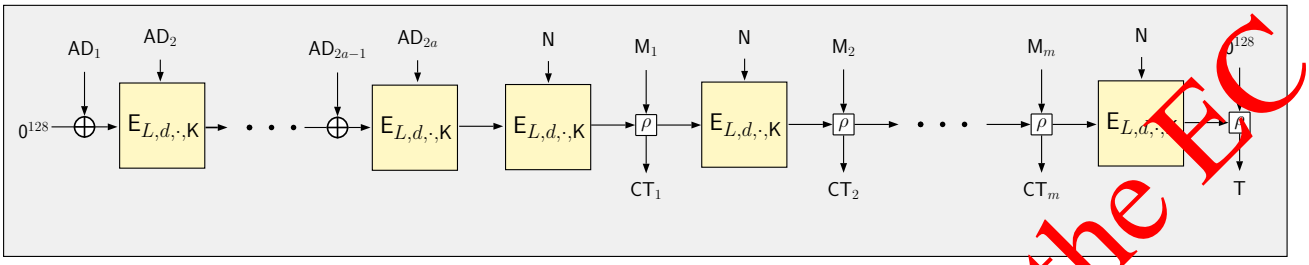


Figure 26 – The high-level view of Romulus-N1, which depicts the processing of $2a$ associated data and m message blocks. L denotes the 56-bit LFSR that counts the number of processed blocks, and d denotes a single byte domain separator followed by 0^{64}.

produces 128-bit tag. in the sense that each output of the block cipher and the incoming data block (associated data or message) are together passed through a light combinatorial function denoted by ρ. $\rho(S,M) = (S',C)$ is defined as $S' \leftarrow S \oplus M$ and $C \leftarrow G(S) \oplus M$. For each byte, G performs the following operation: $G(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := (x_0 \oplus x_7)||x_7||x_6||x_5||x_4||x_3||x_2||x_1$. The output of this function is immediate input to the next block cipher call. Hence a register keeps this *running state*, and at the last step it is encrypted to produce the tag.

Romulus handles odd and even authenticated data blocks differently; the odd blocks are input to ρ, and even blocks are fed to the nonce port of the block cipher, as the underlying cipher SKINNY-128-384 has a 384-bit long tweakey. The actual AEAD nonce is not used before all authenticated data blocks are processed, and later used as block cipher nonce while message blocks are encrypted. A 56-bit LFSR is also a part of the tweakey for SKINNY calls and keeps the count of authenticated data and message block fed to the AEAD circuit since the beginning of the AE operation.

Figure 26 describes the three phases a full AEAD operation passes through, namely processing of (1) associated data, (2) nonce and (3) message blocks.

Figure 27 depicts the ROMULUS-N1 Architecture. Other than the tweakable block cipher we have the LFSR L which supplies a part of the tweak. After all the plaintext/AD have been processed, the mode uses one additional encryption call to produce the MAC. Thus processing n blocks of data takes only $n+1$ clock cycles.

- **ASCON 128** ASCON-128 has been declared the winner of the NIST lightweight cryptography competition It is a permutation based AEAD, as in the core cryptographic primitive used in the design is a permutation function over 320 bits and not a block cipher. The ASCON permutation a state size of 320 bits (consisting of five 64-bit words x_0, x_1, x_2, x_3, x_4) that are updated in four phases: Initialization, Processing of Associated Data, Processing of Plaintext/Ciphertext, and Finalization.

  All phases use the same permutation function p that is applied 12 times in the Initialization and Finalization phase and 6 times in the data processing phase. The data i.e. both the plaintext and AD is handled in 64-bit blocks. The Initialization phase takes the IV, Key and Nonce and runs the ASCON permutation function 12 times on it, followed by xor with the key. After the Initialization phase the optional associated data is processed. In the Encryption phase, each plaintext block P_i is XORed with the secret state to produce one ciphertext block C_i. The Finalization process xors the key K again to the state and extracts the tag T for authentication.

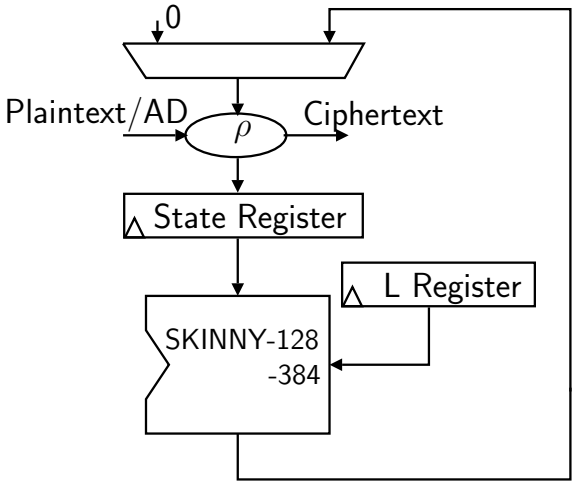


Figure 27 – ROMULUS-N1 circuit.

