http://www.everest-h2020.eu

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

D4.1 – Definition of the compilation framework

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	36 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	WP4
Deliverable No.	D4.1
Deliverable Title	Definition of the compilation framework
Lead Beneficiary	TUD
Type of Deliverable	Report
Dissemination Level	Public
Due date	30/06/2021

Document Information

Delivery date	July 23, 2021
No. pages	49
Version | Status	0.4.2 | Final
Responsible Person	Jeronimo Castrillon (TUD)
Authors	Jeronimo Castrillon (TUD), Felix Wittwer (TUD), Karl Friebel (TUD), Gerald Hempel (TUD), Burkhard Ringlein (IBM), Stephanie Soldavini (PDM), Christian Pilato (PDM), Mattia Tibaldi (PDM), Fabrizio Ferrandi (PDM), Stanislav Bohm (IT4I), Francesco Regazzoni (USI), Kartik Nayak (USI)
Internal Reviewer	Gianluca Palermo (PDM)

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
02.06.2021	0.0	Jeronimo Castrillon (TUD)	Initial draft
08.07.2021	0.2	Jeronimo Castrillon (TUD)	Integration security, HLS, wrote high-level overview in Section 3. Minor modifications to DSL and Orchestration flows.
14.07.2021	0.3	Jeronimo Castrillon (TUD)	First completed draft including contributions by PDM, IT4I, IBM and TUD. Added executive summary and conclusions.
21.07.2021	0.4	Christian Pilato (PDM)	Updates after first internal revision.
21.07.2021	0.4.1	Jeronimo Castrillon (TUD)	Fixing security section and connections to the rest. Ready for full revision.
22.07.2021	0.4.2	Jeronimo Castrillon (TUD)	Clean up, review after full revision.
27.06.2022	0.4.4	Jeronimo Castrillon (TUD)	Revision according to project review. Added table describing existing technologies, extensions to them, and envisioned new tools along with a comparison to commercial offerings.

Quality Control

Approved by internal reviewer	July 22, 2021
Approved by WP leader	July 22, 2021
Approved by Scientific Coordinator	July 23, 2021
Revision approved by Sc. Coordinator	July 12, 2022
Revision approved by Project Coordinator	July 12, 2022

Table of Contents

1 Executive summary

The EVEREST project proposes a platform for implementing big data applications with both high performance and edge workloads following a data-driven model. This document defines the compilation framework, which plays a key role in providing high-level programming support for productivity alongside a methodology for optimization. The latter includes software and hardware-oriented transformations as well as autotuning support for runtime adaptivity. The design presented here is derived from the use case analysis, as reported in Deliverables D2.1, D2.2, and D2.3.

Based on the requirement analysis, the compilation framework follows three main programming flows, for orchestration/dataflow, for HPC kernels and for ML workloads. These three flows help provide dedicated support for the EVEREST use cases. We provide early insights and an earlier specification of how data policies (from WP3), e.g., for security, and how runtime adaptation (WP5) can be accounted for by the compilation framework. On the concrete programming flows, this deliverable specifies the language and framework support, the compiler frameworks and intermediate languages, and the hardware generation approach for the three main programming flows. We detailed extensions to existing solutions, like (1) dataflow language and runtime extension for deterministic execution of workflows, (2) big-data framework extension to manage FPGA resources, (3) stencils and MLIR-based middle end for numerical computation in weather models, (4) EVEREST-specific extension to TVM for partitioning of machine learning models, (5) new high-level system analysis to support irregular applications, (6) novel decoupled design of memory subsystem for data-intensive applications. We also specify how these different tool components exchange information, for instance, by means of source-to-source compilation or by direct interfacing via intermediate languages (e.g., via MLIR). Finally, this deliverable describes the main FPGA target platform including hardware and software interfaces, and how we will perform system integration based on the individual inputs provided by the different tool flow components.

1.1 Structure of this document

Section 2 provides an overview of the three main programming flows and an early account of interfaces to WP3 and WP5. The section provides an overview of the required (1) language and framework support, (2) source-to-source compilation flows for orchestration/dataflow, HPC kernels and ML workload, and (3) high-level synthesis and memory design flow. Section 3 provides further details on the flows, with focus on the frontend and the source-to-source compilation, while Section 4 specifies the high-level synthesis flows (commercial and open source) and the memory design flow. Section 5 describes the EVEREST platform, providing details to access FPGA resources over the cloud. The section also describes how the software and hardware components of applications are

integrated in such a system. A summary of the technologies, their status and a brief positioning with respect to existing commercial tools is provided in Section 6. The deliverable closes with a summary in Section 7.

1.2 Related documents

This report is closely related to:
D2.1 - Definition of the Application Uses Cases
D2.2 - Language Requirements
D4.2 - Intermediate report of the compilation framework (M18)
D4.3 - Alpha version of the software compilation tool flow (M18)
D4.4 - Alpha version of the hardware compilation tool flow (M18)
D4.5 - Final report of the compilation framework (M33)
D4.6 - Beta version of the integrated compilation tool flow (M33)

2 Overall data-driven compilation framework: Specification

The definition of application use cases in Deliverable D2.1 revealed a highly heterogeneous application landscape (cf. Table 2 in Deliverable D2.2), leading to a set of challenging requirements, as detailed in Deliverable D2.2. To cope with these challenges, we foresee a data-driven compilation framework, as depicted in Figure 1, with three main thrusts:

- Orchestration/dataflow programming flow, providing a high-level imperative syntax to describe algorithms that operate distributed over data streams. This is for instance important for the traffic use case (cf. Section 5 in Deliverable D2.1).
- HPC kernel acceleration, providing domain-specific languages with high-level semantics that enable powerful optimizations. This is important for several use cases that require weather simulation (cf. Sections 5-7 in Deliverable D2.1).
- ML programming support, enabling EVEREST application partners to use established ML frameworks to transparently target the EVEREST platform (cf. energy modeling in Section 5 in Deliverable D2.1).

As shown in Figure 1, the different flows shall include (i) programming support (GREQ1 in Deliverable D2.2) via high-level syntax and frameworks, and (ii) optimizations via compilation and hardware generation for performance and energy-efficient execution (GREQ 4 and GREQ5 in Deliverable D2.2). As identified in Deliverable D2.2, interoperability (GREQ2) is particularly important for ML tasks within the use cases. This shall be provided by adhering to the established TVM framework and corresponding exchange formats. Finally, all different flows shall transparently generate code for the EVEREST Platform, accounting for FPGA acceleration if available, and thus enabling high code portability (GREQ 3 in Deliverable D2.2). This also requires automatic vendor-specific changes to the device code, based on the offloading target.

In the following we provide an overview of the different flows. Detailed specifications are provided in Sections 3-4 along with system integration in Section 5 of this Deliverable. This section also briefly touches upon interfaces with the runtime environment developed in Work Package 5 (Section 2.4), and in particular, data policies from Work Package 3, with emphasis on security (Section 2.5).

No need for language-level integration nor tight compiler-level integration has been identified so far. Use cases have clear boundaries, communicating over standard interfaces. For this reason, we do not foresee a tight integration

between the flows. At tighter integration is however required at the backend for seamless execution on the EVEREST platform (cf. Section 5).

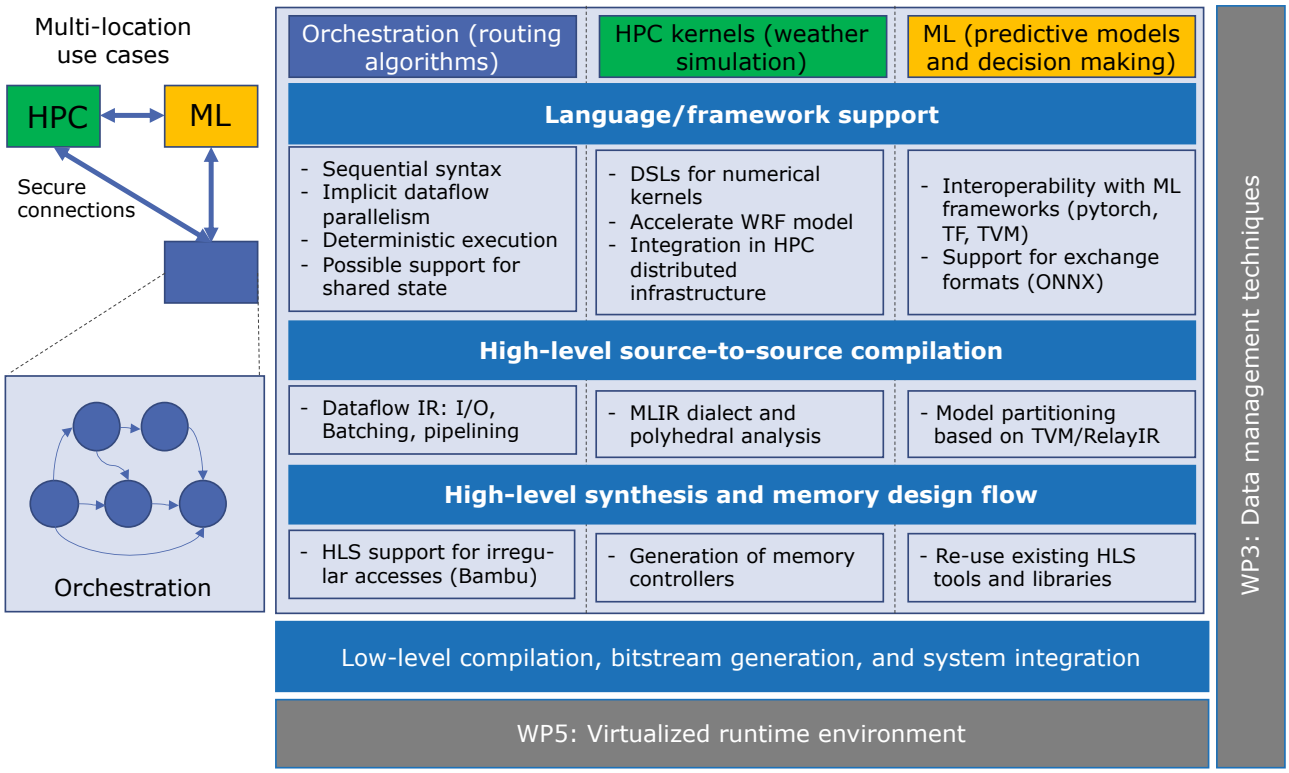

Figure 1 - Compilation framework: Overview of programming flows

2.1 Orchestration/dataflow programming

The use case analysis (cf. Deliverable D2.2) revealed highly compute-intensive workflows. Within the project, HyperLoom [1] shall be used to orchestrate these large application flows. HyperLoom/HyperTools is a platform used to define and execute workflow pipelines in large-scale distributed environments. The existing Python interface is to be extended to improve programmability (cf. Section 3.1.2). This shall include a simple imperative specification, where parallelism is automatically handled by the compiler/framework for finer-grained dataflow. The framework itself shall support batched processing to efficiently distribute shared data between simulated iterations or reuse it across workflows (cf. Section 3.1.1).

The compilation framework shall include a dataflow IR, amenable for semantic-preserving transformations for parallelism and I/O optimization (cf. Section 3.1.2.2). It should be possible to reason about determinism when producing parallel schedules of the dataflow graph.

Within dataflow use cases, irregular access patterns in control-dominated portions of the workflow were identified which can profit from FPGA acceleration. This is the case, for instance, in routing algorithms based on the probabilistic

time-dependent routing (PTDR, cf. C9-C10 in Table 2 of Deliverable D2.2). To enable ease of programming, the HLS flow shall provide support for sparse matrices, lists, dynamic memory allocation, and pointer arithmetic (cf. Section 4.1). These programming constructs are common, for example, in the routing algorithms since they allow software designers to create efficient implementations. However, commercial HLS tools do not support dynamic pointer resolution or circumvent the issue by performing transformations that remove the unsupported memory access pattern. We will extend the existing support in the Bambu HLS tool to cater for these types of use cases, without requiring source code modification.

2.2 HPC kernels

Prior experiences have shown the power of simple DSLs to analyze and accelerate complex mathematical expressions, as those appearing in numerical kernels (CFDLang (Rink, et al., 2018), TeML [2], and TeIL [3]). The compilation framework should include support for high-level specifications of mathematical expressions appearing in the WRF model (cf. Section 3.2). Given the sheer size of the model, at least the costly Radiation Module should be accelerated. Apart from providing a standalone language, the compilation framework should be integrated within the complex WRF build infrastructure. This requires modifying the WRF codes and the build scripts. Additionally, in order to disseminate our DSL and obtain feedback, we have considered building library targets that could also be integrated with existing projects right away.

For source-to-source compilation the compiler will rely on the MLIR compiler infrastructure [4] (cf. Section 3.2.2). This will allow to re-use existing abstractions and make our compiler middle-end interoperable with other middle and backends (including hardware description projects like CIRCT[1]). MLIR shall enable a modular compilation pipeline, leveraging the design effort of the open-source community. By designing EVEREST dialects that the DSL abstractions map to, we can profit from the existing lower-level dialects for linear algebra.

The types of computational kernels appearing in numerical simulation are amenable to HLS, even though, FPGAs are not yet widespread in HPC. We will investigate interfacing to two different HLS tools, namely, Bambu (cf. Section 4.1) and Xilinx Vitis HLS (cf. Section 4.2). Supporting two different HLS tools shows the interoperability of our solutions. For Bambu, in particular, the source-to-source compiler shall generate either C/C++ with OpenMP annotations or directly interface at the MLIR level (see Sections 3.2.2 and 4.1.1). Since data movement is particularly relevant in HPC, this programming flow will focus on automatically customizing the memory subsystem around the accelerators (cf. Section 4.3). To this end, the Mnemosyne tool shall be extended with support

[1] https://github.com/llvm/circt

for more memory-related components for the creation of "intelligent memory managers" that are optimized based on the information extracted during the compilation flow.

2.3 ML programming support

To support a wide range of state-of-the-art ML frameworks (pytorch, TensorFlow) and exchange formats (ONNX), as required by GREQ 2 & 4, D2.2, the Apache TVM framework [5], [6] will be used by the EVEREST tool chain. TVM can import a variety of used ML models and represent them in its unified Relay intermediate representation (cf. Section 3.3). Using this IR, further application or device specific optimizations can be applied before the ML training or inference task is packed into a binary executable that dynamically executes the application's graph.