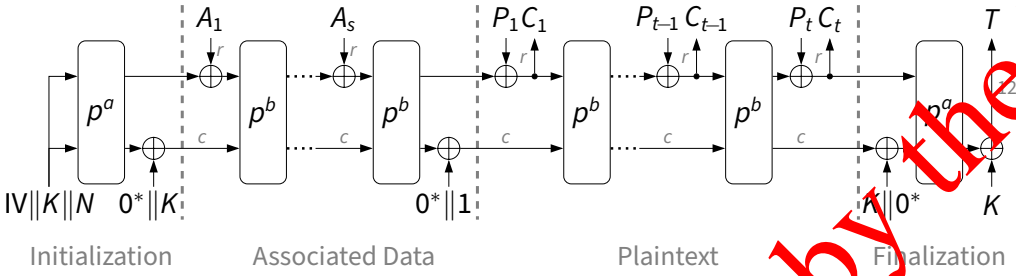


Figure 28 – The figure depicts the processing followed in ASCON 128 encryption cycle.

The architecture of ASCON 128 is reported in Figure 28. The core circuit is the ASCON permutation p^6 i.e. the round function p iterated 6 times. Hence initialization and finalization takes 2 cycles each. Processing each 64-bit block of Plaintext or AD takes 1 cycle only. Thus processing n blocks of 128-bit data takes only $2n+2+2=2n+4$ clock cycles.

Table 16 compares synthesis results for the 2 lightweight schemes with AES-GCM. It is possible to see seen that ASCON 128 and GIFT-COFB have a massive advantage in terms of throughput over the lightweight schemes. In particular the most latency intensive part of the circuit is the permutation circuit that require only 6 rounds to be implemented consecutively. This is one of the reasons that reduces the latency and elevates the maximum operable frequency and hence throughput.

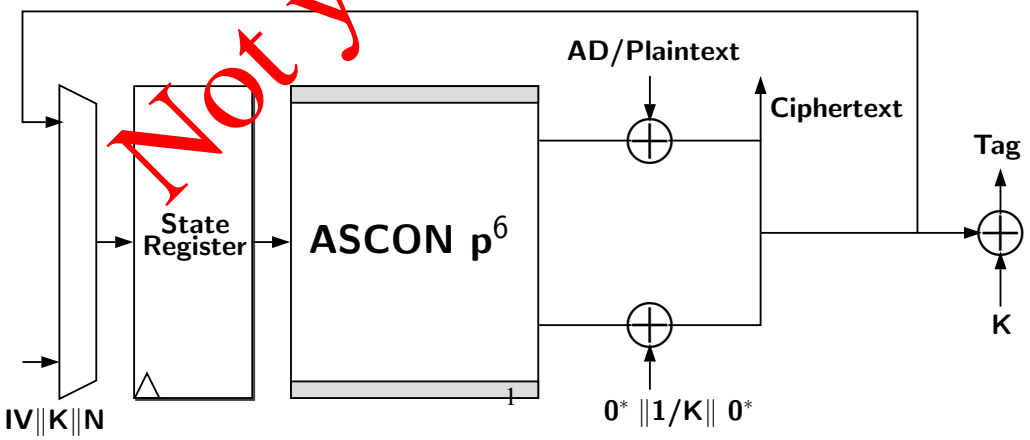


Figure 29 – ASCON 128 circuit.

Table 16 – The synthesis reports.

Device	Design	# LUTs	# FFs	# Slices/CLBs	Latency (ns)	$f_{max}^{\dagger}$ (MHz)	TP_{max} (Gbps)
Artix-7	GIFT-COFB	5931	205	1791	44.422	22.5	2.68
	Romulus-N1	45953	454	15342	180.206	5.5	0.66
	AES-GCM	20204	300	6614	87.922	11.4	1.36
	ASCON 128	2644	327	708	22.133	45.2	2.69
Alveo U280	GIFT-COFB	4008	196	647	31.496	31.75	3.78
	Romulus-N1	9512	191	1546	104.865	9.53	1.14
	AES-GCM	14005	294	2473	32.365	30.90	3.68
	ASCON 128	2736	326	429	15.802	63.3	3.77

† : Note that f_{max} is generated from the Post-PAR simulation.

4 Conclusion

This deliverable refines, extends, and updates the content of Deliverable D3.1 and reports the final version of the DMTs developed in WP3 throughout the duration of the EVEREST project. In addition to the results achieved at M18 of the project (notably the development of the EVEREST data lifetime and the DMTs in the Xilinx Alveo accelerators that are reported in this deliverable as they where presented in Deliverable D3.1) this deliverable reports the activities carried out in the second part of the project. In addition to an update of the Data Management Architecture, we have recently designed a mechanism for graceful detachment of FPGA kernels; we have completed with new analysis and results the DMTs for custom data types. We have also enhanced the data protection support in EVEREST by adding additional algorithms to the anomaly detection library and by providing a library of cryptographic primitives. Finally, we developed the final version of the support for storage and communication at cluster level.

In conclusion, DMTs are a crucial component of any data driven application. When developing the DMTs within the EVEREST project, we considered a wide spectrum of applications and also considered future use and development of the EVEREST SDK beyond the completion of the project. We believe that the data management techniques proposed and developed are fulfilling this goal.

Acronyms

AXI Advanced eXtensible Interface. 10, 12, 14, 15

BRAM Block Random Access Memory. 9, 11, 12

CU Compute Units. 11

DDR Double Data Rate. 14–16

DMA Direct Memory Access. 10, 11, 16, 17

DMT Data Management Technique. 13, 15–17

DMTs Data Management Techniques. 6, 7, 9, 41

HBM High Bandwidth Memory. 11

References

[1] eSTREAM, the ECRYPT stream cipher project. eSTREAM, The ECRYPT Stream Cipher Project, 2012. https://www.ecrypt.eu.org/stream/.

[2] Nist lightweight cryptography project, 2019. Available at https://csrc.nist.gov/projects/lightweight-cryptography.

[3] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised real-time anomaly detection for streaming data. Neurocomputing, 262:134–147, 2017. Online Real-Time Learning Strategies for Data Streams.

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013, pages 526–540. IEEE Computer Society, 2013.

[5] Apache airflow™. https://airflow.apache.org. [Online; accessed 22-March-2024].

[6] Steve Babbage and Matthew Dodd. The stream cipher mickey 2.0. eSTREAM, ECRYPT Stream Cipher Project Report, 2005. http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf.

[7] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II, volume 9453 of Lecture Notes in Computer Science, pages 411–436. Springer, 2015.

[8] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Exploring energy efficiency of lightweight block ciphers. In Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers, pages 178–194, 2015.

[9] Subhadeep Banik, Avik Chakraborti, Akiko Inoue, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift-cofb. Cryptology ePrint Archive, 2020.

[10] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. Further results on efficient implementations of block cipher linear layers. IEICE Trans. Fundam. Electron. Commun. Comput. Sci., 104-A(1):213–225, 2021.

[11] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, pages 321–345, 2017.

[12] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, pages 123–153, 2016.

[13] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. Sosemanuk, a fast software-oriented stream cipher. eSTREAM, ECRYPT Stream Cipher Project Report, 2006. http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_p3.pdf.

[14] Daniel Bernstein. Failures of secret-key cryptography. Invited talk to FSE 2013. Available at https://cr.yp.to/talks/2013.03.12/slides.pdf.

[15] Daniel J. Bernstein. Salsa20/8 and salsa20/12. eSTREAM, ECRYPT Stream Cipher Project Report, 2006. http://www.ecrypt.eu.org/stream/papersdir/2006/007.pdf.

[16] Martin Boesgaard, Mette Vesterager, Thomas Christensen, and Erik Zenner. The stream cipher rabbit. eSTREAM, ECRYPT Stream Cipher Project Report, 2006. http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf.

[17] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings, volume 4727 of Lecture Notes in Computer Science, pages 450–466. Springer, 2007.

[18] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings, volume 7658 of Lecture Notes in Computer Science, pages 208–225. Springer, 2012.

[19] Christophe De Cannière and Bart Preneel. Trivium -specifications. eSTREAM, ECRYPT Stream Cipher Project Report, 2005. http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf.

[20] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based authenticated encryption: How small can we go? In Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, pages 277–298, 2017.

[21] Yuwei Cui, Subutai Ahmad, and Jeff Hawkins. The htm spatial pooler—a neocortical algorithm for online sparse distributed coding. Frontiers in Computational Neuroscience, 11, 2017.