However, the existing TVM flow is limited to a few target devices (CPUs, GPUs, and some domain specific accelerators), is optimized for single-device environments, and does not support distributed execution of ML workloads, let alone heterogeneous platforms. To support the heterogeneous EVEREST platform (REQ 5 & 9, D2.1), two key functionalities should be added to the TVM flow: (1) partitioning of the inference tasks of deep neuronal network models (DNNs) and (2) FPGA support for selected DNN inference operations. Both shall increase the overall throughput of the system. FPGA support shall increase the energy efficiency and could decrease the latency of the required inference tasks. The widely used DNN operations 2d-convolutions, rectifications, pooling, and dense layer shall be supported by both new functionalities.

Classical ML tasks (i.e. without DNNs) shall be supported by the TVM flow as well and could be optimized towards CPU environments or other compilation flows within EVEREST.

In order to increase performance or to be able to use larger models, DNN tasks must be partitioned to be calculated in parallel on multiple nodes. This partitioning shall leverage the unified RelayIR and existing TVM optimization APIs and should include the automatic creation of the required communication and synchronization between the then distributed ML tasks. This partitioning phase shall take the EVEREST heterogeneous platforms into account and schedule different parts of the models on the best fitting available type of node. The compiler backend to support FPGAs shall be compatible to the cFDK/OC-Accel environments (cf. Section 5.1) and should optimize the generated FPGA designs towards different target constraints, like e.g. low-latency, high-throughput, or low-energy.

2.4 Run-time environment auto-tuning

As identified in Deliverable D2.2., the execution characteristics of several kernels greatly vary with the input data and on the status of the execution environment To this end, the compiler shall interface with the mARGOt dynamic autotuning framework [7]by generating and exposing kernel variants that can be selected at run-time. The mARGOt autotuning framework will then have in charge the selection of the alternatives by relying on static or profiling gathered information. A specification of the interfaces will be provided in Deliverables D5.1 and D5.3. For the DSL compiler's perspective on runtime tuning, see Section 3.2.5 of this document.

2.5 Data policies and security considerations

From the analysis carried out in WP2 and reflected in Deliverables D2.1, D2.2, and D2.3, we identified as the most relevant security requirements that the EVEREST environment should provide are (1) confidentiality (only the authorized parties can access the information), (2) integrity (the information communicated between the two parties is not tampered with) and (3) authentication (assurance that a message was sent by the purported party) of the data processed and transferred to and within the environment, and the possibility of tracking the flow of information in the systems. These requirements are provided by means of security primitives, including encryption/decryption and authentication (often coupled together in authenticated encryption), and by library and extensions to support the so-called information flow tracking. The goal of information flow tracking is to follow a program's flow and data progression. When applied to security, information flow tracking helps to keep track of potential security hazards and to avoid that they could affect the security of the overall system.

Given the complexity of the EVEREST platform, the compilation framework should provide automation to help user define and deploy security policies. To do so, we look at the most elementary element we see in our system, the node, and we address the problem by providing security at two levels: inter-node (which deals with the communication between different nodes) and intra-node (which deals with communication within the node).

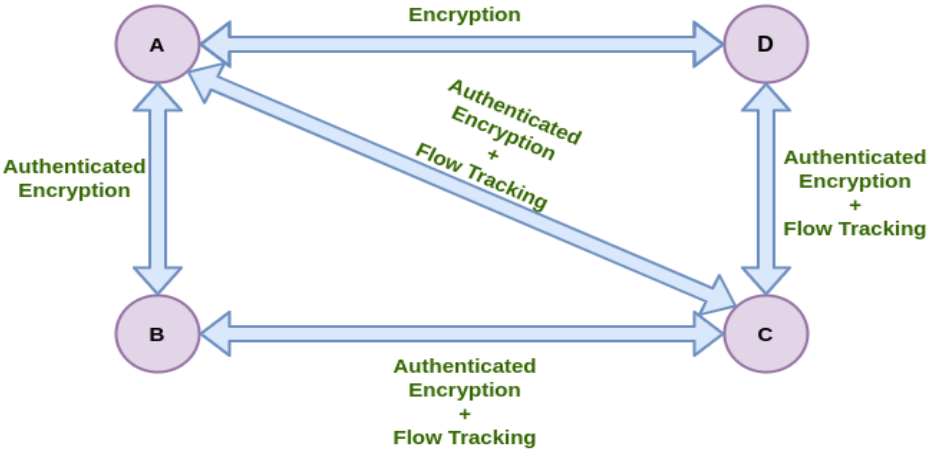

Figure 2 - Internode security considerations

Inter-node means that the node is considered as an atomic (intended as a computational unit that cannot be divided) and secure entity, and all security functionalities should be provided only till the boundary of the node. As an example, in Figure 2, we depict a simple communication model between several nodes in the EVEREST setup. Some communications must be just encrypted, others require authentication in addition to encryption, and, for some interactions, the flow of information must be followed. These types of interactions happen at a course granularity and must be supported by orchestration (cf. Figure 1 and Section 3.1). High-level annotations should allow the programmer to define the desired security functionalities. The system integration should instantiate the required SW and HW components to account for this (cf. 5.2). Exact policies and components will be further defined in WP3.

At the intra-node level, we maintain the property of a node of being an atomic entity, but only at the logic level. This means that different components of the node can be physically separated. The situation is depicted in Figure 3, which depicts a logic node composed of a CPU and an FPGA, each of them residing in physically separated nodes. To maintain the atomicity property of such nodes, we need to ensure that the elements belonging to a node form an enclave, where all the communications between the components of the node are secured and not accessible by elements not belonging to the node, even if physically located close to them. Once the logical node is created, it will be treated exactly like a physical node, where intra-node security properties can be specified. Of course, to ensure logical isolation of the nodes, we also need to be sure that all the elements of the node are extended with the needed security features, which includes, among others, the extension of memory controllers to support such an isolation (Section 4.3). Furthermore, this will require runtime support, in WP5.

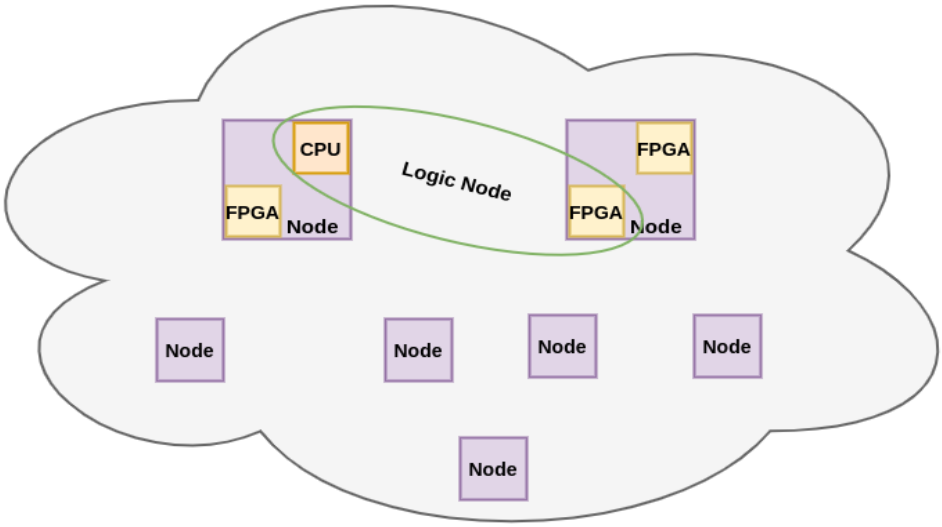

Figure 3 - Single logical node distributed across physical nodes

3 Domain-specific abstractions and intermediate representations

This section provides detailed specifications of the language support and the source-to-source compilation frameworks for the three main flows introduced in Section 2 (cf. Figure 1), namely support for orchestration/dataflow (Section 3.1), for HPC kernels (Section 3.2) and for ML use cases (Section 3.3).

3.1 DSLs for Workflow Orchestration

In this section, we focus on mechanisms for workflow orchestration and efficient dataflow execution. In general, we differentiate between two types of workflows within the project: Batch-enabled workflows intended to be executed on HPC clusters and potentially streaming-enabled workflows deployed on a cloud infrastructure.

For the workflow type, which is primarily featured in the Traffic Simulation Use case as detailed in Deliverable D2.2, HyperTools (which includes HyperLoom [1]) is to be used to orchestrate workflow execution. As efficient resource usage is one of the key goals of the EVEREST project, the framework has to allow for the intuitive specification of node- and system-level resource requirements.

The orchestration of streaming-enabled workflows is currently less well-defined. A suitable abstraction for this task should allow deterministic execution of workflows to easy debuggability and enable result reproducibility. Additionally, insights into the dataflow of the application could be leveraged for improved resource usage and performance. A DSL for this purpose should abstract from the dataflow graph it will use internally and expose a specification language that is ideally sequential, not including any parallelism paradigms. The compilation flow will have to contain a middle-end featuring an intermediate representation that supports transformations to rewrite the dataflow graph in order to introduce parallelism and reorder nodes. The backend of the proposed flow will then generate code in a high-level programming language compatible to the language used elsewhere in the project.

3.1.1 Abstractions and tools for orchestration and batch processing in HyperTools

We are introducing HyperTools as a set of tools that also includes HyperLoom. Other tools in this set are HyperQueue[2], Quake[3], and RSDS[4]. All these tools are based (or will be rewritten) to the same infrastructure core called Tako[5] that includes a basic scheduler, communication with workers, security, and resiliency

[2] https://github.com/It4innovations/hyperqueue
[3] https://github.com/It4innovations/quake/
[4] https://github.com/It4innovations/rsds
[5] https://github.com/spirali/tako/

features. The core may be wrapped in various tools and that expose different interfaces or task types. For example, it may serve for orchestration of standalone applications or Python programs through a Python API. This allows us to decouple the development of the orchestration infrastructure and capabilities from the EVEREST API (REQ2.1, D2.2). The main work in Tako will be focused on the resource management to fulfil REQ2.2 and REQ2.3 in Deliverable 2.2. The tool must be extended to express resources for long-running service-like tasks (REQ2.2) and to express FPGA resources (REQ2.3).

To better cater for the EVEREST Platform, the HyperTools task manager will be extended with better support for FPGAs. The current version allows to specify resource requirements mainly with respect to CPUs and allocation of strategies of cores on NUMA systems. We will extend these capabilities to define FPGA resource requirements also considering the IBM cloudFPGA architecture. To support the inter-node security mechanisms (cf. 2.5), we will leverage the cryptographical layer in Tako, which is based on XChaCha20Poly1305 implemented in Orion crate[6]. It uses a symmetrical encryption. It is expected that key is exchanged outside of the system, e.g. in case of HyperQueue via a shared file system and protected by the file permissions.

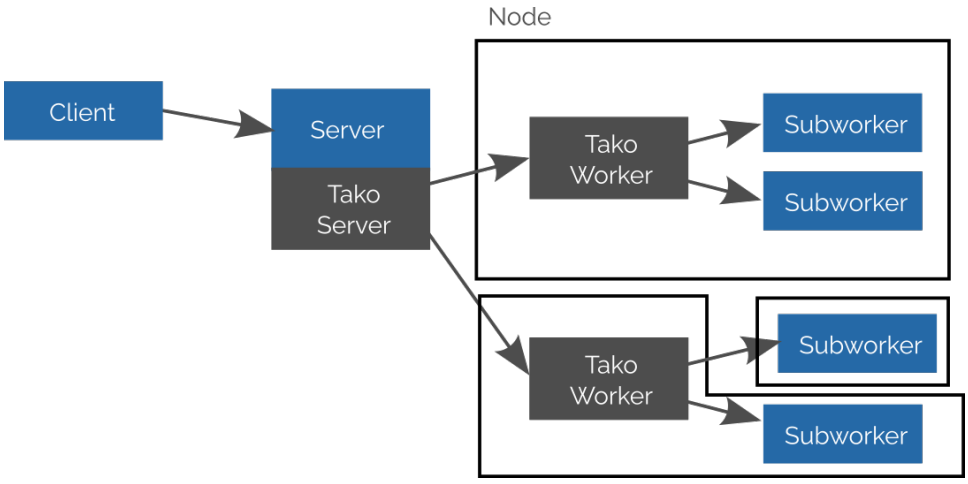

Figure 4 - Tako architecture

3.1.2 A dataflow abstraction for streaming-enabled workflows

For streaming, we plan to extend the Ohua framework [8] a compilation framework for implicit parallelism based on dataflow graph rewrites. Ohua represents a well-tested base for the workflow optimizations envisioned in EVEREST.

[6] https://github.com/orion-rs/orion

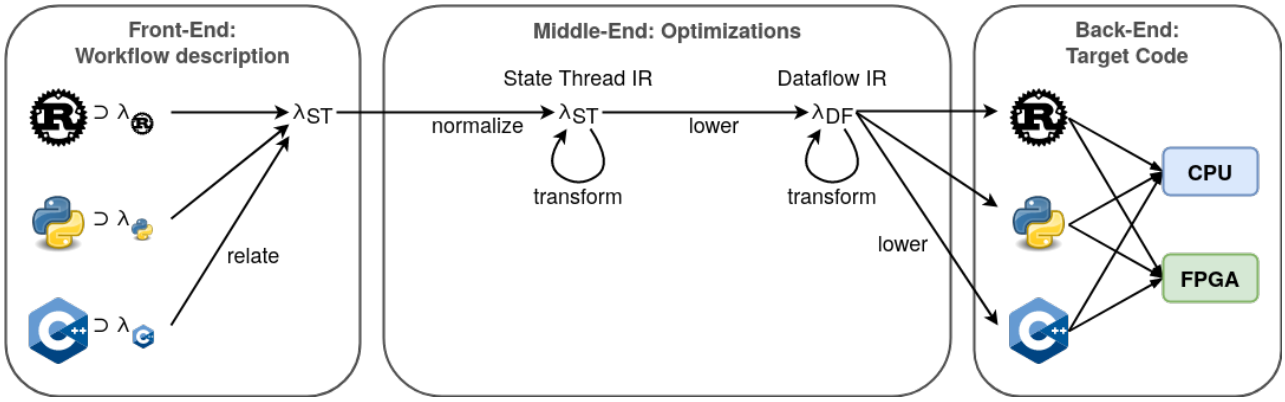

Figure 5 - Overview of the envisioned compilation flow for a language that allows dataflow optimizations and streaming workflows

The compilation flow as shown in Figure 5 outlines the intended source-to-source compilation flow. The frontend, which exposes different language dialects resembling other high-level languages, maps the input to an Intermediate Representation that helps understanding state dependencies. This can be optimized as seen fit before lowering to a dataflow IR, amenable for graph transformations. Both IRs are strongly related to the lambda calculus, which eases reasoning about the transformations. The backend then generates target code that can be compiled down for use on CPUs or FPGAs. The framework is flexible, allowing dataflow nodes to represent complex functions, including HPC kernels like those described in Section 3.2). This flow should be able to handle kernels as black-boxes, optimizing around them, e.g., via simple reordering operations.

3.1.2.1 Frontend

The frontend is defined by the language that is exposed to the application developer and used to define the workflow. In order to fulfill the Global Requirement of *Programmability* (GREQ1 as defined in D2.2), the frontend is designed according to the following core principles:

- **Familiar high-level language**
  The DSL grammar is modeled after a subset of high-level programming languages like Rust, Python or Go. This is done to avoid forcing the developer to learn a new programming language and, by extension, in the interest of maintainability. This way, the dataflow program is easily comprehensible for end-users, lowering the barrier to entry. Using a high-level language also allows to hide the dataflow aspect of the framework from the user, freeing them of the burden to reason explicitly about graphs.
- **No parallelism primitives**
  The frontend does not expose any explicit parallelism paradigms like threads or synchronization primitives. This makes the program appear sequential to the user but allows the compilation flow in fact to freely

optimize the created workflows for an arbitrary number of workers without side effects.

- **Seamless integration of library functions**
  All modern widespread high-level languages feature mechanisms for importing libraries into projects. The frontend also exposes this mechanism to enable the re-use of code. With these mechanisms, library functions can be used, just as if developing a normal program in the chosen high-level language.
- **Multi-language support**
  The frontend has a modular design, allowing for the definition of different frontend languages to cover a wide range of use cases written in different languages, which is relatively easy to achieve as the Dataflow Graph is synthesized from the input and not written directly. This also helps to fulfill the requirement for *Multi-Target Code Generation* (REQ3.8, D2.2) by lowering the abstraction level to the host language to begin with to avoid complex lowering operations later on.

To provide *streaming support* (REQ3.4, D2.2), the language features a primitive for representing streamed data processing. The normal loop syntax of the host language could be used for modelling this, transforming the loop into a dataflow operator that continuously streams data through its body. To support extra-functional requirements for the dataflow, e.g., for security (cf. Section 2.5), we plan to use type annotations. By annotating the types of variables that flow between coarse-grained dataflow nodes, we can pass inter-node security requirements down the compilation flow. This will influence the hardware and software components that are instantiated during system integration (cf. Section 5.2.3). This will be specified in more detail as the implementation of the use cases advances.

3.1.2.2 Intermediate Representation

A suitable intermediate representation for the DSL compilation flow accommodates the fact that the main focus in the flow lies on dataflow optimizations. Hence, a close resemblance to a dataflow graph is desirable.

As Figure 5 shows, the compilation flow will feature multiple Intermediate Representations which are all loosely based on the lambda calculus. While the language allows the definitions of function-like looking algorithms, most calls are made to functions in the host language, which are modeled as operators in the IR. Data dependencies between operators are represented as channels through which the data will be sent.

The compiler internally differentiates between stateful functions, which can hold internal state associated with them and pure dataflow functions which have no side effects but merely process the input data to produce an output. This

differentiation is important when reasoning about possible performance optimizations, as stateful loops (e.g., the continuous updating of an internal value in a loop) cannot be parallelized easily while loops containing pure functions may be executed in parallel without restrictions. Special kernels, like HPC kernels, can be accommodated as a new function type in the IR, which will remain untouched and unoptimized by the Ohua flow. Other options include just mapping the kernel functions to the existing function types although the compiler would have to ensure that no conflicting optimizations are performed.

The State Thread IR is close to the input language and retains higher-order functions such as for or if. It also has a concept of state threads [9], which are functions that can access their privately owned state. This IR is lowered to the Dataflow IR, which no longer has explicit control flow or higher-order functions, as shown in Figure 6. Both IRs differentiate between stateful and pure functions to ensure sane transformations. But apart from that they allow for different optimizations as they expose different aspects of the workflow. Possible transformations are discussed in the following subsection.

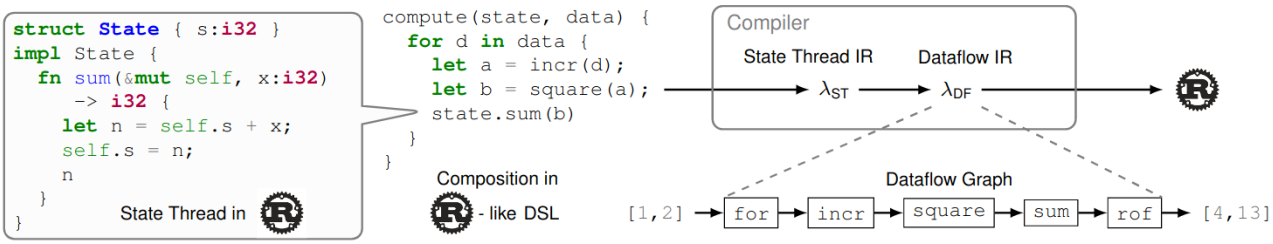

Figure 6 - Example outlining the translation of stateful sequential programs written in the frontend language (middle, here resembling Rust) into dataflow programs. The frontend language uses functions written in the host language

3.1.2.3 Middle-end

All optimizations like the parallelization of operators are part of the middle-end of the compilation flow. Here, transformations are applied to the Intermediate Representation of the program in the form of graph rewrites. Special attention is paid on retaining the deterministic nature of the program throughout all transformations and be transparent about the guarantees that are given by the individual transformations.

Compiler Transformations for Dataflow programs (REQ3.7, D2.2) implemented here could include, but are not limited to:

- **Parallelization of loops**
  The Intermediate Language has a notion of stateful and pure functions. This allows the compiler to recognize loops or parts of loops that are state-free and therefore easily parallelizable.
- **I/O batching**
  I/O operations are by nature very time-consuming and can impact program performance when done frequently. Hence, it makes sense to collect and batch such operations [10] to reduce the latency impact of the operations and make performance more predictable.
- **Batching of State accesses**
  Stateful operations are usually a bottleneck in parallel programs as they either form a bottleneck or require complex synchronization primitives not available in this DSL. As a result, other solutions to improve performance of stateful parallel programs need to be found. One solution that comes to mind is the batching of state accesses to reduce the frequency of accesses, allowing the program to run for longer periods of time in parallel. Of course, such an optimization will have to be discussed from the perspective of determinism of the program.

3.1.2.4 Backend code generation

The backend is tasked with generating code for the target platform from the given optimized Intermediate Representation. In this case, the compiler produces code for the workflow in the host language of the environment, the same language the frontend was modeled after. This allows a seamless integration into the standard compilation process of the workflow and thus helps fulfilling the requirement for *multi-target code generation* (REQ3.8, D2.2). A new target can be easily added by providing an implementation of the relevant frontend and backend functions.

3.2 DSL for numerical applications

In this section, we focus on the proposed domain-specific language (DSL) and custom compiler toolchain for the numerical workloads. This specification targets the language itself and its corresponding DSL compiler infrastructure. The custom compiler toolchain is designed to automate the process of deploying efficient computational kernels on the EVEREST platform. Figure 7 shows how the DSL compiler augments a typical WRF compilation flow, allowing the DSL to be mixed with the large existing code base.

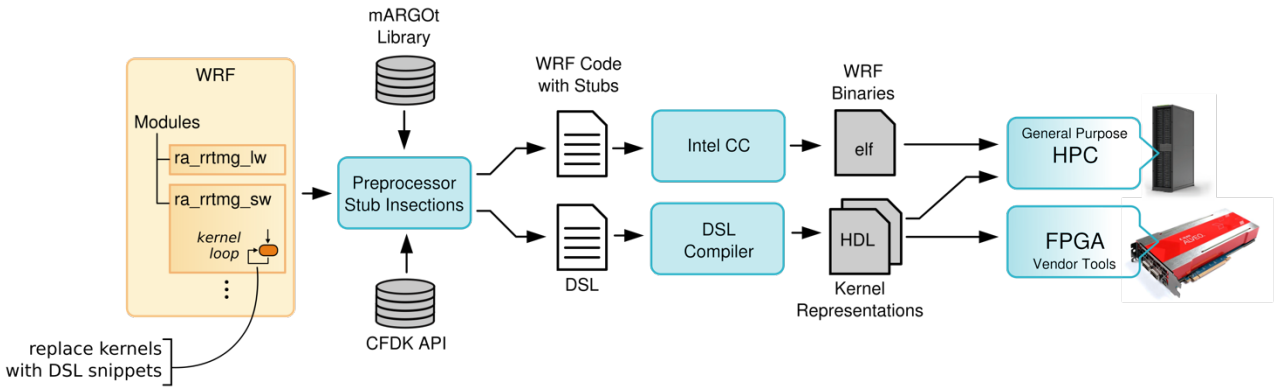

Figure 7 - Building the WRF model with the DSL compiler toolchain

In the weather simulation use case, the DSL compiler will be used to offload the time-consuming radiative-convective processes. Dividing the domain into discrete columns, the radiation driver simulates the scattering of photons and the subsequent transfer of energy. As a result, vertical movement of air and thus the formation of clouds is triggered. Both processes have governing equations with differential formulations that we can implement in the DSL. To improve performance, we plan to move even more operations to DSL, such as stencils and random sampling.

The proposed flow includes a middle-end with an intermediate representation (IR) that can represent high-level transformations. To achieve satisfactory results, some restrictions must be placed on the input kernels for this middle-end, and some domain knowledge is required. Both are to be realized with the kernel DSL, which shall also reduce the burden on the user added by this new indirection. To some extent, DSL and IR are independent in their design to allow for fluid language development as per Deliverable D2.2 Section 8.2.6.

3.2.1 Frontend

The frontend, which is the interface exposed to the user, is designed to meet the following goals:

- **Brevity**
  In the interest of maintainability, and to reduce barrier to entry, the language should adopt a terse and generic notation.
- **Familiarity**
  To avoid users having to learn a new language, syntax and notation should be self-evident, and preferably close to the notation typical in the domain.
- **Convenience** *(REQ 3.2, D2.2)*
  As kernels are integrated into larger, mostly legacy, systems, the language must allow for easy connection with surrounding code.
- **Completeness** *(REQ 3.1, D2.2)*
  The language must allow the user to encode as much of the problem as

required for the target optimizations, providing at least a complete linear algebra abstraction.

All these goals fall under the scope of the DSL language design. As the proposed implementation allows us to adjust the design during development within reason, this specification highlights some of the fixed aspects.

As outlined in Deliverable D2.2, the DSL must provide an abstraction of linear algebra, specifically in terms of tensor operations. **Completeness** is achieved by providing an index-based notation that allows users to formulate almost arbitrary expressions via their individual elements. This is done with the following syntax, which must still be extended to support stencil operators.

```
A_ij = B_i * C_j + D_ij
```

Additionally, the type system and kernel interface must have clearly defined overlap with the supported host languages, i.e. the programs the kernels are embedded in. **Convenience** is achieved through type concepts that map directly to Fortran and C/C++ multidimensional arrays, as well as capturing memory layouts. A candidate syntax is given by the following expression, which leans more towards Fortran for better readability in context.

```
var A(11 11 11) layout(k*121 + i*11 + j)
```

We target the more subjective goals of **Brevity** and **Familiarity** by deriving the DSL grammar from existing languages. The following list names candidates that set the user expectations for languages in the linear algebra domain, and the features they contribute to our implementation:

- **TACO** [11] is a compiler focused on exploiting sparsity of tensors for efficient kernel implementations. *It features a terse syntax that adopts the implicit Einsteinian summation convention.*

  ```
  A_ijk += B_ui * C_vj * D_wk * U_uvw
  ```

  Here, the axes over which reduction is performed are implicitly given by the indices `u`, `v` and `w`, which only appear on the right hand side.

- **TVM** [5] is a large compiler framework for implementing machine learning workloads. It can consume a subset of Python code that represents tensor operations in an explicit and imperative manner. *It features dynamically-sized tensors.*

```
var A : (M N N)
```

This declaration encodes the constraint that the last two ranks of `A` have identical but arbitrary dimensionality.

- **TensorComprehensions (TC)** [12] is a language focused on writing pure functions exclusively defined over tensor values. *It features functional tensors-as-values semantics and embeds into MLIR.* We extend this by making indices optional in more places, such as per-element operations.

```
A = B + C
```

- **CFDlang** [13] is a language initially created to model specific formulations of fluid dynamics problems. *It features CFD-specific short-hands and explicit temporary variables.*

```
A = (S#S#S#u)^uivjwk_uvw
```

The above statement shows a possible evolution of the contraction syntax that is both terser and closer to the mathematical notation.

Given the fine granularity of numerical kernels, we do not foresee a need to annotate security information for inter or intra-node mechanisms at this level.

3.2.2 Intermediate representation

To balance abstraction and target-specific optimizations, the IR shall be layered. In a layered IR, different levels of abstraction are used based on the compiler strategy, and may exist simultaneously within a translation unit.

The only such framework that is close enough to maturity to be usable for our purposes is **MLIR**. Different abstractions are encoded in dialects, which may declare custom types and operators. **MLIR** excels at rapid development of DSLs, as custom dialects with subsequent lowerings to existing ones can quickly create a functioning pipeline. Originally developed for machine learning, and with existing frontends for **TensorFlow** and **TC**, it aims to unify all methods and targets for machine-learning applications. Growing support and incubator projects in the realm of HLS make it a very promising development target, also for ongoing standardization efforts.

An amenable encoding of our DSL in **MLIR**, however, requires a dialect that allows for more flexible transforms on tensor expression trees. **TeIL** is an IR for tensor expressions that was created to unify the different tensor frontends. It

reduces all tensor programs to primitives, featuring more abstract value semantics than **MLIR**'s usual treatment of tensors.

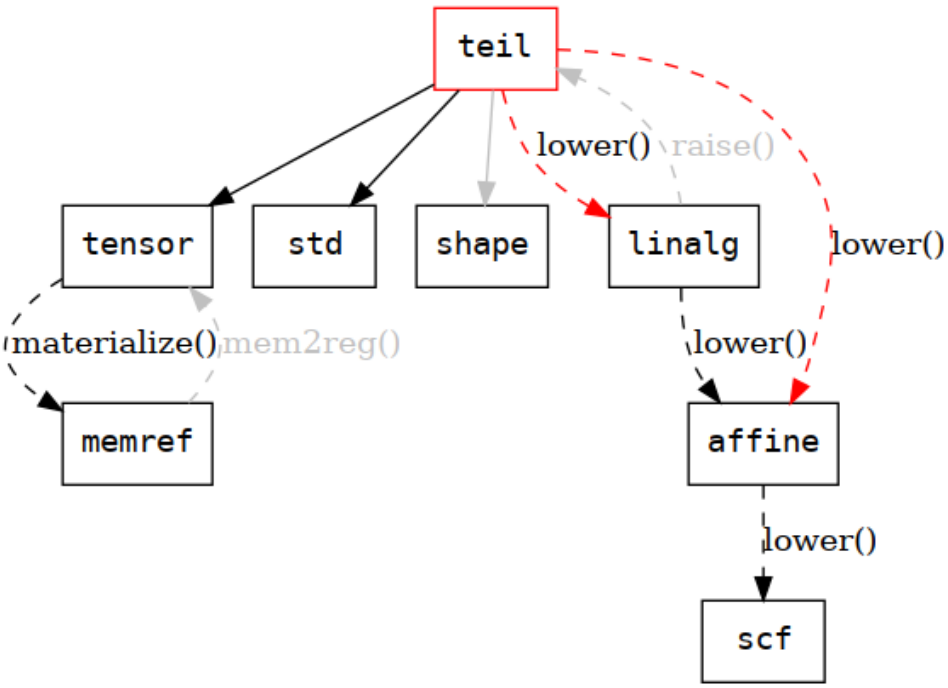

Figure 8 - Teil dialect interactions

Therefore, our IR will be a value-based tensor expressions dialect "`teil`" based on **TeIL** that is integrated using the **MLIR** framework. The language constructs of our DSL will map directly to this dialect, or may bypass (parts of) our middle-end explicitly via the standard dialects. Additionally, the optimizations identified in **TeIL** and **CFDlang** can be expressed directly as transformations on this dialect. Figure 8 shows the hierarchy of existing dialects below `teil`, with red arrows indicating new components. The gray arrows show future extensions that may extend the scope of optimizations.

Current **MLIR** development is heading into a direction of value-based semantics for tensor expressions, which indicates a readiness for further standardization. If successful, this would amplify the reach of the EVEREST project. Support for a variety of additional use cases may be achieved in this way.

3.2.3 Middle-end

The middle-end, which operates on the IR and thus **MLIR**, requires a more rigid design than the other components. Its architecture and implementation is largely dictated by **MLIR** guidelines. It must achieve the following goals:

- **Completeness** *(REQ 3.1 3.2, D2.2)*
  All kernels falling under the desired abstraction level must be supported, and require embedding into potentially arbitrary code.
- **Performance** *(REQ 3.6 3.7, D2.2)*
  The middle-end must produce outputs that achieve notable improvements in terms of energy and/or performance metrics.
- **Heterogeneity** (*REQ 3.8 3.10 3.11, D2.2*)
  The different execution targets of the EVEREST platform must be supported transparently.
- **Extensibility** (*REQ 3.5, D2.2*)
  Apart from future host compilers and languages, additional targets, such as GPUs, are among likely extensions. The toolchain should support an exchange format for these purposes.
- **Tunability** *(REQ 3.9 3.11, D2.2)*
  The compiler shall be able to export parameters for auto-tuning.

The middle-end is roughly a collection of functions that serve the following purposes within the **MLIR**-based pipeline.

- Canonicalization of the `teil` dialect.
- Optimization passes on the `teil` dialect.
- Verification and diagnostic visitors for kernels.
- Lowering of the `teil` dialect onto the `linalg` and `affine` dialects. (cf. Figure 8)
- Code injection passes for runtime (tuning).
- Code generation passes for HLS artifact output.
- In-MLIR HLS and/or hardware toolchain specific translation. *(Optional*, cf. Section 4.1)
- Raising from `linalg` to `teil` constructs. *(Optional)*

By opting for **MLIR** and providing lowerings, we achieve **Completeness** and also a reasonable level of **Performance** by falling back onto the extensive existing compiler infrastructure. Similarly, the design of **MLIR** provides inherent **Extensibility**, especially with respect to GPU targets.

The following are possible paths to interface with the HLS flow. At least one of these paths must be supported by the DSL compilation toolchain:

- Interfacing via source code to HLS tools.
- Extend HLS tools to accept (a subset of) **MLIR** directly, allowing for more precise control from the middle-end. The **Bambu** HLS flow, for instance, will be extended with such support (cf. Section 4.1). Also, Xilinx's Vitis HLS now accepts **LLVM-IR**, which **MLIR** can be lowered to.

- **MLIR** dialects such as the **CIRCT** project aim to move the HLS task entirely into the IR, eliminating the need for external HLS altogether. These flows would end in an export to a vendor-specific place-and-route tool.

All of the above methods can be used to create a more vendor-independent HLS flow as discussed in Deliverable D2.2.

It shall be noted that the previously mentioned **TVM** also uses its own unique IR to implement code generation for a variety of devices (cf. Section 4.3). Extensibility is achieved by allowing compiler developers to hook into this process, with more elaborate methods being enabled through the "Bring Your Own Codegen" (BYOC) API. This does, however, provide far less opportunity for applying domain-specific transformations.

3.2.4 Analysis and transformations for HLS

Existing vendor HLS flows already allow for the user to guide the synthesis process using additional attributes. These are indispensable for achieving high performance, but do not mix very well with the input code. They often rely on a declarative (`#pragma`) extension or force the user to adopt specific patterns in their code.

The DSL compiler shall automate this task by generating code that conforms to these requirements. The most basic solution would use the exact same mechanisms exposed to an HLS user, but with more extensive and automated code transformations.

By abstracting the problem of embedding kernels into larger systems, the interactions between them become more implicit. This opens up opportunities for optimization, in particular with respect to the memory subsystem. In this context, the compiler will provide support for:

- **Memory Allocation**, which refers to the process of committing physical memory resources to store values of variables in a program. A common goal is to reduce the amount of committed resources while trading with contention on them.
- **Data layout and representation**, meaning the way values are stored in memory. Inherently structured values, such as tensors, can be mapped to linear address spaces in different ways. Additionally, a compiler may reason about different data representations for individual elements based on precision and other constraints.
- **Data transfer schedules**, which correspond to decisions of when to move data between memories of one or multiple systems. In a heterogeneous system, this becomes an increasingly important problem.

Section 5.3 shows how these optimizations can be implemented. The required information is inferred by the compiler, which produces the machine-readable interchange format described there. The analysis processes required are rooted deep within the kernel compiler reasoning and often cannot be decoupled from other tasks. That also means it is largely referential to a particular kernel representation in the compiler's IR. As a result, we consider tying this information to the IR in a more standardized way such that it can be consumed easily, as part of a more composable flow.

3.2.5 Exploiting variance at runtime

According to *REQ 3.9*, D2.2, a runtime tuning mechanism is to be implemented, which adapts implementation parameters according to on-line data (see also Section 2.4) and execution environment. There are two consequences for the proposed system that are common to all methods of implementing this feature:

- A **runtime system** is required to implement policy and mechanism for this behavior. Roughly, this means it must make decisions on the runtime tunable parameter's values (see **mARGOt**) and then apply them.
- At compile time, **explicit variance** must be made available in the produced artifact. In other words, the compiler must identify and expose runtime tunable parameters in its implementation and provide a tie-in for the runtime.

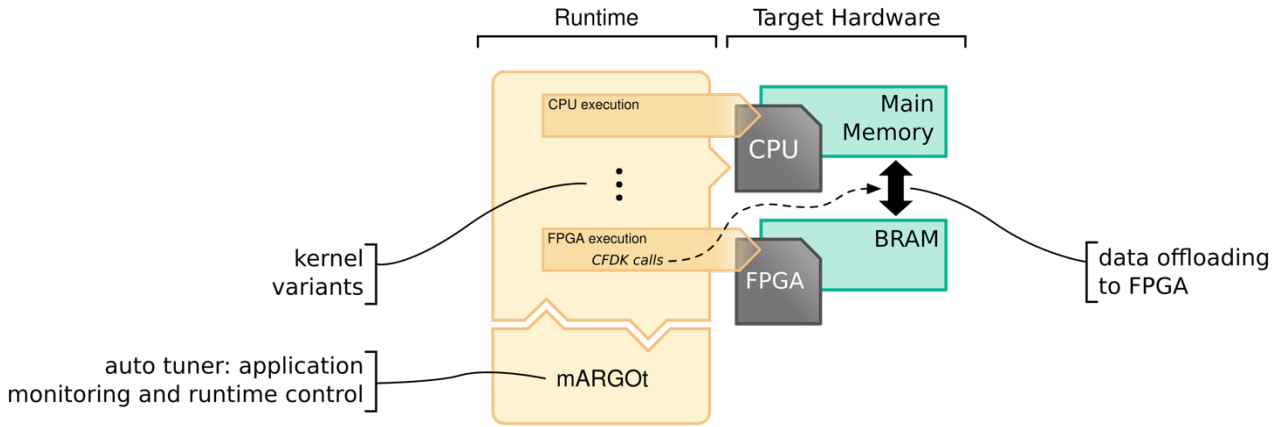

Figure 9 - Kernel runtime

Figure 9 shows the view the runtime has over the running system. In order to make a beneficial decision, it must monitor the runtime performance of the depicted components. Using that data, it enacts at least the following decisions:

- **Parameter tuning**
  The runtime adjusts the values of statically-known tuning variables, such as accuracy thresholds and discretization constants. In general, these must be exposed manually.

- **Retasking**
  The runtime uses the cloudFPGA API to (re-)assign (parallel) tasks to execution nodes to balance latency, utilization, and efficiency. A minimal strategy is always required for the deployment of any application.

3.3 Machine Learning workload integration

To additionally enable the execution of machine learning (ML) workloads by the EVEREST toolchain, domain-specific abstractions popular for ML models -- like ONNX, pytorch, or TensorFlow -- will be supported. These interchange formats for ML applications evolve rapidly and to reduce the risk of incompatibilities or uncontrollable dependencies, the **TVM** framework is used to import, optimize and represent these workloads in a unified way.

Apache TVM [5], [6] is a compiler framework maintained by the Apache Software Foundation (cf. Figure 10). Its goals are to compile ML models to deployable modules while providing a large compiler infrastructure to automatically optimize the models to achieve better performance. TVM's high-level IR, called "RelayIR", can be created from the popular ONNX format. The industry-driven standard NNEF is not directly supported, but it can be easily converted to ONNX.

Within EVEREST, all three use cases specified in Deliverable D2.1 will use ML for some steps. As example, the general weather forecast will be adapted for each individual industrial site using sequential aggregation (cf. Section 4.1, D2.1). Similarly, a ridge regression using Gaussian kernels will be used to adapt the weather forecast to local measurements to better predict reusable energy production (cf. Section 3.1, D2.1). Lastly, the traffic prediction use case will train deep neural networks (DNNs) on daily traffic data (cf. Section 5.1, D2.1). Those ML workloads should also be able to leverage heterogeneous hardware, especially in the case DNNs are used (*REQ5, D2.1, REQ3.3, D2.2*).

The flow presented in Figure 10 fulfills this requirement and support the inference of DNN models on distributed heterogeneous hardware. In case of the traffic prediction model, this will be DNNs that consists of compute intensive convolutional and fully-connected layers.

As depicted in Figure 10, this proposed EVEREST flow to support the execution of ML workloads on multiple FPGA or CPUs consists of multiple steps: After the modules are imported as RelayIR module, the ML application is optimized using TVM build-in optimization passes. Using the unified community-supported RelayIR has the advantage that subsequent optimizations can be applied irrespective of the original ML framework.

To support the requirement of distributed DNN execution (*REQ8, D2.1*), algorithms will be developed that analyze the optimized RelayIR module and derive a portioning scheme for each specific DNN. This partitioning step will take detailed characteristics of the ML operations and the user-given target constraints into account. During this step, the original single module of the DNN workload will be split into one module per node. The type (FPGA or CPU) and number of nodes will be automatically derived and optimized within the constraint ranges given by the user. Please note, that these nodes can be scheduled to run on heterogeneous hardware independent of each other, based on performance and efficiency predictions of the newly developed algorithms. For example, the DNN could be split across 2 CPUs and 5 FPGAs, if this would have the best predicted performance and is within the user-given resource budget.

After the partitioning of the workload is decided, the proposed tool continues to determine the communication configuration among the nodes. This step ensures that operations that can be executed in parallel will be executed in parallel and that each node knows where and when to get and to send data. Hence, besides the data communication, also the execution synchronization will be ensured by the inserted communication modules. This communication modules can re-use existing heterogeneous communication frameworks, e.g. MPI. At this step, also use case specific pre- or post-processing can be considered and the data streams linked (*REQ8, D2.1*).

Individual modules will be optimized for their specific target (CPU or FPGA) and lowered further so that either synthesis tools can synthesize the HLS or HDL descriptions or, using the TVM infrastructure, emitting the binary for the targeted CPU. Please note that this step is individual for each type of CPU or FPGA in order to adapt the workload as much as possible to the chosen target devices (*REQ 9, D2.1*). In case FPGA-target nodes, HLS code will be emitted and EVEREST HLS design flow will be used (cf. Section 4).

As last step, depicted at the bottom of Figure 10, after the target specific binaries or bitstreams are synthesized, an automatic deployment framework will be developed, so that the ML workload can be distributed to the target devices and launched, automatically.

The TVM community is also working on adding the option of exporting RelayIR to MLIR. Depending on the availability, this feature may serve as a additional bridge to the EVEREST MLIR toolflow described above, as depicted on the left-hand side of Figure 10.

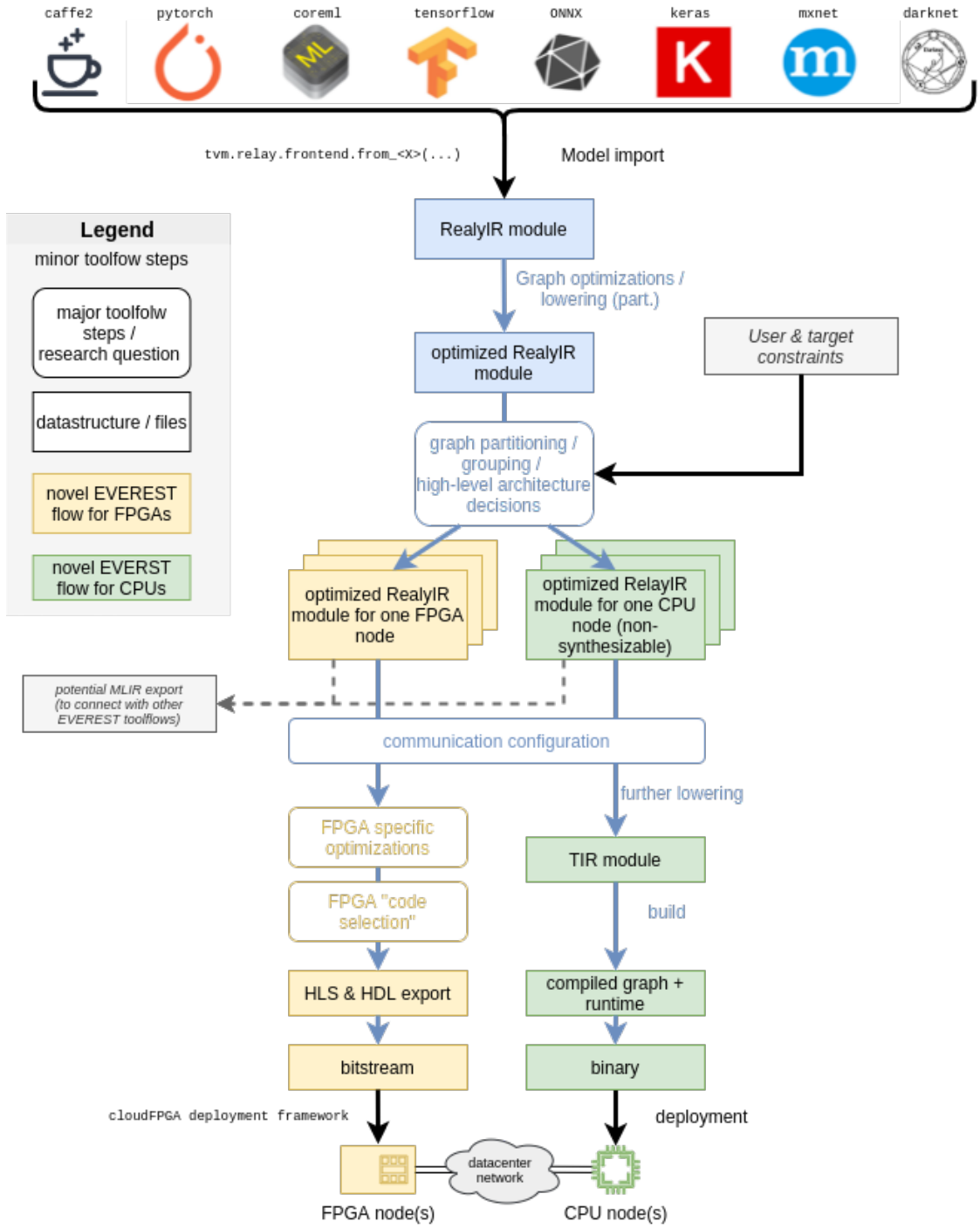

Figure 10 - EVEREST machine learning to distributed FPGA/CPU flow

4 High-level synthesis and memory design flow

EVEREST has a strong focus on FPGA acceleration and the compilation framework is thus requested to generate efficient hardware architecture to coordinate computation and data movements. The computational part of the FPGA system is generated with high-level synthesis, while the data movements are optimized with the customization of the memory architecture. The integration of the two parts is based on standardized interfaces. Also, the project has a strong focus on modularity and interoperability (*REQ 4.9, D2.2*), allowing the use of different HLS tools, namely for example Bambu [14] and Xilinx Vivado/Vitis HLS [15].

To combine these two aspects (separation of computation and data movements, and interoperability), we rely on the following:

- We export the internal memory modules from the accelerators generated either with Bambu or Vivado/Vitis HLS to enable further optimizations. For example, in case of C functions, when data structures are moved from local variables to function parameters, the HLS tools will add local-memory interfaces to the top module. This will enable memory optimizations (cf. Section 4.3) and ease system integration (cf. Section 5.2)
- Based on the HLS tool to be used, the compiler may need to slightly adapt the output format of the code to be synthesized. For example, it can introduce pragma annotations for code optimization with Xilinx Vivado/Vitis HLS. However, Bambu does not support the same pragmas, and, in some cases, it can enable further optimizations with command-line options or by writing the code in a slightly different way. The compiler may thus need to adapt the intermediate artifacts and metadata with a customization of the specific backend.
- Bambu is also going to support a direct interfacing with MLIR dialects. So, the compiler and the HLS tool will need to define the subset of information that can be exchanged between the two parts.
- We assume fixed-latency accesses to the private local memories, currently using the Xilinx Vivado/Vitis naming convention for the signals (cf. 4.3). Accesses with variable latency (e.g., to external data) will require a latency-insensitive protocol (`ready`/`valid`/`stop`). We will then create a proper wrapper to access the external memory controllers or interfaces (cf. Section 5.2).
- Additional compiler information (e.g., security annotations) can be passed through code annotations (and directly synthesized with HLS) or represented in XML annotations to be used by HLS extensions, memory generation flow, or system integration.

Figure 11 summarizes the hardware generation flow, showing how the compiler can pass information to the other parts.

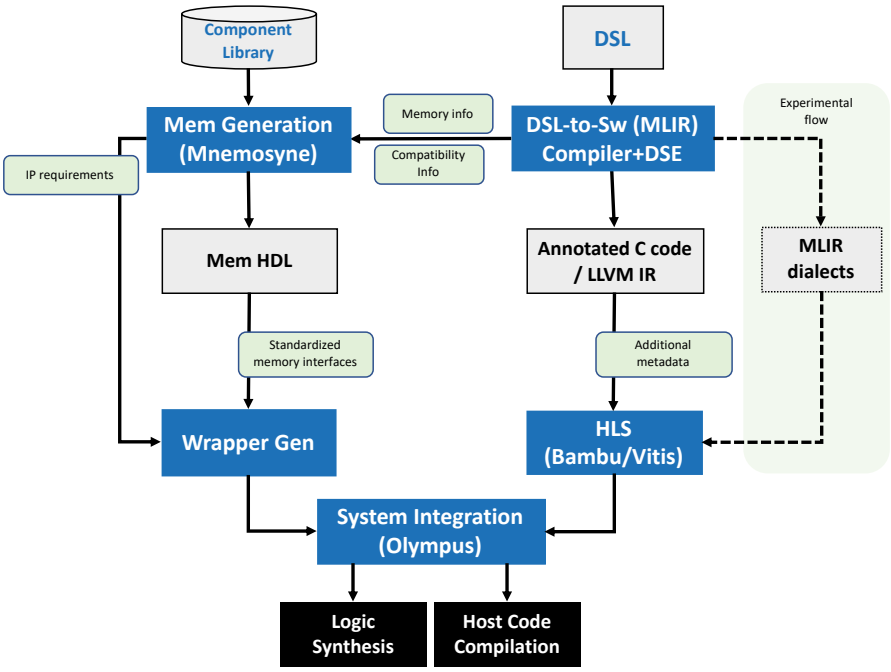

Figure 11 - Integration of compilation and hardware generation flow

4.1 Bambu HLS flow description

Bambu is a command-line tool aimed at assisting the designer during the HLS of complex applications. It supports most of the C/C++ constructs, including function calls and sharing of the modules, pointer arithmetic and dynamic resolution of memory accesses, accesses to arrays and structs, parameters passed by reference or copy, and many more.

Like in a standard software compilation flow, Bambu has three phases (see Figure 12: frontend, middle-end, and backend.

Bambu frontend. Bambu interfaces with existing compilers, such as GCC and Clang. With GCC, a plugin extracts the call graph and the control data flow graph of the functions under analysis from GCC's internal IR. Similarly, a Clang plugin extracts the same information and serializes them into a textual format easy to parse. Bambu then parses back all the compiler serialized information plus all the annotations to build a Static Single Assignment in-memory IR.

This approach decouples the compiler frontend code from the rest of the HLS process. Localizing all the changes in a GCC or LLVM/Clang plugin allows rapid and easy integration of many different versions of the compilers.

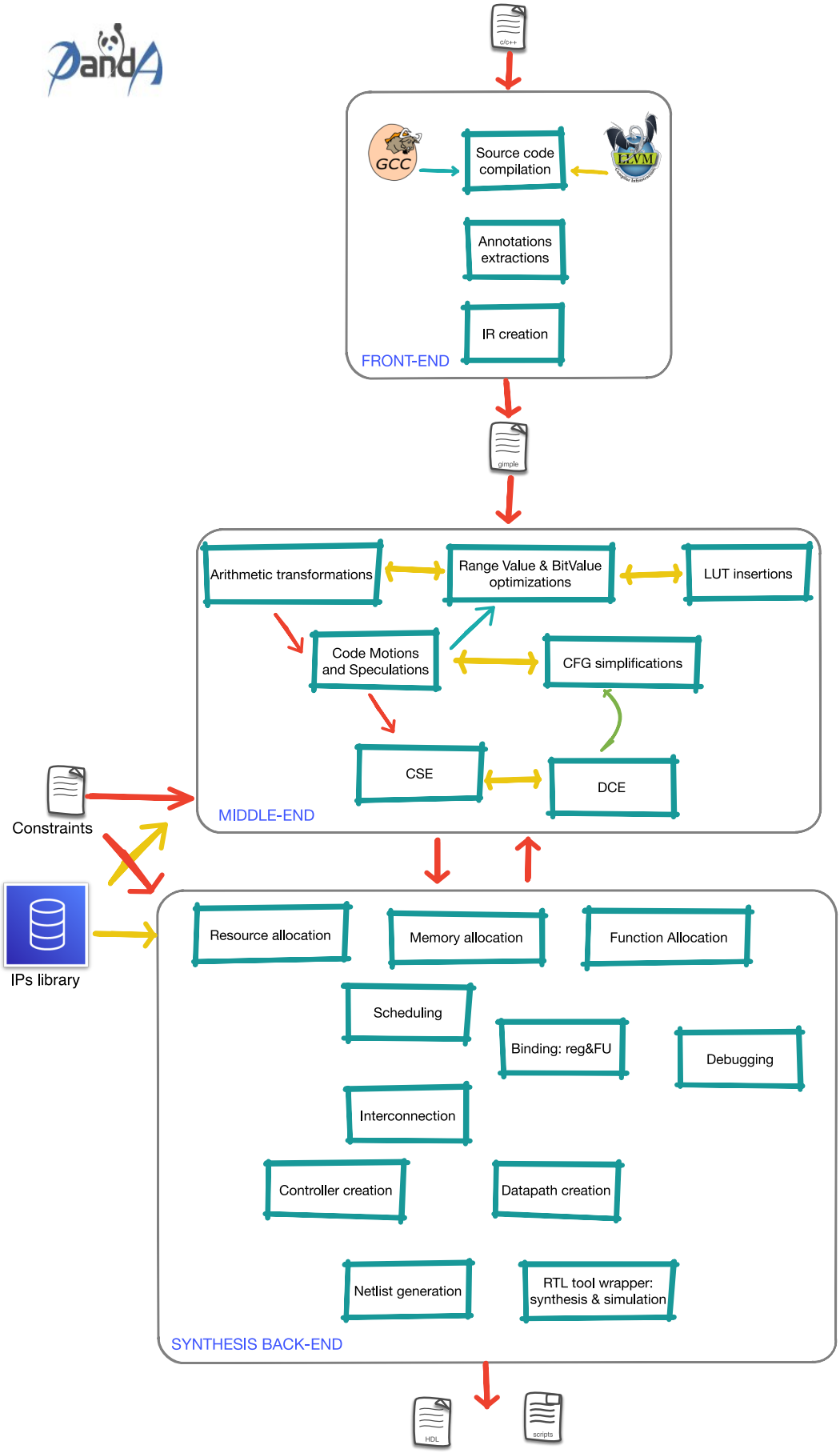

Figure 12 - Bambu HLS flow

Bambu supports GCC versions ranging from 4.5 to 8, and LLVM/CLANG versions ranging from 4.0 to 11. The Bambu frontend needs some information from the upstream compiler, such as the top function *(REQ 4.6, D2.2)* on which the high-level synthesis is performed and the specification of the Block-level/Top component and Port-level interfaces *(REQ 4.7 4.8, D2.2)*. This information will be passed through command-line options or source code annotations.

Bambu middle-end. Starting from the intermediate representation extracted from GCC/Clang, Bambu rebuilds data structures, such as the Call Graph and the Control Data Flow Graphs, and builds additional data structures such as the Program Dependence Graphs. Next, it applies a set of device-independent analyses and transformations. Some of these steps are commonly used in a software compilation flow (e.g., data flow analysis, loop recognition, dead code elimination, constant propagation, LUT expression insertion, etc.).

Multiplications and divisions by constant values are transformed into expressions that use only shifts and adders to reduce area utilization and improve timing. The resulting expression structure depends on the target device and technology, since adders and multipliers may have different performances on different devices. Bambu, for these purposes, accepts XML file and command-line options to constrain the number of resources used, the type of FPGA used, and the clock constraint at which the application must meet *(REQ 4.10 D2.2)*.

Differently from general-purpose software compilers, designed to target a processor with a fixed-sized data-path (usually 32 or 64 bits), a HLS compiler can exploit custom-size operators (e.g., a multiplier with the minimum number of I/O bits) and registers.

Consequently, we can select the minimal number of bits required for the specific algorithm's operations and value storage, which leads to less area, less power, and shorter critical paths. At this stage, Bambu also performs Bitwidth and Range Analysis, aiming at reducing the number of bits required by data-path operators. Floating-point computation is usually demanding in terms of computing resources, but it is even more demanding when the target technology is based on FPGAs. In Bambu, Bit-Value analysis and Range Analysis are done to reduce as much as possible this impact. In addition to these analyses, Bambu accepts annotations to the top function parameters and command-line options that trade accuracy with resource reductions *(REQ 4.13 D2.2)*. This will impact all the applications where double-precision accuracy is not really required, but the EVEREST users sometimes use it during the application specification (e.g., PTDR or machine learning applications).

This analysis is crucial during the optimization process because it impacts all non-functional requirements (e.g., performance, area, power) of a design without affecting its behavior.

Another important part relevant for the EVEREST project is the Memory Allocation. It defines the memories used to store aggregate variables (arrays and structures), global variables, and how the dynamic memory allocation is implemented. Bambu adopts an architecture for memory accesses that support a wide range of cases. Statically analyzing the memory accesses, Bambu builds a hierarchical data-path where memories can be classified as read-only, local, with aligned or unaligned memory accesses, or which require dynamic resolutions of the addresses to identify the physical location. Accesses to dual-port BRAMs or memory controllers with complex parallel channels are supported by replicating such memory interconnections as needed. The same memory infrastructure can also connect to external components (e.g., scratchpads, caches, and DRAMs) or directly to the bus to access off-chip memories. Supporting protocol-based accesses (e.g., FIFO or stream-based access) is obtained by generating specific components that replace the load/store instructions. Interfacing with the actual memory banks and physical controllers will be optimized by sharing information with the memory generation flow (cf. Section 4.3).

Bambu backend. In this phase, Bambu performs the actual architectural synthesis of the specification. The synthesis process acts on each function separately. The resulting architecture reflects the structure of the call graph.
Each function includes at least two sub-modules: the control logic and the data-path. Control logic modeled as a Finite State Machine handles the routing of the data values and the temporal execution of the operations. The data-path is a custom mux-based architecture with optimized data types to reduce the number of flip-flops and bit-level multiplexers. It implements all the operations and memories required during the function execution.

Bambu currently generates Verilog/VHDL compatible with many different target technology AMD/Xilinx ISE, AMD/Xilinx Vivado, Yosis-Vivado, Intel/Altera Quartus, Lattice Diamond, NanoXplore, and OpenRoad 4.5. Extension and customization for the EVEREST platform will be considered, but they should be limited since the target FPGAs are very similar to those already supported *(REQ 4.4 4.5, D2.2)*.

4.1.1 Bambu Input specification

The default input specification supported by Bambu is a behavioral description of the specification, written in C/C++ language *(REQ 4.1, D2.2)*. The C/C++ language supported is in line with the ones supported by many other commercial tools, such as Vitis HLS or Intel HLS. Bambu supports additional patterns such

as the synthesis of pointer functions and of double de-referenced pointers accesses that are usually not supported by commercial tools. These patterns are used especially when the application has irregular memory access to the memory, like in the EVEREST Probabilistic Time-Dependent Routing (PTDR). Indeed, this algorithm works on sparse graphs using such pattern.

The tool has a partial support for standard libraries such as **libc** and **libm** and a complete support for single and double precision basic arithmetic functions. An experimental port of the `ac_types` Mentor Graphics library allowing C++ specifications with arbitrary integer/fixed-point precision arithmetic has been integrated into Bambu. The tool supports both the types used by Mentor Graphics (ac_types) and the ones supported by Vitis HLS (ap_types). These types can be used to explore custom representations of the data and reduce the resource requirements of the accelerators.

If the frontend compiler is based on Clang/LLVM, Bambu can perform high-level synthesis starting directly from LLVM Bitcode File Format *(REQ 4.2, D2.2*

MLIR interfacing

Bambu will be extended by interfacing directly via the **affine** MLIR dialect *(REQ 4.3, D2.2)*. This will allow for a tighter integration with the DSL compilation flow (cf Figure 8), enabling rich information passing between the tools. The **affine** dialect represents loops as polyhedral-friendly structures and offers loop and memory operations. The **affine** dialect classifies values as symbols, dimensions, and non-affine values. The type of symbols and dimensions must be index. Symbols represent an unknown quantity that can be treated as a constant for a region of interest - loop body; symbols are, thus, loop invariant variables. Dimensions correspond to the dimensions of the underlying structure being represented. Dimensions have the same constraints as symbols, except they can also accept induction variables of the affine loops. Affine dialect offers several operations, of which **affine.for** is the most important one. It represents a loop nest with one region with one block as its body. The block usually has one argument: the induction variable of the loop. Index bounds have the same restrictions as dimensions and symbols. These constructs are widely used in HPC and ML kernels and their optimization can bring significant benefits. Other dialects can be lowered to the affine dialect, so interfacing with this dialect is not too restrictive. The following textual description shows an example of affine MLIR description.

```
func @example(%arg0: memref<1000xi32>) {
    affine.for %arg1 = 0 to 1000 {
      affine.load %arg0[%arg1]
      %1 = mul %0, %0
      affine.store %1, %arg0[%arg1]
```

```
    }
    return
}
```

MLIR provides the concept of passes to expose entry points for IR analyses and transformations; they are implemented as program traversers to either collect useful information or apply transformations. MLIR passes are often used to convert one dialect to another, as well. This use case is helpful for moving to another abstraction level. For example, affine dialect can be lowered to standard dialect, which can then be lowered to the LLVM dialect - the middle step between MLIR and LLVM IR. This is the approach we will follow to interface affine MLIR descriptions with the high-level synthesis tool Bambu. This is particularly important to directly interface with the dialects defined for HPC kernels (cf. Section 3.2 and Figure 8) and for the ML compilation flow (cf. Section 3.3). The **.ll** file generated with these lowering steps can be passed to Bambu HLS as is.

OpenMP support

Bambu supports the efficient generation of accelerators for graph kernels. The methodology enables the programmer to naturally write shared memory graph algorithms annotated with OpenMP-like pragmas and using atomic memory operations while generating related architecture templates that maximize external memory utilization through latency tolerance (see Figure 13 for the architectural template used by Bambu).

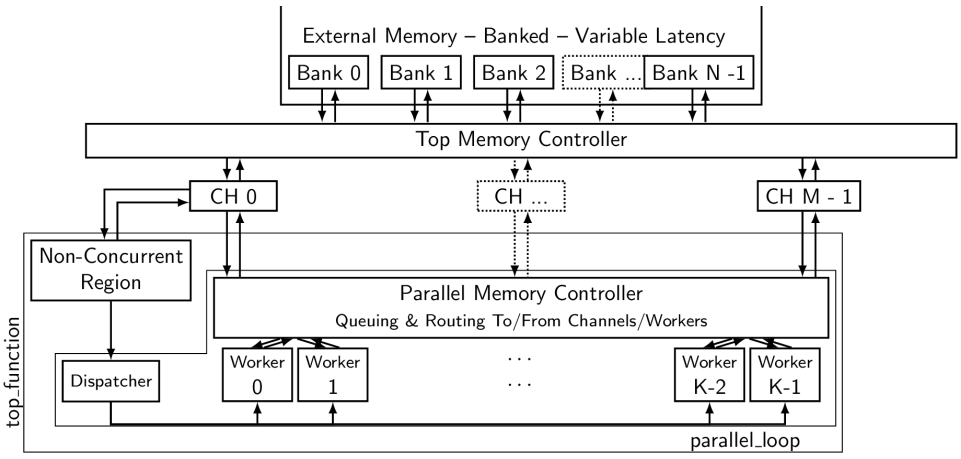

Figure 13 - Architectural template associated with the OpenMP oriented synthesis [16]

In EVEREST, the Probabilistic Time-Dependent Routing (PTDR) application contains irregular memory accesses that can be parallelized with OpenMP *for* pragmas. Bambu supports the synthesis of OpenMP parallel applications with a pattern similar to the following one:

```
void atomic_update(...) {
  #pragma omp atomic
```

```
  update_results(...);
}

void loop_iteration(size_t i, ...) {
{...} // loop body X
atomic_update(...);
{...} // loop body Y
}

void parallel_loop(...)
{
  #pragma omp parallel for
  for (size_t i = 0; i < N; ++i)
    loop_iteration(i, ...);
}
void top_function(...) {
  {...} // code block A
  parallel_loop();
  {...} // code block B
}
```

The current implementation requires that the parallel loop body and the parallel functions are wrapped in standard C functions *(REQ 4.12, D2.2)*. Pragma `omp for` support will be extended by adding dynamic/static scheduling attributes management. Another thing added to the current high-level synthesis of OpenMP for is the support of OpenMP for reductions. Support to data-sharing attributes will be improved as well.

4.2 Vitis high-level synthesis flow

Xilinx Vitis/Vivado HLS is a commercial high-level synthesis tool based on LLVM. It includes a complete design environment with several important features to fine-tune the generation of hardware accelerators. C, C++ *(REQ 4.1, D2.2)* and SystemC are accepted as input, and hardware modules are generated in VHDL *(REQ 4.5, D2.2)*, Verilog *(REQ 4.4, D2.2)* and SystemC. During the compilation process, it is possible to apply different optimizations, such as operation chaining, loop pipelining, and loop unrolling. Furthermore, different parameter mappings to memory can be specified. Streaming or shared memory type interfaces are both supported to simplify accelerator integration. In EVEREST, Vitis HLS will be used as alternative high-level synthesis flow, and the upstream compiler should use VITIS pragmas to steer optimizations. In addition to this, the compiler should insert the appropriate pragmas and annotations for:

- the top function specification *(REQ 4.6, D2.2)*;
- the Block level/Top component interfaces *(REQ 4.7, D2.2)*;
- the port-level interfaces *(REQ 4.8, D2.2)*;
- the constraints for the clock and the resources *(REQ 4.10, D2.2)*

4.3 Memory generation flow

Our memory generation flow is based on **Mnemosyne** [17]. Mnemosyne is an open-source CAD tool for the generation of local memory architectures. These memory architectures are targeted for loosely-coupled hardware accelerators where each accelerator is composed of computation logic and private local memories (PLMs). This separation between computational logic and memory simplifies the integration with HLS that can be executed in parallel.

The computation logic performs the functionality of the accelerator and is assumed to have a standard interface for the memory ports. This is a valid assumption for most of the HLS tools (e.g., Xilinx Vivado HLS or Bambu) that can be used to generate such logic. Figure 14 shows an example of accelerator interface before HLS and the corresponding hardware module. Mnemosyne decouples the optimization of the computational logic and, more in general, the memory subsystem. This is an important aspect in EVEREST to separate the optimization of data communication and storage, and the optimization of the computational aspects. The accelerator logic will interface only with the Mnemosyne-generated memory infrastructure, while the coordination of data transfers will be transparently executed.

The memory ports of the accelerator can be then connected to one of the PLM elements. Mnemosyne assumes the following ports and behaviour (where *x* in an identifier of the interface):

- `CEx` (chip enable): this signal must be active every time there is a valid operation on the memory
- `Ax` (address): this signal carries the address of the request.
- `WEx` (write enable – only for write ports): this signal must be activated when the write request is active
- `Dx` (data in – only for write ports): this signal carries the value to be written into the memory
- `Qx` (data out – only for read ports): this signal carries the value read from the memory

All command signals (i.e., CE and WE) are active high. This definition addresses requirement *REQ 4.15*, D2.2. In case of different interfaces of the accelerator logic or the other system component, the system integration logic will introduce proper adapters).

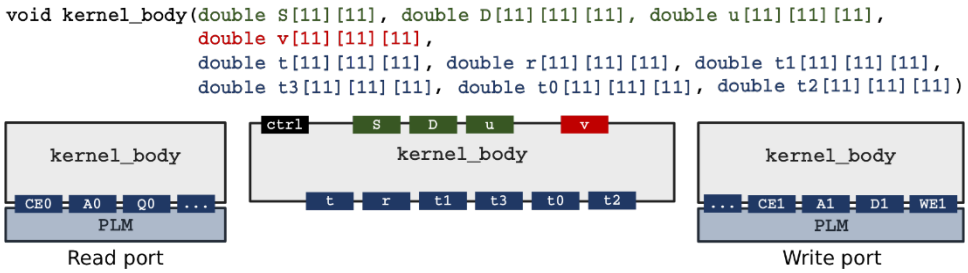

Figure 14 - Example of accelerator interface and connection to the PLM elements

The PLMs store the data structures needed for the computation. Mnemosyne-generated PLMs guarantees that the accesses have **fixed latency** (one clock cycle) by using multiple memory banks implemented with Intellectual Property (IP) blocks to offer multiple ports for concurrent accesses (e.g., FPGA BRAMs). Each physical bank is generally expressed as a vendor macro and requires a wrapper with the same interface and semantics described above. These wrappers must be generated only once for every new technology.

Mnemosyne optimizes the PLM architecture by analysing certain characteristics of the system. Mnemosyne takes as input information on the data structures to be stored in the PLMs, the compatibilities between these data structures, and details on the memory interfaces for each accelerator. The accelerator memory interfaces are used to automatically determine port direction and simplify integration. By using this information, Mnemosyne shares the physical memory banks whenever possible and generates RTL for this optimized memory architecture. The produced RTL is generated in Verilog (*REQ4.4*, D2.2). Currently, VHDL backend is not planned but this limitation does affect the integration with other components.

To identify sharing opportunities, Mnemosyne looks for **data structure compatibilities**. In many cases, data structures are not used throughout the entire duration of execution. While a data structure is unused, the memory IP it resides in is wasted space and could be used for storing a different data structure. To determine which data structures can share the same physical memory banks, their lifetimes must be known. The lifetime of a data structure is the interval between the first write and last read of the data structure. If two data structures have disjoint lifetimes, that means only one of them is active and valid at a time. This means they can fully share the same physical memory IPs and use the same address space. These data structures are considered *address-space compatible*. In some technologies, the area of one IP is smaller than the area of smaller individual IPs combined to store the same amount of data. Also, if a data structure is much smaller than the smallest available memory IP, the remaining memory space is wasted. In either case, storing two smaller data structures in one larger physical memory IP, if there are no port conflicts, would save this extra overhead area. If it can be confirmed that these two data structures are never read from at the same time or written to at the

same time, then they are *memory-interface compatible* and can be allocated within the same memory IP.

The metadata required as input to Mnemosyne are formatted as YAML files. This format affects the metadata produced by the compiler to pass memory-related information (*REQ3.10*, D2.2). The information about the data structures to be stored in the PLM are formatted as follows:

```
arrays:
   - name       : A0
     width      : 2048
    height     : 16
    interfaces : [w, r, r]
  - name       : [...]
    [...]
```

"`name`" is the name of the data structure, "`width`" is the bitwidth of each element, "`height`" is the number of elements to be stored, and "`interfaces`" is a list of all interfaces required by the accelerator to access the data structure (write must be listed first, then read/write, and lastly read). The compatibilities between the data structures (the **memory compatibility graph**) are formatted as follows:

```
nodes: [A0, B0, C0]
 edges:
   - compatibility : [A0, B0]
     type          : a
   - compatibility : [A0, C0]
     type          : b
  - compatibility : [...]
    [...]
```

"`nodes`" is the list of all data structures to be stored in the PLM (and must match the data structures described in the previous file) and "`edges`" describes each compatibility between a pair of compatible nodes. "`compatibility`" is a list of the two compatible nodes and "`type`" is the type of compatibility between them where "**`a`**" is an *address-space compatibility* and "**`b`**" is a *memory-interface compatibility*. If a pair of nodes is not listed, they are conservatively assumed to have no compatibility.

During EVEREST, Mnemosyne will be extended to include more memory components based on the information coming from the compilation flow. For example, it will automatically include a DMA engine for making the accelerator able to perform autonomous data transfers and prefetchers to hide the communication latency by anticipating data transfers. All components will then interact with the physical memory or ethernet controllers made available by the target platform. This integration will rely on standard interfaces, like AXI (*REQ4.14*, D2.2) and will be managed during system integration. Also, the

Mnemosyne-generated memory architectures will include variants that can perform data transfers in different ways (e.g., variable lengths of data bursts, format, or source-destination). The memory architecture will expose an additional set of input ports that will represent the configuration to be activated. These ports will be interfaced with the autotuning part for dynamic management (*REQ5.8*, D2.2).

Finally, security extensions will be added during the generation of the memory architecture, like automatic encryption/decryption of the memory data. The use of intelligent data management will allow for an automatic management of such operations that minimize the performance overhead.

5 Target platform and system integration

5.1 FPGA-based target platform

Using energy-efficient and heterogeneous platforms containing FPGAs are a main focus of EVEREST (among others *REQ9, D2.1, GREQ2, D2.2*). EVEREST targets two state-of-the-art research platforms that leverage the FPGAs with different design paradigms, both will be described in the following.

Those platforms will feature one or more FPGAs and one or more physical memories (either local or external to the FPGA), as shown in Figure 15. For both types of platform, at least one CPU host is required that run Linux as Operating System and the software (SW) part of the application. The application can communicate with the accelerated hardware (HW) kernels either via tightly-coupled OpenCAPI connection or via a loosely-coupled network (UDP/TCP/IP) connection, depending on the platform. In the latter case, the application may consist of multiple network-attached FPGAs. The first platform will be referred to as "OC-Accel", the latter as "cloudFPGA (cF)".

Both systems abstract the development of deployment of FPGA applications and therefore offer a high flexibility to the EVEREST consortium to account for interoperability and portability of the developed accelerated solutions to different platforms (even out of EVEREST's platforms). This abstraction is enabled by a predefined set of interfaces. To achieve portability for EVEREST workflows, the interfaces between the SW part of the application (left hand side of Figure 15) and the HW kernel (right hand side of Figure 15) will be based on a unified set of interfaces.

In the FPGA, the accelerated kernels in both platforms are interfaced solely via AXI channels. Both OC-Accel and cF provide a Memory Mapped I/O register access over an AXILite bus to pass runtime parameters, as well as a full AXI master bus to access the memory. CF also enables AXI-stream based access. Differences exists in the number of AXI master buses to connect the accelerators to the FPGA DRAM channels or HBM, depending on the specific FPGA platform. Please note, that the EVEREST approach is not limited to these platforms, because the specified HW/SW interfaces can be easily ported to other, similar FPGA platforms.

The OC-Accel logic and the cF logic will be referred to as platform specific "Shell". Those Shells implement all the necessary low-level processing of the communication or memory links respectively, to provide those AXI interfaces (cf. *REQ4.14, D2.2*). This way the compiler is only required to generate the accelerators with this higher-level interfaces. AXI interfaces are standardized and commonly used in the FPGA design ecosystem and supported by HLS tools.

Based on the selected platform, the compiler chooses the subset of necessary interface connections (e.g. with or without stream access, or with or without HBM). Due to this approach, the application can be generated independently of the final platform, to a large extend. Additionally, this also enables high reusability of compiler-backend modules developed for these FPGA target platforms (*REQ3.11, D2.2*).

Similarly, a set of unified interfaces will be provided at the SW side. This unified interface will be a wrapper to the platform specific APIs. As can be seen in the upper left corner of Figure 15, OC-Accel relies on the libcxl user-space and the ocxl kernel-space libraries. On top of them a C/C++ interface is provided and based on that, different language can bind to those libraries.

On the other side in the lower left corner of Figure 15, cF requires TCP/UDP sockets and needs to know the IP addresses and ports of the corresponding network-attached FPGA. This library is written in C++ and again it can be used by multiple language-binding tools. The cF platform also provides a development kit that contains build scripts, test cases and necessary SW/HW abstractions. This development library is referred to as "cloudFPGA Development Kit (cFDK)".

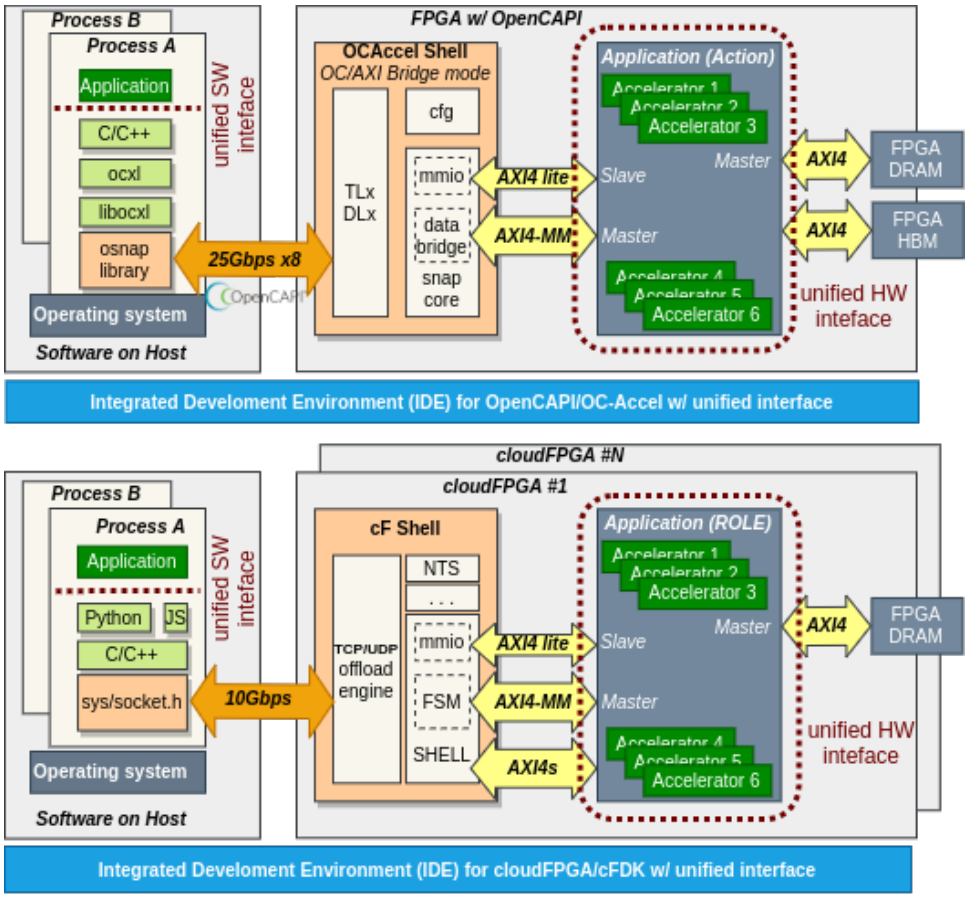

Figure 15 - EVEREST FPGA IDE with unified interfaces

5.2 System integration

5.2.1 Hardware integration

After generating the accelerator kernels and the memory architecture, the components must be interfaced with the rest of the system and the subsequent logic synthesis tools for bitstream generation. Synthesis options and constrained will be specified to the single HLS tools and memory generators (*REQ4.10*, D2.2) via command-line options and meta-data information.

Before logic synthesis and bitstream generation, it is necessary to create the complete system description that includes the generated components and additional IPs, like soft IPs to interface with the physical memory controller and the ethernet controllers, hardware monitors for runtime selection (*REQ5.9*, D2.2) and hardware API for exchange information virtualization environment (*REQ5.11*, D2.2). Such components will be described in a component library with standard formats (e.g., IP-XACT). Similarly, HLS-generated components will follow a standardized interface (*REQ4.7, REQ4.8,* and *REQ4.9, D2.2*) In the case of cloudFPGA, the generated system will be placed in the *Role* module and interfaced with the Shell module through standard interfaces (*REQ4.14*, D2.2).

To do so, EVEREST will develop **Olympus**, which is an automated system integration tool for creating such architecture. It will integrate all components, also creating parallel accelerator architectures from a high-level description. This is particularly important for the applications of the project that include massively parallel computation (e.g., WRF simulations).

Olympus generates a **memory system wrapper** for parallelization of the RTL modules of the Private Local Memory (PLM) architecture. From the total number of available resources and the resource estimates of the accelerator and the PLM architecture, maximum degrees of parallelism can be determined. Let *m* be the number of PLM architecture instances that can be instantiated on the platform (based on the available number of BRAMs and the requirements of the optimized Mnemosyne units), and *k* be the number of accelerators that can be instantiated (based on the available logic resources and the ones required after HLS). Olympus will generate an architecture with the following assumptions:

- *k* must be less than or equal to *m*, since an accelerator can only execute if it has a corresponding PLM architecture available.
- *k* can be less than *m* so that one accelerator can execute several iterations in sequence using several pre-loaded PLM architectures (in case, for example, the DMA engine and the prefetchers will be used to hide communication latency.

Depending on the values of *m* and *k*, Olympus will automatically generate an **RTL wrapper** which instantiates all memory modules for the entire system and connects them with the necessary control logic so that external memory access

is completely transparent. Similarly, Olympus will provide the necessary logic to handle streaming interfaces (*REQ3.4*, *REQ4.11*, D2.2), if needed.

Olympus will also perform **system partitioning** to divide the computation across multiple FPGA cards in a transparent way with respect to software allocation and execution (*REQ4.17*, D2.2). To do so, it generates the logic to interface with the Ethernet controller in case of off-chip data transfers from/to other boards and data exchanges with the host. This logic will hide the communication details to the accelerators and the local memory architectures.

Finally, Olympus will automatically generate **TCL scripts** to create the system descriptions and interface with the rest of the system (e.g., cloudFPGA Shell) and synthesis tools. The flow will generate single descriptions and bitstreams for each of the target boards.

5.2.2 Hardware-software interfacing

Olympus also simplifies and automates hardware/software integration by automatically generating the host code that interfaces with the generated hardware. Such code will be based on the low-level libraries provided for the target platform (e.g., the cloudFPGA Development Kit) where common functions (like data bursts and more complex/secure data transfers) will be abstracted in software libraries (to be used at the host side during code generation) and hardware IP components (to be used at the FPGA side during hardware integration). Such integration will be completely transparent to the application designers.

5.2.3 Hardware-software security flow

As discussed in Section 2.5, our goal is to provide security support for inter and intra-node mechanisms. Application designers should annotate the communication between software components (e.g., in a workflow or dataflow graph) to specify the required security functionality at the inter-node level, to define the nodes.

The envisioned automation flow to extend the SW-HW system with required security capabilities is depicted in Figure 16. The EVEREST platform will be extended with a library with security primitives that are needed to ensure the required security properties (developed in WP3). The library will include primitives implemented in hardware (at RTL level), in software (that can be used in the system as software, but they can even be the starting point for generating hardware using high level synthesis), and will be connected with the rest of the system by means of a standardized interface, such as AXI. Regardless of the implementation, security components will be integrated in the system in a transparent way for the rest of the application (or, at least, in the most transparent way possible). Furthermore, the designer must be able to specify

the composition of a logic node, simply instantiating the elements that compose it. The first step of the flow is the creation of logical nodes when the logic node is defined. To do so, the primitives needed to create the virtual enclave will be automatically selected from the ones included in the library and the node architecture will be extended with them. Where needed, communication will be re-routed to make use of the instantiated primitives. In the second step, we will handle the inter-node security requirements. In this case, the most appropriated security primitive will be automatically selected based on the policies defined in WP3 from the ones present in the library and the node architecture will be extended with the necessary components (either software or hardware). The selection of the primitives, in this case, will be based on the inter-node requirements specified, for instance, with the dataflow DSL (cf. Section 3.1). The flow will then continue with the other steps.

For the EVEREST use cases, the specification of the requirements will be done according to the requirements collected in D2.3: all the applications require the enforcement of inter-node data confidentiality and authentication and, in some cases, also the tracking of the information flow. Intra-node security functionalities will be provided to ensure the maximum flexibility in allocation of node resources to all the application.

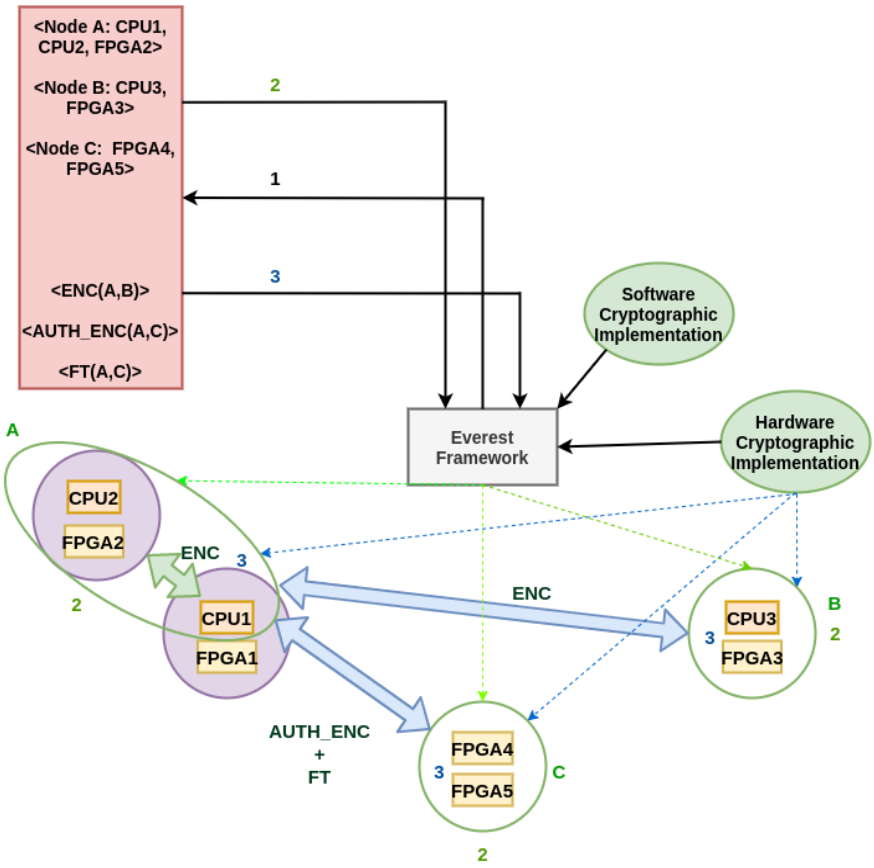

Figure 16 - Envisioned flow for providing encryption, authentication, and information flow tracking in EVEREST.

To exemplify the concept, let's begin to realize the nodes at the application level. For instance, for the traffic prediction model, we need to run specific machine learning algorithms on distributed resources including FPGAs and CPUs. Let's imagine that, within that application, we need a logical node that includes FPGAs and CPUs, and that they should be completely isolated from the rest of the resources. Since the algorithm will be executed on FPGAs and CPUs, our node will be formed by a logic enclave containing FPGAs and CPUs. Regardless of their physical placement, these computing elements will be logically isolated by the appropriated cryptographic primitives from the library. The definition of enclaves shall be integrated with the resource definition Hyperloom/HyperTools (cf. Section 3.1.1).

Once the nodes are defined, we need to ensure the security of the communications, for instance, the confidentiality between the traffic simulator and the traffic prediction model. Similar to what is done for defining the nodes, we will define a communication channel between the traffic simulator and the traffic prediction module. The communication channel definition will include the security functionalities that needs to be ensured (in our case confidentiality). From this, the hardware modules needed to ensure confidentiality will be automatically inferred and instantiated in the architecture.

6 Compilation Technologies: Advancing the State of the Art

The previous sections of this deliverable list several technologies, with focus on the frameworks, languages, compilers, intermediate representations, high-level synthesis and hardware generation tools that will serve as the basis for the compilation flow in the EVEREST SDK. To make the contributions to the state of the art more explicit, this section provides details on the original status of the technologies, the foreseen extensions thereof, and envisioned new technologies to be developed within EVEREST. Since the project is ongoing, the list provided here is by no means exhaustive. We also discuss how our contributions complement and, if applicable, compare to existing commercial tools.

6.1 Technologies Overview

Table 1 presents an overview of the different components of the compilation framework. For each component (Tool column), the table names the main lead and whether the tool originates from a partner of the consortium or if it is maintained by an external entity (Source column). The next two columns express whether the tool will be completely developed within the EVEREST project (New column), or if it already existed and was extended in the project (Extended column). In either case, the table describes the major features these components will have. The final column of the table lists the major tool flow in which the component will be integrated at first.

While the table provides a tool-specific view, it is to be noted, that a major contribution of EVEREST lies in building a framework in which these tools can seamlessly operate to provide a transparent and efficient use of the EVEREST platform. Only by investigating tool interfaces, identifying possible mismatches between tool abstractions and finally building an end-to-end tool flow in such a collaborative approach can one achieve the holistic approach envisioned by EVEREST.

The description of the tools in the table is kept succinct, since they are the matter of ongoing research within the project. Details on the tools themselves, the new features and how they are implemented will be provided in subsequent deliverables, most notably, in Deliverable 4.2.

The first three entries of the table correspond to the languages and intermediate languages defined in Section 3. For instance, TeIL-mlir will integrate into the MLIR language stack as described in Figure 8. Most likely, other smaller dialects will be needed to implement abstract scalars types that can be then mapped to custom number representations. DOSA is a framework that shall automatically select ML operators provided by existing frameworks so as to improve the system efficiency while providing support for distributed execution across FPGAs.

As discussed above, we will initially rely on the TVM framework to manipulate the ML model.

WRF is an external framework that we will extend to be able to extract kernels that can be then offloaded to reconfigurable accelerators. This includes modifications to the build system, to the interfaces of modules and a new testing infrastructure. As mentioned in Section 2.2, we will initially focus on the radiation module. To expose more parallelism, we plan to extend the default WRF setup with a the recently proposed RTE RTMGP module.

As discussed in Section 4, Bambu and Mnemosyne are two tools developed by partners of EVEREST that will be used for hardware generation. In contrast to many other tools, Bambu is an open-source HLS tool. Within EVEREST, Bambu will be extended to receive MLIR, allowing for a tighter interaction between the DSL compiler and the HLS flow. To demonstrate interoperability, the compilation framework will also support downstream commercial HLS tools, such as Vitis/Vivado from Xilinx/AMD. Mnemosyne will be extended to receive high-level buffer life-time information from the DSL compiler. Since the DSL generates code for kernels, and these kernels can be generated to work on different data granularities, a new tool called Olympus will be designed that composes accelerators, sets up the HW interfaces and ultimately produces the final design.

Finally, for network-attached FPGAs, we will rely and extend the cloudFPGA SDK (cFDK). This will initially focus on providing support for the ML workload of the EVEREST use cases.

Table 1. Summary of the technologies in the EVEREST compilation framework

Tool	Lead	Source	New	Extended	New features	Validated in
TeIL-mlir	TUD	Internal	Yes	Yes	MLIR implementation of the TeIL language specification, domain optimizations, optimizer driver	Integration and unit tests; DSL-to-FPGA flow
CFDlang	TUD	Internal	No	Yes	New compiler infrastructure based on MLIR, extended primitives, implemented new intermediate language, new frontend, new optimizations, extended language support	DSL-to-FPGA flow
Ohua	TUD	Internal	No	Yes	Improved support for Rust and Python, implemented dataflow transformations for batching, state update and roll-back	PTDR flow
DOSA	IBM	Internal	Yes	Yes	all (partitioning of ONNX, analysis and reuse of existing libraries, automatic system generation, improved heterogeneous communication library)	End-to-end analysis of ML flow

TVM	IBM	External	No	Yes	Minor improvements in ONNX parsing	End-to-end analysis of ML flow
WRF	CIMA	External	No	Yes	Isolation of the RRTMG radiation driver, Modified allocation strategy, testing infrastructure	WRF flows, stand-alone validation, DSL-to-FPGA flow
RTE RTMGP	TUD	External	No	Yes	Interfacing external module that exposes more parallelism into the WRF simulation	WRF flows
Bambu	PDM	Internal	No	Yes	Custom floating-point support, loop pipelining, MLIR interfacing	DSL-to-FPGA flow and ML Flow
Mnemosyne	PDM	Internal	No	Yes	Automatic generation of the input metadata	DSL-to-FPGA flow
Olympus	PDM	Internal	Yes	Yes	Automatic generation of the system/memory architecture	DSL-to-FPGA flow
Vitis HLS	Xilinx	External	No	No	None	DSL-to-FPGA flow and ML Flow
cFDK	IBM	Internal	No	Yes	Stabilization of platform, improvement of networking, new build flow, new debugging flow, more tools to support the user, improved documentation	ML flow

6.2 Added Value to Commercial Tools

As mentioned multiple times, the main goal of the EVEREST SDK is to integrate technologies so as to provide a better end-result than with fragmented tools. This includes aspects such as increased productivity, i.e., making reconfigurable hardware accessible to application experts, and system efficiency, i.e., making it possible for the framework to explore configurations that are not possible with existing (commercial) tools.

Concretely in the case of commercial tools, we make explicit how the framework can rely on tools like Vitis (cf. Figure 11). For instance, in [18] we show that by implementing high-level transformations in the DSL compiler, not available to the HLS tools, the accelerator's throughput increases from around 3 GFLOPs to 43 GFLOPs.

DOSA builds atop existing ML frameworks such as hls4ml, haddoc2, FINN VitisAI/DPU and Vitis AI/custom, by automatically selecting the best operator for a given FPGA. With the exception of VitisAI/DPU, DOSA is the only framework that supports distributed model execution on FPGAs. In contrast to FINN, DOSA does so with a set of scripts, requiring no user intervention. In terms of ease of use, DOSA is as easy to use as hls4ml which is limited to a single FPGA.

From the tools presented in Table 1, Bambu does have an overlap with commercially available tools. Bambu, like many others, supports operator chaining, bit-width analysis and optimization, memory space allocation, speculation and code motion [19], and if-conversion transformations. It also supports spatial parallelism through OpenMP annotations [16]. Moreover, it is the only HLS tool that can start both from the intermediate representations generated by Clang and GCC. More importantly for this project, Bambu accepts MLIR and LLVM as input. While XILINX recently open-sourced their Clang/LLVM front-end, it is well-documented by developers that the subset of the accepted LLVM IR is restricted. The MLIR entry point of Bambu has been already leveraged by the Soda-opt compiler [20]. In terms of quality of results, as shown in [21], Bambu is on par or better than a well-known commercial tool, LegUp (later commercialized), and to the academic HLS tool Dwarf.

7 Conclusions

In this deliverable we have defined how to architect the complex compilation framework to transparently and automatically compute efficient execution implementations for the use cases on the EVEREST Platform. We have shown how language support, frameworks extensions, novel intermediate languages and transformations in source-to-source compilers, and extension to HLS tools and memory generators can seamlessly interoperate to produce efficient HW and SW implementations. This document described the concrete implementation plans in work package WP4 and provided early insights into the interaction with work packages WP3 and WP5. Stronger connections to these work packages will be made, as the use case implementation advances and the runtime environment is brought up.

8 References

[1] V. Cima *et al.*, "HyperLoom: A Platform for Defining and Executing Scientific Pipelines in Distributed Environments," in *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '18*, Manchester, United Kingdom, 2018, pp. 1–6. doi: 10.1145/3183767.3183768.

[2] A. Susungi, N. A. Rink, A. Cohen, J. Castrillon, and C. Tadonki, "Meta-programming for cross-domain tensor optimizations," in *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, Boston MA USA, Nov. 2018, pp. 79–92. doi: 10.1145/3278122.3278131.

[3] N. A. Rink and J. Castrillon, "TeIL: a type-safe imperative tensor intermediate language," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming - ARRAY 2019*, Phoenix, AZ, USA, 2019, pp. 57–68. doi: 10.1145/3315454.3329959.

[4] C. Lattner *et al.*, "MLIR: A Compiler Infrastructure for the End of Moore's Law," *ArXiv200211054 Cs*, Feb. 2020, Accessed: Mar. 26, 2021. [Online]. Available: http://arxiv.org/abs/2002.11054

[5] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, USA, 2018, pp. 579–594.

[6] *Apache TVM*. [Online]. Available: https://tvm.apache.org

[7] D. Gadioli, E. Vitali, G. Palermo, and C. Silvano, "mARGOt: A Dynamic Autotuning Framework for Self-Aware Approximate Computing," *IEEE Trans. Comput.*, vol. 68, no. 5, pp. 713–728, May 2019, doi: 10.1109/TC.2018.2883597.

[8] S. Ertel, C. Fetzer, and P. Felber, "Ohua: Implicit Dataflow Programming for Concurrent Systems," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, Melbourne FL USA, Sep. 2015, pp. 51–64. doi: 10.1145/2807426.2807431.

[9] J. Launchbury and S. L. Peyton Jones, "Lazy functional state threads," *ACM SIGPLAN Not.*, vol. 29, no. 6, pp. 24–35, Jun. 1994, doi: 10.1145/773473.178246.

[10] S. Ertel, A. Goens, J. Adam, and J. Castrillon, "Compiling for concise code and efficient I/O," in *Proceedings of the 27th International Conference on Compiler Construction*, Vienna Austria, Feb. 2018, pp. 104–115. doi: 10.1145/3178372.3179505.

[11] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–29, Oct. 2017, doi: 10.1145/3133901.

[12] N. Vasilache *et al.*, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *ArXiv180204730 Cs*, Jun.

2018, Accessed: Jul. 22, 2021. [Online]. Available: http://arxiv.org/abs/1802.04730

[13] N. A. Rink, A. Susungi, J. Castrillon, J. Stiller, and C. Tadonki, "CFDlang: High-level code generation for high-order methods in fluid dynamics," in *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*, Vienna, Austria, 2018, pp. 1–10. doi: 10.1145/3183895.3183900.

[14] *PandA/Bambu - A framework for Hardware-Software Co-Design of Embedded Systems*. [Online]. Available: https://panda.deib.polimi.it/

[15] Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis." [Online]. Available: http://xilinx.com

[16] M. Minutoli *et al.*, "Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics," *IEEE Trans. Comput.*, pp. 1–1, 2021, doi: 10.1109/TC.2021.3057860.

[17] *Mnemosyne: Multi-Bank Memories for Heterogeneous Architectures*. Politecnico di Milano and Columbia University. [Online]. Available: https://github.com/chrpilat/mnemosyne

[18] S. Soldavini, K. F. A. Friebel, M. Tibaldi, G. Hempel, J. Castrillon, and C. Pilato, "Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics." arXiv, Jun. 03, 2022. Accessed: Jun. 28, 2022. [Online]. Available: http://arxiv.org/abs/2203.10850

[19] M. Lattuada and F. Ferrandi, "Code transformations based on speculative SDC scheduling," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2015, pp. 71–77. doi: 10.1109/ICCAD.2015.7372552.

[20] N. B. Agostini, S. Curzel, D. Kaeli, and A. Tumeo, "SODA-OPT an MLIR based flow for co-design and high-level synthesis," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, Turin Italy, May 2022, pp. 201–202. doi: 10.1145/3528416.3530866.

[21] R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016, doi: 10.1109/TCAD.2015.2513673.