[22] Joan Daemen and Vincent Rijmen. Rijndael/aes. In Henk C. A. van Tilborg, editor, Encyclopedia of Cryptography and Security. Springer, 2005.

[23] Christophe De Canniere and Bart Preneel. Trivium. In New Stream Cipher Designs: The eSTREAM Finalists, pages 244–266. Springer, 2008.

[24] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. Posits: The good, the bad and the ugly. In Proceedings of the Conference for Next Generation Arithmetic 2019, CoNGA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. IEEE Design & Test of Computers, 28(4):18–27, 2011.

[26] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. J. Cryptol., 34(3):33, 2021.

[27] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, National Institute of Standards and Technology, 2007.

[28] Xin Fang and Miriam Leeser. Open-source variable-precision floating-point library for major commercial fpgas. ACM Trans. Reconfigurable Technol. Syst., 9(3), jul 2016.

[29] Luc Forget, yohann Uguen, and Florent de Dinechin. Hardware cost evaluation of the posit number system. In Compas'2019 - Conférence d'informatique en Parallélisme, Architecture et Système, pages 1–7, Anglet, France, June 2019.

[30] Karl F. A. Friebel, Jiahong Bi, and Jeronimo Castrillon. BASE2: An IR for binary numeral types. In 13th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2023), HEART2023, pages 19–26, New York, NY, USA, June 2023. Association for Computing Machinery.

[31] Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In International workshop on cryptographic hardware and embedded systems, pages 33–48. Springer, 2011.

[32] Jian Guo, Jérémy Jean, Ivica Nikolić, Kexin Qiao, Yu Sasaki, and Siang Meng Sim. Invariant subspace attack against full midori64. Cryptology ePrint Archive, 2015.

[33] Jeff Hawkins and Subutai Ahmad. Why neurons have thousands of synapses, a theory of sequence memory in neocortex. Frontiers in Neural Circuits, 10, 2016.

[34] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A stream cipher proposal: Grain-128. eSTREAM, ECRYPT Stream Cipher Project Report, 2008. http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain128_p3.pdf.

[35] Martin Hell, Thomas Johansson, and Willi Meier. Grain - a stream cipher for constrained environments. eSTREAM, ECRYPT Stream Cipher Project Report, 2005. http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain_p3.pdf.

[36] HEAppE middleware. https://github.com/It4innovations/HEAppE. [Online; accessed 22-March-2024].

[37] Lexis platform. https://opencode.it4i.eu/lexis-platform. [Online; accessed 22-March-2024].

[38] Lexis platform documentation. https://docs.lexis.tech. [Online; accessed 22-March-2024].

[39] Py4lexis. https://opencode.it4i.eu/lexis-platform/clients/py4lexis. [Online; accessed 22-March-2024].

[40] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.2. NIST Lightweight Cryptography Project, 2019.

[41] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY. In Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, pages 687–707, 2017.

[42] Monica Lam. A Systolic Array Optimizing Compiler. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[43] Da Lin, Zejun Xiang, Xiangyong Zeng, and Shasha Zhang. A framework to optimize implementations of matrices. In Kenneth G. Paterson, editor, Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings, volume 12704 of Lecture Notes in Computer Science, pages 609–632. Springer, 2021.

[44] Alexander Maximov and Patrik Ekdahl. New circuit minimization techniques for smaller and faster AES sboxes. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2019(4):91–125, 2019.

[45] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. Handbook of applied cryptography. CRC press, 2018.

[46] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, and Fabrizio Ferrandi. Enabling the high level synthesis of data analytics accelerators. In Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Raul Murillo, Alberto A. Del Barrio, Guillermo Botella, and Christian Pilato. Generating posit-based accelerators with high-level synthesis. IEEE Transactions on Circuits and Systems I: Regular Papers, 70(10):4040–4052, 2023.

[48] Christof Paar. Implementation options for finite field arithmetic for elliptic curve cryptosystems. The 3rd workshop on Elliptic Curve Cryptography, (October 1999).

[49] Stephanie Soldavini, Donatella Sciuto, and Christian Pilato. Iris: Automatic generation of efficient data layouts for high bandwidth utilization. In Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC), page 172–177, 2023.

[50] David B. Thomas. Templatised soft floating-point for high-level synthesis. In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 227–235, 2019.

[51] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In Lars R. Knudsen, editor, Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings, volume 2332 of Lecture Notes in Computer Science, pages 534–546. Springer, 2002.

[52] Hongjun Wu. Hc-128. eSTREAM, ECRYPT Stream Cipher Project Report, 2006. http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf.