http://www.everest-h2020.eu

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

D4.5 — Final report 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	42 months
Project Website	http://www.everest-h2020.eu

Copyright

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

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

Scientific Coordinator: Christian Pilato - Politecnico di Milano

Disclaimer

Deliverable Information

Work-package	WP4
Deliverable No.	D4.5
Deliverable Title	Final report of the compilation framework
Lead Beneficiary	TUD
Type of Deliverable	Report
Dissemination Level	Public
Due Date	31/01/2024

Document Information

Delivery Date	24/06/2024
No. pages	55
Version | Status	0.7 | Final
Responsible Person	Jeronimo Castrillon (TUD)
Authors	Jeronimo Castrillon (TUD), Karl A. F. Friebel (TUD), Felix Suchert (TUD), Burkhard Ringlein (IBM), Serena Curzel (PDM), Michele Fiorito (PDM), Fabrizio Ferrandi (PDM), Donatella Sciuto (PDM), Christian Pilato (PDM), Stephanie Soldavini (PDM)
Internal Reviewer	Francesco Regazzoni (USI)

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
24.11.2023	0.1	Jeronimo Castrillon (TUD)	Initial structure.
04.01.2024	0.2	Jeronimo Castrillon (TUD)	Initial text, revisited structure and refined outline of the chapters.
09.02.2024	0.3	Jeronimo Castrillon (TUD)	Cleaned up remaining issues in Sections 1–6. First draft of conclusions.
08.03.2024	0.4	Jeronimo Castrillon (TUD)	Added tool links, requirement assessment and comparison to the state of the art.
14.03.2024	0.5	Christian Pilato (PDM)	Revision of hardware generation flow and general clean up.
15.03.2024	0.6	Jeronimo Castrillon (TUD)	Final clean up and approval.
24.06.2024	0.7	Jeronimo Castrillon (TUD)	Addressed feedback from EC reviewers.

Quality Control

Approved by Internal Reviewer	March 18, 2024
Approved by WP Leader	March 15, 2024
Approved by Scientific Coordinator	March 15, 2024
Approved by Project Coordinator	March 18, 2024

Table of Contents

1 Executive Summary 6
1.1 Structure of this Document . . . 6

2 Overview of the Compilation Framework 7

3 Language Support 11
3.1 EVEREST Kernel Language . . . 11
3.2 Fortran Integration for WRF . . . 12
3.3 Dataflow: Ohua/Condrust . . . 13
3.4 Machine Learning . . . 14

4 Intermediate Representations and Transformations 15
4.1 `ekl` . . . 16
4.2 `esn` . . . 16
4.3 `dfg` . . . 17
4.4 `evp` . . . 19
4.5 `olympus` . . . 20
4.6 Connecting to High-Level Synthesis (HLS) . . . 22
4.7 Machine Learning Abstractions and Optimization . . . 22
4.7.1 Operation Set Architectures . . . 22
4.7.2 `DOSA`: Distributed Operation Set Architectures using organic compiler principles 24

5 Hardware Generation Flow 26
5.1 Hardware-Oriented Optimizations and Kernel Generation . . . 26
5.1.1 Loop Pipelining . . . 27
5.1.2 Custom Precision Floating-point Data Types . . . 28
5.2 Partitioning to Distributed FPGAs . . . 29
5.3 Memory-Related Optimizations . . . 30
5.4 System Integration . . . 31
5.4.1 Network-attached Field Programmable Gate Arrays (FPGAs) . . . 31
5.4.2 Bus-attached FPGAs . . . 32

6 Code Generation and Runtime Integration 36
6.1 Host Code Generation . . . 36
6.2 EVP: The Offloading Linker . . . 36
6.3 Bundling Variants with `basecamp Climbs` . . . 37

7 Final Assessment 40
7.1 Requirements: Language and Compilers . . . 40
7.1.1 REQ3.1 – WRF expression abstraction . . . 40
7.1.2 REQ3.2 – WRF Fortran integration . . . 40
7.1.3 REQ3.3 – ML integration . . . 40
7.1.4 REQ3.4 – Streaming support . . . 40
7.1.5 REQ3.5 – Integration with compiler frameworks . . . 41
7.1.6 REQ3.6 – Compiler transformations for kernels . . . 41
7.1.7 REQ3.7 – Compiler transformations for dataflows . . . 41
7.1.8 REQ3.8 – Multi-target code generation . . . 41
7.1.9 REQ3.9 – Generation of tunable parameters . . . 42
7.1.10REQ3.10 – Interface to HLS . . . 42
7.1.11REQ3.11 – cFDK/OC-Accel software integration and language compatibility . 42
7.1.12REQ3.12 – Reasoning about heterogeneous applications . . . 42
7.1.13REQ3.13 – Glue code generation for heterogeneous applications . . . 43
7.1.14REQ3.14 – Abstractions for offloaded kernels . . . 43

7.2 Requirements: HLS and Memory Design 43
7.2.1 REQ4.1 – C/C++ support 43
7.2.2 REQ4.2 – Bambu LLVM IR support 43
7.2.3 REQ4.3 – Bambu MLIR dialect support 44
7.2.4 REQ4.4 – HLS Verilog output 44
7.2.5 REQ4.5 – HLS VHDL output 44
7.2.6 REQ4.6 – Top function specification 44
7.2.7 REQ4.7 – Block level/Top component interfaces 44
7.2.8 REQ4.8 – Port-Level interfaces 45
7.2.9 REQ4.9 – Bambu Vivado HLS I/O interface interoperability 45
7.2.10 REQ4.10 – Technology options specification 45
7.2.11 REQ4.11 – Bambu Data flow annotations 45
7.2.12 REQ4.12 – Bambu OpenMP support 45
7.2.13 REQ4.13 – Bambu floating point precision 46
7.2.14 REQ4.14 – cFDK/OC-Accel top component interface 46
7.2.15 REQ4.15 – Memory interfaces 46
7.2.16 REQ4.16 – Software-level support 46
7.2.17 REQ4.17 – Hardware/software data sharing 46
7.3 Advancing the State-of-the-art 47

8 Conclusions 49

References 52

1 Executive Summary

The EVEREST project proposes a platform and System Development Kit (SDK) to deploy demanding workflows to suitable high-performance or edge hardware [25, 24]. This document provides a final report on the compilation framework of the SDK. The compilation framework plays a key role in providing high-level programming support for productivity together with a methodology for optimization. The latter includes software and hardware transformations as well as autotuning support for runtime adaptivity.

In this document, we describe the technologies, tools, and components of the compilation framework and assess how they contribute to fulfilling the EVEREST requirements described in Deliverable D2.2 and Deliverable D2.5. Details on how to use the tools are provided in Deliverable D4.6. The design presented here follows the specification provided in Deliverable D4.1, which depicted a complex landscape of programming languages (Fortran, Rust, C++, Python) and computational requirements (large workflows combining High-Performance Computing (HPC), Big Data and machine learning). In this document we describe the language abstractions, high-level representations and hardware-oriented transformations for computational motifs that are representative of those found in the use cases of the EVEREST project. We use examples to demonstrate the different representations and transformations enabled in the EVEREST SDK. Use-case specific descriptions are a matter of Deliverable D6.3.

1.1 Structure of this Document

This document starts with an overview of the compilation framework in Section 2. The overview details the evolution of the SDK, connecting to previous project deliverables, and describes *basecamp*, a unified tool that provides access to the multiple tools within the SDK. Section 3 summarizes contributions to Domain-Specific Languages (DSLs) and the integration into Fortran for HPC use cases. A major contribution of the SDK is a collection of Intermediate Representations (IRs) that enable reuse across domains. These IRs, implemented in Multi-Level Intermediate Representation (MLIR), are described in Section 4 along with sample transformations. Hardware programming and integration flows are described in Section 5. This includes extensions to HLS flows and system-level integration support for the EVEREST platform. The presented flow is concluded with Section 6, which describes how the components in this document integrate with the dynamic runtime environment. In Section 7, we provide a final assessment concerning the requirements in Deliverable D2.5 and a high-level comparison with the state of the art. This deliverable finishes with conclusions in Section 8.

2 Overview of the Compilation Framework

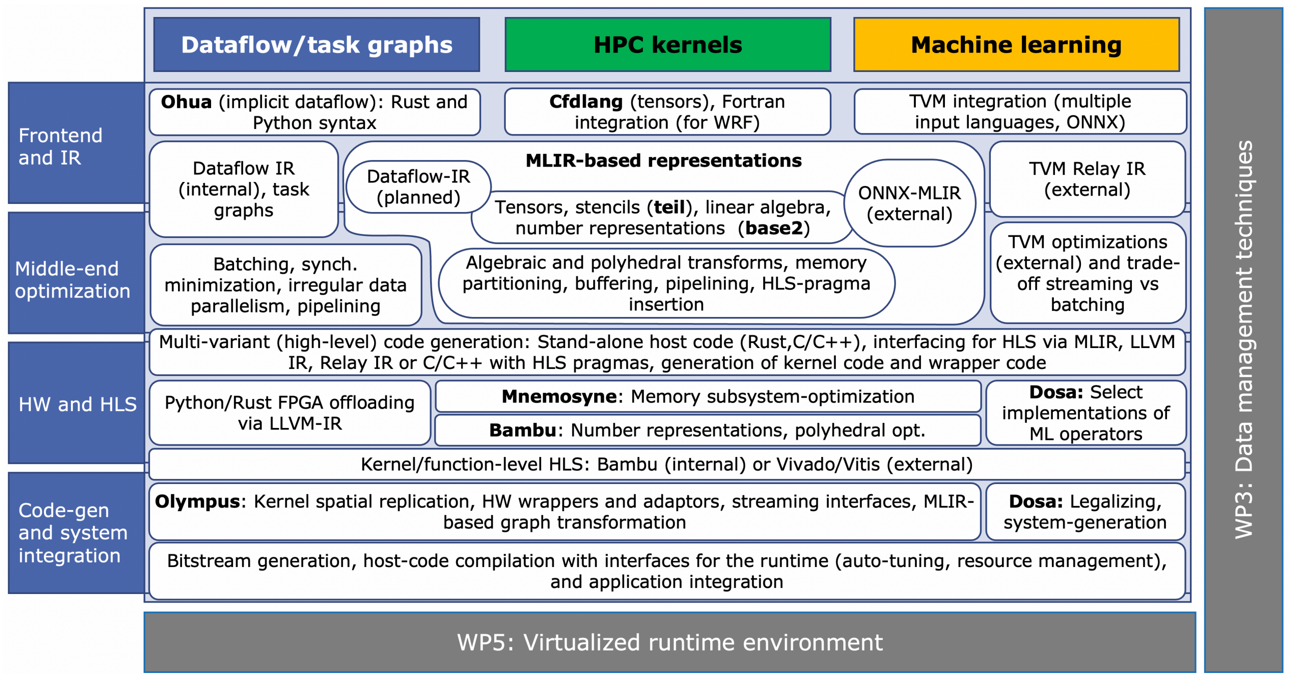

Dataflow/task graphs
HPC kernels
Machine learning
Frontend and IR
Middle-end optimization
HW and HLS
Code-gen and system integration
Ohua (implicit dataflow): Rust and Python syntax
Cfdlang (tensors), Fortran integration (for WRF)
TVM integration (multiple input languages, ONNX)
Dataflow IR (internal), task graphs
MLIR-based representations
Dataflow-IR (planned)
Tensors, stencils (teil), linear algebra, number representations (base2)
ONNX-MLIR (external)
TVM Relay IR (external)
Batching, synch. minimization, irregular data parallelism, pipelining
Algebraic and polyhedral transforms, memory partitioning, buffering, pipelining, HLS-pragma insertion
TVM optimizations (external) and trade-off streaming vs batching
Multi-variant (high-level) code generation: Stand-alone host code (Rust,C/C++), interfacing for HLS via MLIR, LLVM IR, Relay IR or C/C++ with HLS pragmas, generation of kernel code and wrapper code
Python/Rust FPGA offloading via LLVM-IR
Mnemosyne: Memory subsystem-optimization
Bambu: Number representations, polyhedral opt.
Dosa: Select implementations of ML operators
Kernel/function-level HLS: Bambu (internal) or Vivado/Vitis (external)
Olympus: Kernel spatial replication, HW wrappers and adaptors, streaming interfaces, MLIR-based graph transformation
Dosa: Legalizing, system-generation
Bitstream generation, host-code compilation with interfaces for the runtime (auto-tuning, resource management), and application integration
WP5: Virtualized runtime environment
WP3: Data management techniques

Figure 1 – Overview of the compilation flow.

Previous reports detail the evolution of the compiler tooling developed within the EVEREST project. Most notably, the evolution can be identified in Figure 1 of Deliverable D4.1, the examples of the Alpha Release in Deliverable D4.3 and in Figure 1 of Deliverable D4.2, which is replicated in Figure 1 for better reference. This figure reflects the heterogeneity of the requirements identified early in the project, with (i) different types of workloads (dataflow/task graphs, HPC kernels and machine learning), (ii) different languages (Fortran, C/C++, ONNX, Rust, Python and DSLs), (iii) rich interfaces (via intermediate code in LLVM or MLIR, via generated C/C++ or Rust code and other metadata for variant generation), and (iv) the integration with multiple external tools and frameworks (WRF build system, TVM, the MLIR infrastructure, Vivado/Vitis HLS tools.

Today, the complexity depicted in Figure 1 is shielded from developers by a simple tool interface that we call *basecamp*. As shown in Figure 2, basecamp provides a common interface to pass applications to the compilation flow (see format and example at the bottom). Basecamp also interfaces with the deployment flow in EVEREST with leverages the LEXIS platform[1], which has been extended to offload the execution of selected kernels on FPGAs. It also contains interfaces to the runtime system (see Deliverable D5.5). In this way, if an accelerated task requires more resources, the EVEREST runtime can adapt the computation accordingly.

Basecamp offers a command line interface (CLI) and a python Application Programming Interface (API), whereas the former is a wrapper for the latter. The general CLI structure is shown at the bottom of Figure 2. In the given example, the dataflow tools are invoked and the result is written to the `build/` directory. In that example, basecamp acts as a unified wrapper for the different optimization flows of the SDK. However, basecamp can also bundle different optimized variants of one application, i.e., different outputs of basecamp flows, together into one executable application, as described in Section 6. The complete CLI of basecamp is given in Listing 1.

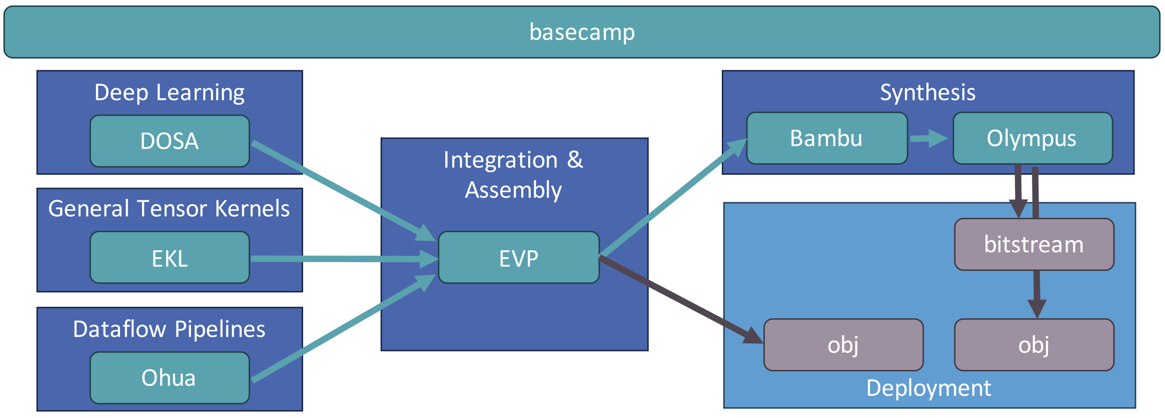
```
$ ebc-cli --help
EVEREST basecamp -- the basis for all EVEREST endeavors.

Usage:
    ebc-cli dataflow <input-file> -o <path-to-output> --target <target> --threads <num> --enable-↩
        parallelism <bool> --c-limit <num> --amorphous <bool>
    ebc-cli hpc [--lang <lang-id> --pipeline <pipeline> (-D <define>...) (-I <include>...)] <input↩
        > -o <path-to-output>
```

[1] https://lexis-project.eu/web/lexis-platform/

```
    ebc-cli ml_inference onnx|torchscript (--json-constraints <path-to-json-constraints> | --app-↩
        name <app-name> --target-throughput <target-sps> --batch-size <batch-size> --↩
        used_bit_width <used-bit-width> --onnx-input-name <onnx-input-name> --onnx-input-shape <↩
        onnx-input-shape>) [--map-weights <path-to-weights-file>] <path-to-model.file> <path-to-↩
        output-directory> [--calibration-data <path-to-calibration-data>]
    ebc-cli climbs (describe --flow <flow> | create --name <name> <path-to-file.climb> | ↩
        add_module --module <path-to-module.section> <path-to-file.climb> | add_file --file <path-↩
        to-source.file> --language <language> <path-to-file.climb> | emit <path-to-file.climb> [--↩
        output-directory <path-to-output-directory>])
    ebc-cli airflow ( create | get_params | get_state | execute [--params-json-path <path-to-json-↩
        params>]) <workflow-name>
    ebc-cli -h|--help
    ebc-cli -v|--version

Commands:
    dataflow                                          Invoke the dataflow flow of the EVEREST SDK.
    hpc                                               Invoke the HPC flow of the EVEREST SDK.
    ml_inference                                      Invoke the ML inference flow of the EVEREST ↩
        SDK.
    climbs                                            Combine different flows (i.e., "everest ↩
        climbs") to one application.
    airflow                                           Allows the execution of Airflow workflows ↩
        via Py4Lexis.

Options:
    -h --help                                         Show this screen.
    -v --version                                      Show version.

    -o <path-to-output>                               Path to save generated files under (defaults↩
         to `generated`).
    --target <target>                                 Target for the code generator (supported ↩
        values: rust, mlir).
    --threads <num>                                   Number of threads to parallelize for (↩
        default: number of local cores).
    --enable-parallelism <bool>                       Whether to enable the parallelization ↩
        optimization (defaults to `true`).
    --c-limit <num>                                   Number of maximum collisions for computation↩
         with amorphous data parallelism.
    --amorphous <bool>                                Whether to enable the transformation of ↩
        amorphous data parallel tasks (defaults to `false`).
    --json-constraints <path-to-json-constraints>     Import the ML target constraints of the ↩
        given JSON file.
    --app-name <app-name>                             The name of the target application (to ↩
        create human-readable labels).
    --target-throughput <target-sps>                  The targeted throughput (in samples-per-↩
        second (sps) of the inference application.
    --batch-size <batch-size>                         The used batch size per inference request (i↩
        .e. sample).
    --used_bit_width <used-bit-width>                 The bit width for input, activations, and ↩
        weights.
    --onnx-input-name <onnx-input-name>               The name of the input node in the ONNX graph↩
        .
    --onnx-input-shape <onnx-input-shape>             The input shape in the ONNX graph.
    --map-weights <path-to-weights-file>              The file containing the weights for the ↩
        kernel-weight-mapping schema.
    --calibration-data <path-to-calibration-data>     Point to the .npy file containing example ↩
        data to calibrate transformation to quantized data types.
    describe                                          Describe the required API for the flow.
    --flow <flow>                                     Specifies the flow to describe.
    create                                            Create a new flow.
    add_module --module                               Add a new application variant to an existing↩
         climb.
    add_file --file                                   Add a file (with annotations) of the main ↩
        application to the climb.
    --language <language>                             The language of the added file. Currently ↩
        supported are: python, docker, copy. (Copy means the file will be copied without change.)
    emit                                              Emit created climb to build directory.
    create                                            Create a new Airflow workflow.
    get_params                                    Get the current parameters of a workflow.
    get_state                                     Request the current state of a workflow.
    execute                                       Trigger the execution of a workflow.
    --params-json-path <path-to-json-params>      Optional update of workflow parameters for ↩
        execution.

Copyright EVEREST Consortium, licensed under the Apache License 2.0.
For contact and more details, please visit: https://everest-h2020.eu
```

Listing 1 – CLI of the everest basecamp tool. The cli is a wrapper of the python API, discussed in Section 6.

As will be described in this document, the compiler infrastructure for the different workloads, the integration, and assembly, as well as the interfacing to HLS tools are handled, for the most, within a common framework implemented in MLIR (see, for instance Figure 8). This effectively contributes to the *unification* of methods, which was one of the original goals of the EVEREST SDK to enable the convergence of data-centric applications involving HPC, Big Data, and machine learning.



```
# Format: ebc-cli <flow-select> <flags> <inputs> <outputs>
> ebc-cli dataflow mma.mlir --target="mlir" -o "build/"
```

Figure 2 – Basecamp, an interface to interact with the compilation flows in the EVEREST SDK.

This deliverable describes the final design of the DSLs in Section 3. We describe the adaptations to the Ohua DSL to cater for implicit dataflow applications, like the routing algorithm described in Deliverable D2.1, with extensions to support hardware offloading; the basic Cfdlang DSL for tensor kernels in HPC, like in Computational Fluid Dynamics (CFD) applications in Deliverable D2.1 and new abstractions to support the expressions identified in the radiation module of WRF; and how we leverage the prominent TVM framework for machine learning for decision-making in multiple use cases as described in Deliverable D2.1.

The MLIR-based intermediate languages to support the different workloads in the EVEREST project are described in Section 4. This includes novel frontend dialects, intermediate dialects for generalized Einstein notation, dataflow and system-level integration as well as for custom number representations. Section 4 also describes transformations and lowering passes within the MLIR stack. This includes typical coarse-grained optimizations for dataflow, algebraic and polyhedral optimizations for mathematical kernels in HPC, and standard optimizations for deep neural networks. Software-only versions can be produced after the optimization that can run on standard CPU nodes. Multiple such versions can be passed to the mARGOt autotuner for further optimization at runtime.

For nodes with reconfigurable hardware, the compiler middle-end can produce different interfaces and descriptors for the hardware generation part (*HW and HLS* in Figure 1). From the Rust backend of the dataflow language, LLVM IR can be generated for the Bambu HLS tool. For the HPC kernels, the compiler can interface directly via MLIR or LLVM with Bambu, or generate C/C++ code with HLS pragmas for Vitis/Vivado or Bambu. All this process is orchestrated and handled by EVP (see Figure 2). Machine learning applications are processed within the DOSA framework. The DOSA framework intelligently selects the best implementation for the operators used in the deep neural network. After this phase, multiple different Register Transfer Level (RTL) implementations for standalone kernels, functions (nodes in the dataflow graph) or machine learning operators are generated via HLS using either Bambu or Vivado/Vitis.

The final phase of the compilation framework (code-gen and system generation in Figure 1 is responsible for creating the entire system on the FPGA(s), performing the HW-SW integration and finally generating the binaries and bitstreams. This system-level generation is handled by Olympus (see Figure 2) as will be described in Section 5. Olympus also performs several optimizations to effectively use the FPGA resources and balance computation and communication time (especially relevant for systems with multiple memory channels). It also allows for the integration of kernels generated with different flows (e.g., different HLS tools or manual RTL blocks), enhancing the interoperability of the whole SDK. If an application requires more than one network-attached FPGA node, DOSA and the ZRLMPI tool handle the system generation. Here, the inter- and intra-FPGA communication is derived and optimized, and the FPGA design is generated. Links to the main tools of the compilation framework can be found in Table 1. For the final release, tool names and paths may change.

Tool	URL
DOSA	https://github.com/cloudFPGA/DOSA
ub	https://github.com/KFAFSP/ub-mlir
base2	https://github.com/KFAFSP/base2-mlir
cfdlang	https://github.com/everest-h2020/messner
Ohua/Condrust	https://github.com/ohua-lang/condrust
dfg-mlir	https://github.com/Feliix42/dfg-mlir
Bambu	https://github.com/ferrandi/PandA-bambu
Olympus	https://github.com/StephanieSoldavini/olympus
basecamp	https://github.com/everest-h2020/everest-basecamp

Table 1 – Links to main open source tools in the compilation flow

3 Language Support

As specified in the project plan, Deliverable D4.1 and Deliverable D4.2, the EVEREST SDK provides support for kernels in HPC, machine learning and dataflow computing. This section describes the final versions of the languages, building on the preliminary versions described in Deliverable D4.2. This section also explains how the languages integrate within the use cases.

3.1 EVEREST Kernel Language

In Deliverable D4.1, we chose the CFDlang language [36] as the starting point for the EVEREST HPC workflow. This DSL was originally designed for CFD applications and later extended for cross-domain tensor expressions in [42]. It was chosen for its modeling of regular linear algebra, providing the freedom needed to implement data management techniques as described in Deliverable D3.1.

In Deliverable D4.2, we detailed how CFDlang and other DSLs can be integrated into an MLIR compiler stack for rapid prototyping and language design. Moving to MLIR, the EVEREST HPC workflow became a composition of a separate DSL front-end and a reusable domain- and target-specific middle-end. In addition, we reported on extensions to the CFDlang language that we deemed necessary to support the computational motifs identified within RRTMGP [27]. These included richer index expressions, but also user-level annotations to specify non-functional requirements (e.g., quantization) for our pipeline to use.

As described in Deliverable D4.2, RRTMGP became our final target application for the EVEREST HPC workflow. RRTMGP is a state-of-the-art implementation of the Correlated K-Distribution (CKD) algorithm to solve the gas optics problem required to calculate atmospheric radiative transfer. Focusing on the short-wave bands only, this means that EVEREST ultimately targets the 4 kernels of interpolation, minor- and major absorber contributions, and Rayleigh scattering. To implement them, we derived the EVEREST Kernel Language (EKL) DSL from our earlier CFDlang prototype, with EKL now being our final version of the EVEREST DSL.

In terms of language and middle-end features, the above kernels demand support for in-place construction, broadcasting, index re-association, and subscripted subscripts. Figure 3 shows an example of the mathematical expression and the corresponding DSL snippet for the major absorbers kernel. The given expression approximates how the local gas concentration of two key species ("flavor") causes their broadened absorption lines to impact transmission. A key feature of the expression is the subscript-of-subscript into the tabulated distributions (dependent on thermodynamic state p and T) and linear interpolation in 3 dimensions (dT, dp and $d\eta$). This example highlights some features of the DSL that make it vastly more concise and easier to optimize than the old RRTMG Fortran implementation:

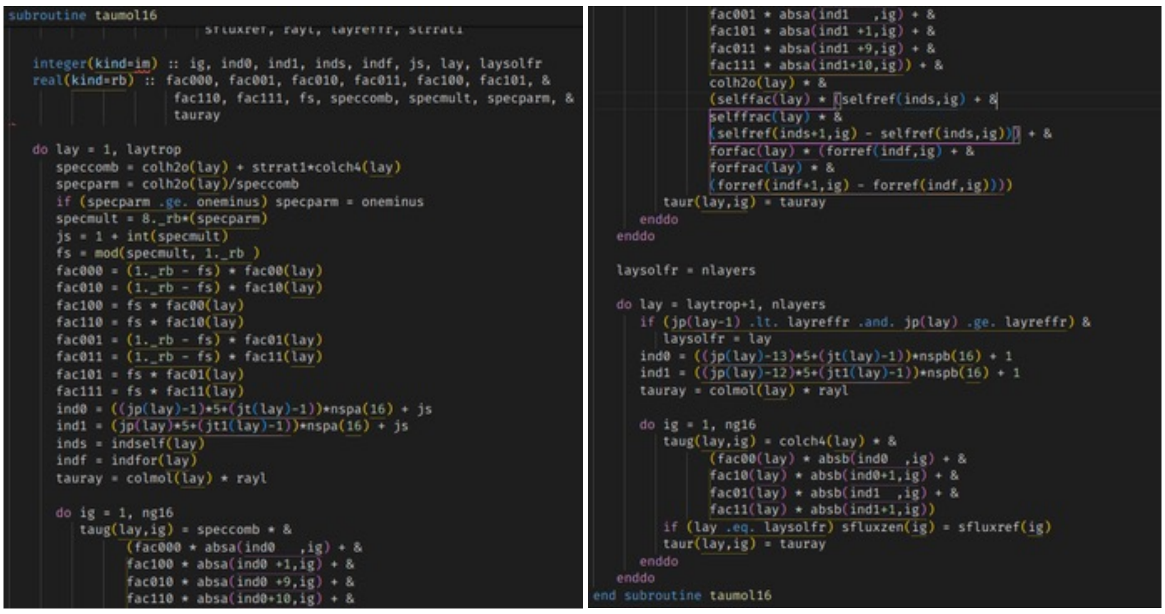
- Purity: Programs in EKL are pure, which means that expressions have no side-effects. This is closely related to their mathematical formulation, and leaves the compiler open to solutions that make clever use of memory. In the language, this is reflected through the `let` binding primitive, which supports scoping and shadowing (see line 3).
- Implicit loops: Instead of explicit loops, EKL programs are based on tensor primitives or index-metavariable expressions. This ensures that order dependencies are conscious decisions by the programmer and leaves the loop structure maximally open for implementation. Going further, Fortran's element-wise function opt-in is replicated, allowing those with scalar signatures to be implicitly applied to a tensor (note line 5).
- Einstein Summation Notation (ESN) The well-known ESN [21] allows tensor expressions that include reductions over known rings to be written naturally and concisely. This transforms tensor expressions into scalar expressions, which describe how individual result elements are produced. Using ESN, reductions prone to inadvertently inhibit compiler optimization are natural and safe (see line 9 ff.)

In the legacy RRTMG implementation, the kernel equivalent to line 9 is distributed over around 300 lines of Fortran code, spread across the 14 band functions. Some of this code is shown in Figure 4 for a direct

$$\tau_g^M = \sum_{dT} \sum_{dp} \sum_{d\eta} n_{\sigma_x, x, d\eta} \alpha_{\sigma_x, x, dT, dp, d\eta} k_{\overline{T}+dT, \overline{p}+dp, \overline{\eta}+d\eta, g}$$

```
// Interpolation (snippet)
let p_prime  = (log(p) - C_LOG_MAX_P_REF) / C_DELTA_LOG_P_REF
let p_prime  = clamp(0, p_prime, 58)
let i_strato = select(p < C_P_TROPO, 1, 0)
let j_p, f_p = intfrac(p_prime)
let i_p      = [j_p, j_p + 1] + i_strato

// Major contributors
let tau_g_M  = (n_mix[i_flav[x], x, deta]
                * f_major[i_flav[x], x, dT, dp, deta]
                * k_major[i_T[x, dT], i_p[x, dp], i_eta[x, deta], g])
```

Figure 3 – Example code in EKL.



```fortran
subroutine taumol16
                        sfluxref, rayl, layreffr, strrat1

   integer(kind=im) :: ig, ind0, ind1, inds, indf, js, lay, laysolfr
   real(kind=rb) :: fac000, fac001, fac010, fac011, fac100, fac101, &
                    fac110, fac111, fs, speccomb, specmult, specparm, &
                    tauray

   do lay = 1, laytrop
      speccomb = colh2o(lay) + strrat1*colch4(lay)
      specparm = colh2o(lay)/speccomb
      if (specparm .ge. oneminus) specparm = oneminus
      specmult = 8._rb*(specparm)
      js = 1 + int(specmult)
      fs = mod(specmult, 1._rb )
      fac000 = (1._rb - fs) * fac00(lay)
      fac010 = (1._rb - fs) * fac10(lay)
      fac100 = fs * fac00(lay)
      fac110 = fs * fac10(lay)
      fac001 = (1._rb - fs) * fac01(lay)
      fac011 = (1._rb - fs) * fac11(lay)
      fac101 = fs * fac01(lay)
      fac111 = fs * fac11(lay)
      ind0 = ((jp(lay)-1)*5+(jt(lay)-1))*nspa(16) + js
      ind1 = (jp(lay)*5+(jt1(lay)-1))*nspa(16) + js
      inds = indself(lay)
      indf = indfor(lay)
      tauray = colmol(lay) * rayl

      do ig = 1, ng16
         taug(lay,ig) = speccomb * &
               (fac000 * absa(ind0   ,ig) + &
               fac100 * absa(ind0 +1,ig) + &
               fac010 * absa(ind0 +9,ig) + &
               fac110 * absa(ind0+10,ig) + &
               fac001 * absa(ind1   ,ig) + &
               fac101 * absa(ind1 +1,ig) + &
               fac011 * absa(ind1 +9,ig) + &
               fac111 * absa(ind1+10,ig)) + &
               colh2o(lay) * &
               (selffac(lay) * (selfref(inds,ig) + &
               selffrac(lay) * &
               (selfref(inds+1,ig) - selfref(inds,ig))) + &
               forfac(lay) * (forref(indf,ig) + &
               forfrac(lay) * &
               (forref(indf+1,ig) - forref(indf,ig))))
         taur(lay,ig) = tauray
      enddo
   enddo

   laysolfr = nlayers

   do lay = laytrop+1, nlayers
      if (jp(lay-1) .lt. layreffr .and. jp(lay) .ge. layreffr) &
         laysolfr = lay
      ind0 = ((jp(lay)-13)*5+(jt(lay)-1))*nspb(16) + 1
      ind1 = ((jp(lay)-12)*5+(jt1(lay)-1))*nspb(16) + 1
      tauray = colmol(lay) * rayl

      do ig = 1, ng16
         taug(lay,ig) = colch4(lay) * &
               (fac00(lay) * absb(ind0  ,ig) + &
               fac10(lay) * absb(ind0+1,ig) + &
               fac01(lay) * absb(ind1  ,ig) + &
               fac11(lay) * absb(ind1+1,ig))
         if (lay .eq. laysolfr) sfluxzen(ig) = sfluxref(ig)
         taur(lay,ig) = tauray
      enddo
   enddo
end subroutine taumol16
```

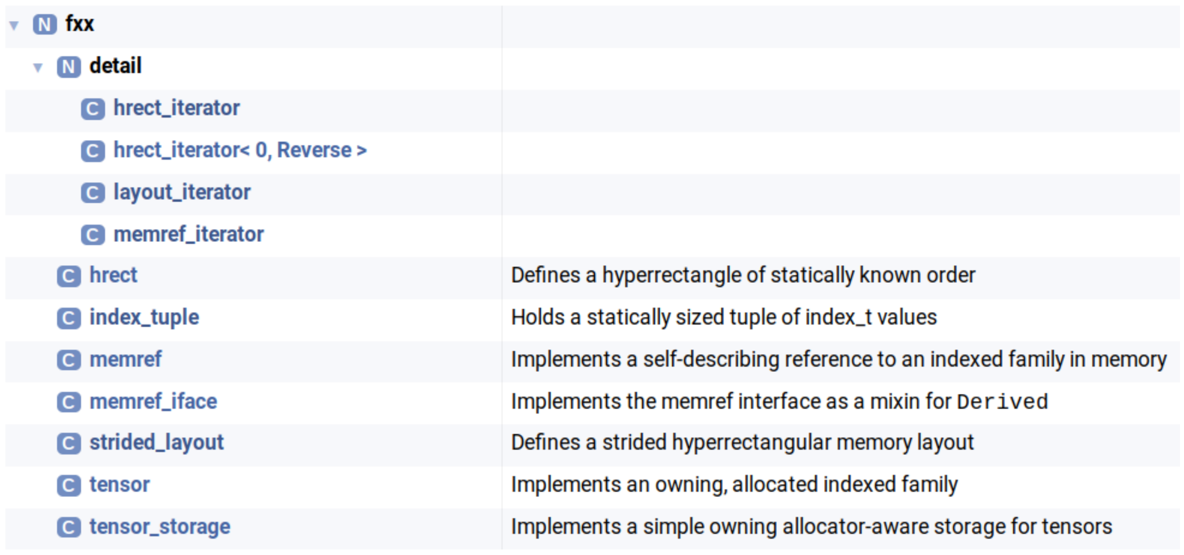
Figure 4 – RRTMG code snippet in Fortran.

comparison with the conciseness of the DSL. A meaningful comparison is almost impossible due to this well-known shortcoming of the legacy code. Although the more modern RRTMGP implementation eliminates much of this code duplication as well, it is still an order of magnitude longer than the EKL implementation.

3.2 Fortran Integration for WRF

In Deliverable D4.2, we described how a DSL such as EKL can be embedded into an existing legacy application by mimicking the host language. In particular, we were referring to how CFDlang can integrate into Fortran by syntactically allowing the array definitions to be copied, thus eliminating indexing and Application Binary Interface (ABI) issues. Our goal was to establish a Fortran Foreign Function Interface (FFI) layer that would allow embedding the DSL directly into Fortran, and compiling using a preprocessing step.

Embedding in Fortran turned out to be infeasible and less practical than we expected. This is mainly due to unsolvable ABI issues with Fortran and the structure and data layout of the legacy RRTMG implementation. To obtain a workable and easily maintained implementation, an ABI has to be frozen manually, unless massive scope creep at the level of WRF modularization is acceptable. In fact, new radiation schemes in WRF are known to be notoriously hard to ship and existing ones are tightly coupled within WRF (requiring WRF modifications). Moreover, the legacy RRTMG scheme is outdated and was not designed for offloading. Hence, a tight integration within that module would quickly become obsolete. As mentioned before we thus focus on the CKD approach which is state-of-the-art, and is the one used by RRTMGP's gas optics and other modern schemes such as the ECMWFs ecRad. Our new design came also after consulting experts in the field, including the authors of RRTMG and RRTMGP.



▼ N fxx	
▼ N detail	
C hrect_iterator	
C hrect_iterator< 0, Reverse >	
C layout_iterator	
C memref_iterator	
C hrect	Defines a hyperrectangle of statically known order
C index_tuple	Holds a statically sized tuple of index_t values
C memref	Implements a self-describing reference to an indexed family in memory
C memref_iface	Implements the memref interface as a mixin for `Derived`
C strided_layout	Defines a strided hyperrectangular memory layout
C tensor	Implements an owning, allocated indexed family
C tensor_storage	Implements a simple owning allocator-aware storage for tensors

Figure 5 – FXX template library overview

Nevertheless, solid FFI is still required between MLIR and C++, the latter of which being the language we wrote the WRF plug-in in. While solutions exist, we provide a more modern C++20 wrapper that facilitates using `memref` primitives as introduced by MLIR (see Figure 5). This effectively trivializes the inclusion of kernels code-generated via MLIR, pinned to some C ABI for subsequent linking into the application, which is the main practical use of any currently feasible companion DSL compiler.

3.3 Dataflow: Ohua/Condrust

As basis for a language that can express dataflow, we use the Ohua compiler framework [10, 9]. An earlier prototype of the framework used its own frontend DSL that closely resembled the host language, but differed in the supported language features and the grammar definition [46]. The current version of the language, instead, uses a subset of the host language as input (e.g., Rust and Python). This allows the easy reuse of existing code, which significantly improves the usability of Ohua, one of the key requirements defined in Deliverable D2.2. More importantly, it also allows for quicker compiler extensions to support new languages. We can use existing parser and Abstract Syntax Tree (AST) definitions in both the front-end for parsing and the back-end for code generation. The current description of the syntax and the semantics of the language can be found in [41], with a more modern version of the language called **Condrust**. The compilation flow described in Deliverable D4.2 is now updated as shown in Figure 6.

An Ohua/Condrust program is a composition of calls to both stateless and stateful functions in a separate file, making it a formal coordination language. As described in Deliverable D4.2, the programming model imposes restrictions, some beneficial to safety and ease hardware-software designs. Most notably, passing by reference is prohibited, and programs follow a strict move semantics. This means that all dataflow dependencies become visible in the program and thread-safety can be ensured.

A key motivation behind using a dataflow-driven DSL in the EVEREST project is that a Dataflow Graph (DFG) abstracts over the individual computations that form the algorithm. This comes in handy when deploying such a program onto heterogeneous architectures using the `dfg` MLIR dialect. The abstraction allows for a tight integration for off-loading single nodes of the DFG to FPGAs using HLS, as outlined in Section 5. Since the off-loaded functions themselves are not part of the compilation process, the main difference between offloaded functions and normal functions is the communication with the nodes. Normal nodes of the DFG communicate with one another using FIFO queues, both in the parallel Rust runtime as in the parallel MLIR generated

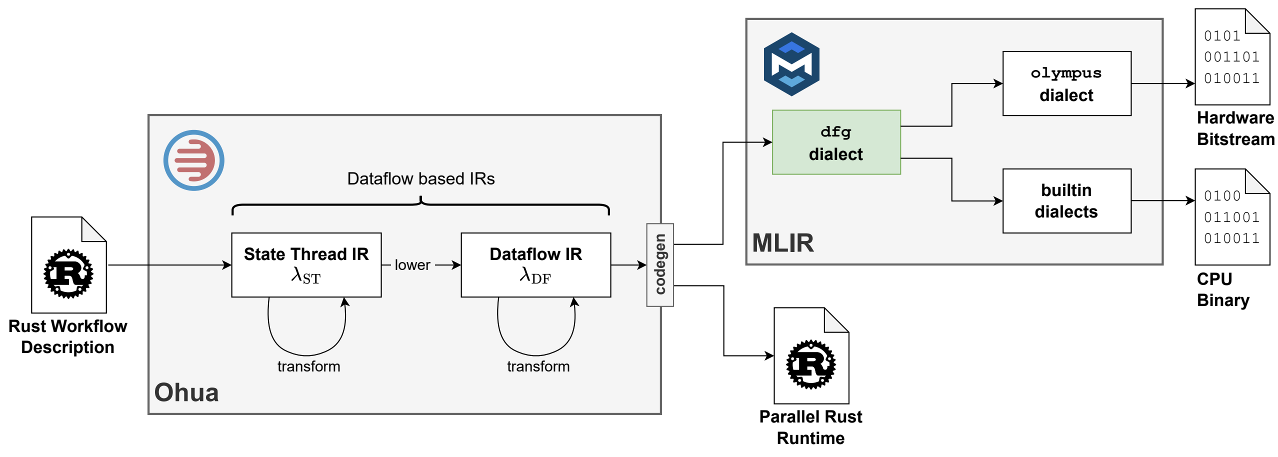


Figure 6 – Current status of the Ohua/Condrust and dfg compilation flow.

runtime. Communication with an off-loaded node, however, requires actual data transfers from and to the accelerator. Furthermore, not all data-flow transformations provided by Ohua/Condrust can be trivially applied to code running on hardware, such as parallelization. To notify the Ohua Compiler (`ohuac`) of this change, functions that will be deployed onto the FPGA are annotated with a macro:

```
#[kernel(offloaded = true, multiplicity = [1, 1, 1], path = "projection.cpp")]
let cv: CandiVector = projection(gv, mapcell);
```

The `offloaded` flag allows the simple toggling of the offloading of a node in the graph. All additional information in the annotation is required by down-stream tools of the EVEREST SDK, namely *Olympus*, as will be explained in Section 4.3 and Section 4.5.

An implementation of the *Map Matching* algorithm from the Traffic Simulation Use Case, would then look like this:

```
fn match_one(gv: GpsVector, mapcell: MapCell) -> RoadSpeedVector {
    #[kernel(offloaded = true, multiplicity = [1, 1, 1], path = "projection.cpp")]
    let cv: CandiVector = projection(gv, mapcell);

    let t: Trellis = build_trellis(gv, cv, mapcell);
    let rsvbb: RoadSpeedVector = viterbi(t, cv);
    interpolate(rsvbb, mapcell)
}
```

3.4 Machine Learning

With respect to language support for Machine Learning (ML) no fundamental changes were applied to the compilation flow described in Deliverable D4.2. This is mainly due to fact that ML frameworks and the community-standard Open Neural Network eXchange (ONNX) [43] specification used in EVEREST are mature and have remained stable. Recently, the ML flow also supports `torchscript` [6] as input language, besides ONNX. The support of `torchscript` improves the connection to the community framework Pytorch, which is used to develop and train DNNs. Also, the support of `torchscript` allowed the integration of post-training quantization (PTQ) via the open-source tool Brevitas [22].

4 Intermediate Representations and Transformations

Figure 4 in Deliverable D4.2 sketched the status of the intermediate representations in the project in the alpha release (replicated in Figure 7 for better readability). The figure captured the concerted effort to achieve unification at the level of the intermediate representations. Apart from a frontend for Ohua/Condrust and integration of ML applications, an important missing component was an end-to-end integration with Olympus for system-level design.

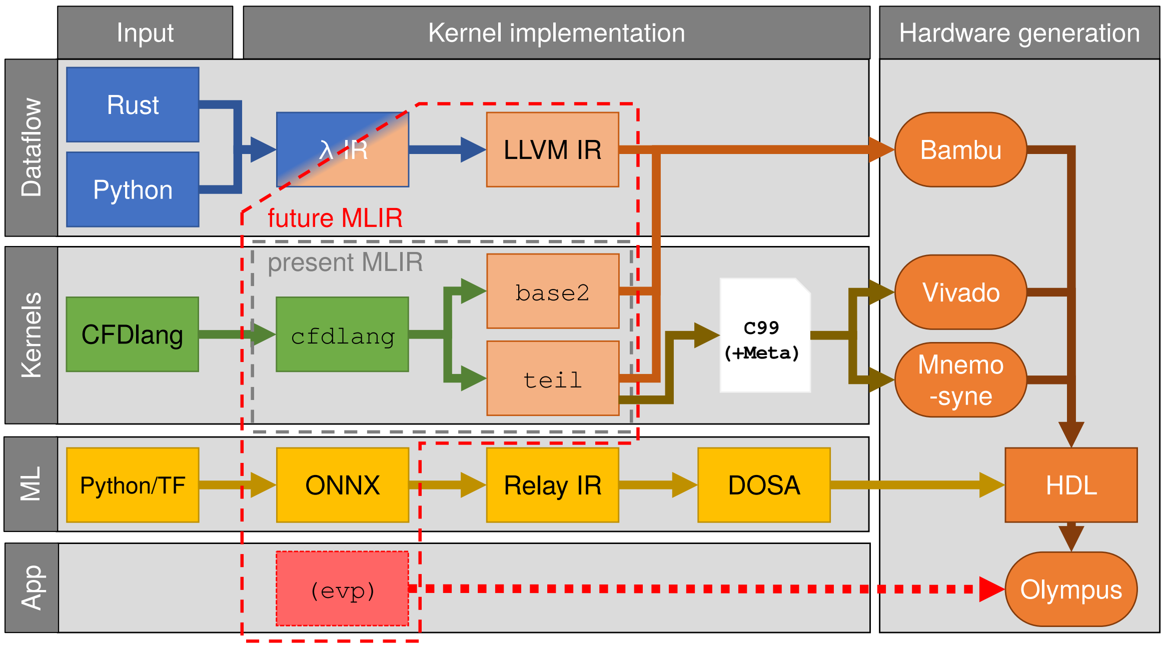


Figure 7 – IR status and plans in Deliverable D4.2.

Today, the MLIR stack of the EVEREST SDK is much more complete as depicted in Figure 8.This design follows the design rationale described in Deliverable D4.2. As shown in the figure, external frontends can generate representations to enter the EVEREST SDK, like `torch` and `tosa`. ML applications from TVM can be read into the `jabbah` dialect. To converge and optimize the different DSLs for ML, we follow the concept of Operation Set Architectures [30]. This level of abstraction, captured by `jabbah`, is also used to optimize the distribution of ML applications [34]. The SDK provides dialects for the frontends of the kernel language (`ekl`), the coordination language (`dfg`), and the legacy CFDlang (`cfdlang`). The dialects `ekl` and `cfdlang` can be lowered to an MLIR implementation of the intermediate tensor language [35] (`teil`) and a new dialect for the Einstein notation (`esn`) described in Section 3.1. These abstractions are used to implement a series of transformations as will be described in this section.

Custom data representations are often needed to truly exploit the efficiency of hardware implementations. To this end, the SDK includes a set of dialects to properly model custom data types in MLIR[13], namely, `base2`, `cyclic`, `bit` and `ub`. The latter is being moved to core MLIR for proper support for undefined behavior. The remaining dialects handle integration within the EVEREST platform (`evp`) and system-level optimization based on the dataflow of the application (`olympus`). Both are highly relevant for the hardware generation flow described in Section 5.

In previous deliverables, we have provided details about dialects such as `base2`, `teil`, and `cfdlang`. In this section, we focus on the new dialects `ekl`, `esn`, `dfg`, `evp`, and `olympus`. While most target- and domain-specific optimizations have been described in Deliverable D4.2 and our corresponding publications, this last phase increased the technical complexity of our MLIR dialect stack. As a result, additional utility transforms were introduced, which we mention in the following whenever they become relevant.

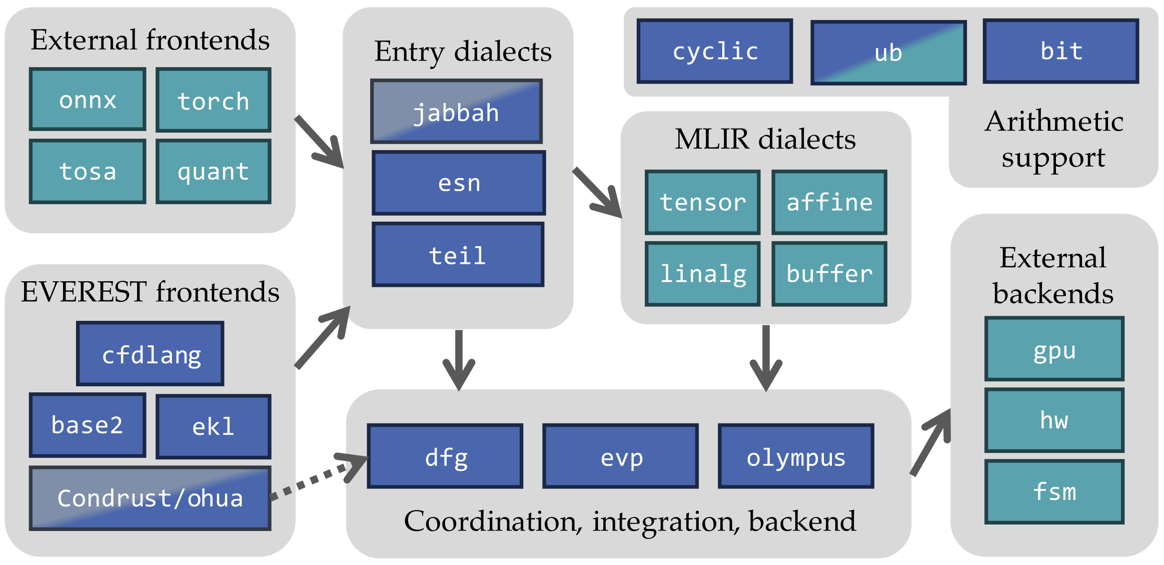


Figure 8 – EVEREST MLIR dialects (in blue) and their integration with other core MLIR dialects (green). Dialects under construction in grey-blue.

4.1 ekl

In Section 3.1, we described how the EKL is designed and what features its front-end possesses. This is realized using the `ekl` dialect, which is an MLIR implementation of the EKL front-end analogous to what was previously reported in Deliverable D4.2. In particular, `ekl` is an AST for EKL, as well as the IR that holds the result of expression elaboration during semantic analysis.

The `ekl` dialect is directly based on `cfdlang`, which means that it models the syntactical structure of an EKL program. Conceptually, an `ekl` program is compiled into an accelerator by simplifying expressions, deciding on tensor storage and lifetime, and then implementing the resulting side-effecting expressions. In Deliverable D4.2, we briefly described the canonicalization and expression implementation steps required. In addition to `cfdlang`'s features, the `ekl` dialect now also has first-class support for type-level annotations.

MLIR does not feature a dependently-typed type system. Cutting-edge MLIR-backed language implementations, such as Mojo and, to some extent, CIRCT, come with their own mock-up of dependent types. EKL also requires such a feature to implement the user-facing type annotations used to perform automatic Domain-Space Exploration (DSE) of scalar data types. As shown in Figure 9, we decided to encode such constructs by representing types as values. Consequently, the MLIR types of `ekl` operations are in fact only of syntactic nature, e.g., "any tensor".

$$\mathbb{Q}^+ := [0, \infty) \in \mathbb{Q} \qquad\qquad \mathtt{si22\_9} := \mathtt{si16\_8} \sqcup \mathtt{ui21\_9}$$

```
%rat     = ekl.type : !ekl.rat                   %si16_8  = ekl.type : !base2.si16_8
%rat_pos = ekl.type.narrow [0, inf) in %rat      %ui21_9  = ekl.type : !base2.ui21_9
                                                 %si22_9  = ekl.type.super %si16_8, %ui21_9
```

Figure 9 – Example of EKL type constraints.

As a result, type checking on an `ekl` program has to be performed in a separate pass, which instantiates the SSA values representing the concrete types. The well-formedness of such a fully annotated program is verified using standard MLIR verifiers, which can then act fully locally. Another aspect related to type checking is the correctness of subscripting and consistent use of index-metavariables. This is mostly left to the `esn` dialect, which `ekl` expressions translate to.

4.2 esn

In Section 3.1, we described a new feature of EKL, which is its ability to represent expressions in ESN. An ESN expression is a type of tensor expression that is based on index-metavariables and simplifies reductions over known rings.

Let $\mathbf{A}$, $\mathbf{B}$ denote values of $\mathbb{R}^2$. The matrix-matrix product $\mathbf{C} = \mathbf{AB}$ is given by the the following ESN expression

$$\mathbf{A}_{ik}\mathbf{B}_{kj} \tag{1}$$

By convention, an index appearing only once is *bound*, and an index appearing exactly twice is a *reduction*. In our example, the indices i and j are bound, and k is a reduction. The shape of the result of an ESN expression is implied by the domains of the bound indices in the order they appear in. We can make this explicit by turning the expression into an assignment

$$\mathbf{C}_{ij} = \mathbf{A}_{ik}\mathbf{B}_{kj}$$

The indices i, j, k vary over the result elements, and are therefore *local* to the expression. An arbitrary but fixed parameter is called a *free* index, and must be distinguished from the local indices. The reduction in k can be made explicit to resolve this issue

$$\begin{aligned}\mathbf{C}_{ij} &= \mathbf{A}_{i\boxed{k}}\mathbf{B}_{\boxed{k}j}\\ &= \sum_k \mathbf{A}_{ik}\mathbf{B}_{kj}\end{aligned}$$

By convention, the domain of a local index is implied by *all* of its subscript uses. This means that every domain implied by an operand subscript must be congruent for a given local index. Also, subscript index arithmetic is not allowed. An ESN expression involving a reduction implies some commutative semigroup. It follows that the order of reductions is irrelevant and that scalar-associative ESN expressions are also tensor-associative. In general, ESNs are used when there is an unambiguous semiring defined over the involved scalars.

We implemented an MLIR dialect called `esn` that represents ESN expressions. An example program is provided in Figure 10. Its primary purpose is to make tensor expressions accessible to scalar arithmetic transformations, a strong point of ESN. Since ESN expressions imply parallelism and regular loops, `esn` also serves to estimate and reduce both complexity and memory. Finally, `esn` provides a more convenient intermediary for a front-end lowering path due to its implied shapes.

The `esn` dialect features an elaboration step that "type checks" the index-metavariables involved in the expression. This makes it able to work with dynamically-sized tensors, while still eagerly checking static correctness. This feature is used by the `ekl` front-end, to elaborate the user expressions, which do not involve the dimensions. Although this system could be extended for non-congruential index geometries, this is not part of the ESN, and we were unable to find any uses for, e.g., affine indexing.

4.3 dfg

The `dfg` dialect has been designed to express KPN-style dataflow graphs with heterogeneous node placement. It can be generated from a more high-level graph representation, as outlined in Section 3.3. This can then leverage some of the dataflow transformations described in Deliverable D4.2. However, `dfg` also features a custom syntax which allows users to even ergonomically express a graph directly using the dialect.

The dialect consists of the types and primitives necessary to model the two main components of a data flow graph: *nodes* (i.e., encapsulated library functions provided by the developer), and *edges* (i.e., FIFO communication channels between the nodes). It provides a `dfg.operator` op to define the semantics of the nodes and `dfg.instantiate` to instantiate the operator as a node in the graph. For streaming semantics, `dfg.loop` provides a way to loop operator execution until connections are closed. The `dfg.channel` operator creates two connected `dfg.input` and `dfg.output` values. These can be used within an operator to move data to and from the channels using `dfg.push` and `dfg.pull`. Offloaded kernels can be marked as such when instantiating an `dfg.operator`.

A complete example of an algorithm expressed in the `dfg` dialect can be seen in 1. It shows a graph constructed from three nodes, `sum`, `get_op` and `pow2`. The latter is offloaded onto hardware, the sources of which reside in the specified file. The `sum` node runs in a loop, streaming new inputs.

```
dfg.operator @sum inputs (%op_a: ui32, %op_b: ui32)
                  outputs (%a: ui32) {
  dfg.loop inputs (%op_a: ui32, %op_b: ui32) outputs (%a: ui32) {
    %inp1 = dfg.pull %op_a : ui32
    %inp2 = dfg.pull %op_b : ui32

    %result = arith.addi %inp1, %inp2 : ui32

    dfg.push(%result) %a : ui32
  }
}

dfg.operator @get_op outputs (%op_b: ui32) {
  %b = func.call @get_op() : ui32
  dfg.push(%b) %op_b : ui32
}

dfg.operator @pow2 inputs(%op_c_in: ui32) outputs(%op_d_out: ui32)
        attributes { evp.path = "src.cpp", multiplicity = array<i64: 1, 1>}

// algorithm entry point
func.func @run_dfg(%op_a: ui32) -> ui32 {
  %op_a_in, %op_a_out = dfg.channel() : ui32
  %op_b_in, %op_b_out = dfg.channel() : ui32
  %op_c_in, %op_c_out = dfg.channel() : ui32
  %res_in, %res_out = dfg.channel() : ui32
  // inputs
  dfg.push(%op_a) %op_a_out

  dfg.instantiate @get_op outputs(%op_b_out)
  dfg.instantiate @sum inputs(%op_a_in, %op_b_in) outputs(%op_c_out)
  // kernels seamlessly integrate
  dfg.instantiate offloaded @pow2 inputs(%op_c_in) outputs(%res_out)

  %res = dfg.pull %res_in : ui32

  return %res : ui32
}
```

Listing 1 – A complete example of an algorithm implemented in the `dfg` dialect.

```
%C = esn.let [%i, %j] : f64 {
    %k = reduction
    %A_ik = subscript %A[%i, %k] -> f64
    %B_kj = subscript %B[%k, %j] -> f64
    %C_ij = arith.mulf %A_ik, %B_kj : f64
    yield %C_ij : f64
} reduce (%lhs, %rhs) {
    %sum = arith.addf %lhs, %rhs : f64
    yield %sum : f64
}
```

(a) Domain-implicit form

```
%c1 = index.constant 1
%dimA_1 = tensor.dim %B, %c1 : tensor<8x?xf64>
%dimB_1 = tensor.dim %B, %c1 : tensor<3x?xf64>
%C = esn.let [%i, %j] : f64 {
    %k = reduction
    upper_bound(%i) = 8
    %pi = to_index %i
    upper_bound(%k) = %dimA_1
    %pk = to_index %k
    %A_ik = subscript %A[(%pi), (%pk)] :
    ↪   tensor<8x?xf64>
    upper_bound(%k) = 3
    upper_bound(%j) = %dimB_1
    %pj = to_index %j
    %B_kj = subscript %B[(%pk), (%pj)] :
    ↪   tensor<3x?xf64>
    %C_ij = arith.mulf %A_ik, %B_kj : f64
    yield %C_ij : f64
} reduce (%lhs, %rhs) {
    %sum = arith.addf %lhs, %rhs : f64
    yield %sum : f64
}
```

(b) Domain-explicit form

Figure 10 – esn program for eq. (1)

A data flow graph runtime described by this dialect spawns all instantiated nodes of the graphs with their connections in place. All nodes execute in parallel, the blocking channel semantics ensuring an efficient use of computing resources during execution. When an operator terminates, all used input and output channels will be closed. Operators that are internally using a `dfg.loop` will be executed in an endless loop until one input or output channel is broken, i.e., the other end of the channel is missing. If that happens, the operator terminates immediately, not processing any further input and instead triggering all linked nodes to terminate execution, too. This ensures that the runtime cannot get stuck in an endless loop without further data.

Non-looped operators are used, for instance, to capture environment variables that are injected into the graph. This usually includes the initial inputs to the computation, ensuring the computation pipeline shuts down after processing all inputs.

For offloaded nodes, a wrapper will be generated for the host-side driver of *Olympus*, which will be explained in Section 4.5.

Other dialects that provide abstractions for this Model of Compute (MoC) exist, most notably the `handshake` dialect [19] from the CIRCT project. However, this abstraction is not usable for the EVEREST project, as it has been designed for use in hardware generation and, therefore, uses the handshake protocol for data exchange, which is not necessary here. Furthermore, it uses implicit FIFO channels, giving users no explicit control over their use and sizing, which is an integral part of the abstraction provided through `dfg`.

4.4 evp

The `evp` dialect is a dialect that, looking forward, will represent high-level connections between software and hardware components for heterogeneous deployments. Its operations assist in relocating nodes of dataflow graphs across heterogeneous compute devices. It is used by the `evp` tool, which can be seen as the equivalent of what a linker is in traditional software deployments. More information will be provided in Section 6.2 in the context of the final code generation flow of the SDK.

4.5 olympus

Kernel operator

```
"olympus.kernel"(%2, %3, %4) {
    callee = "matmul",
    latency = 795, ii = 268,
    ff = 3106, lut = 6174, bram = 61,  uram = 0, dsp = 48,
    operand_segment_sizes = array<i32: 2, 1>,
    path = "dir/matmul.cpp"
} : (
    !olympus.channel<i32>,  !olympus.channel<i32>,
    !olympus.channel<i32>
) -> ()
```

Return: void (Any outputs are the last parameters)

Attributes:

`callee` : The name of the kernel implementation (C function, Verilog module, etc)

`latency, ii` : Timing estimates (latency, initiation interval) from kernel HLS/synthesis

`ff, lut, bram, uram, dsp` : Resource estimates from kernel HLS/synthesis

`operand_segment_size` : Defines which parameters are inputs and outputs. (In this example, the ‘2’ in index 0 means the first two parameters are inputs. The ‘1’ in index 1 means the next 1 parameter is the output.)

`path` : Where to find the kernel implementation file.

Parameters: The inputs and outputs as determined by `operand_segment_size`. Either scalar data of primitive types or olympus.channel types. In the same order as in the kernel implementation.

Channel operator

```
%2 = "olympus.channel"() {
    paramType = "stream",
    depth = 20
} : () -> (
    !olympus.channel<i32>
)
```

Return: The channel, to be used as input/output operands in kernel operators.

Attributes:

`persistent` : *(optional, default:* `false`*)* Boolean. If true, this data is transferred between the host and HBM/DDR once during initialization. An `olympus.index` can be associated in order for the kernel to use different portions per iteration.

`paramType` : describes the properties of the data in one of three ways:

- “`stream`”: Must be produced and consumed in the same order and consist of small, statically sized elements.
- “`small`”: Can be random access, but in total the data needed for a single kernel iteration should be at most on the scale of 100s of kB and be organized of simple structures without nesting or indirection.
- “`complex`”: Can be anything: huge, random access, have indirection, and/or be constructed of nested structures.

`depth`: Describes how large the data is in total. If `paramType==stream`, `depth` is the maximum necessary channel depth. If `paramType==small`, `depth` is the number of elements. If `paramType==complex`, `depth` is the number of bytes.

Channel type

```
!olympus.channel<i32>
```

Type parameter: A signless integer of arbitrary bitwidth. The interpretation of the data is not important, only the width. Therefore, a 32-bit float, a fixed-point value with 10 integer bits and 22 fraction bits, and a 32-bit integer should all be represented as ‘`i32`’.

Index operator

```
%2 = "olympus.index"( %2 ) {
    depth = 4
} : (
    !olympus.channel<i32>
) -> ()
```

Return: void

Attributes:

`depth`: Describes how large the data needed for one iteration is. It should be a factor of the depth of the associated persistent channel.

Parameters: The associated channel. The value of the index along with the depth (used as a stride) will be used to select a portion of the data from this channel for transferring between HBM/DDR to the kernel.

Index type

```
!olympus.index<i32>
```

Type parameter: The type of the scalar value used as an index.

One advantage of this flow is that it directly enables us to target CPUs, and also puts graphical processing units (GPUs) in range of future extensions. As an example, Figure 11 shows the inverse Helmholtz operator kernel in the `cfdlang` dialect in MLIR, next to an excerpt of its LLVM-IR lowering. This lowering is obtained using standard MLIR components directly from the bottom of our dialect hierarchy and can be processed with LLVM optimizers for targeting many different CPU architectures.

```
module {
  cfdlang.program {
    cfdlang.input @S : [11 11]
    cfdlang.input @D : [11 11 11]
    cfdlang.input @u : [11 11 11]
    cfdlang.define @t : [11 11 11] {
      %0 = cfdlang.eval @u : [11 11 11]
      %1 = cfdlang.eval @S : [11 11]
      %2 = cfdlang.prod %1, %0 : [11 11], [11 11 11]
      %3 = cfdlang.cont %2 : [11 11 11 11 11] indices [2 5]
      %4 = cfdlang.prod %1, %3 : [11 11], [11 11 11]
      %5 = cfdlang.cont %4 : [11 11 11 11 11] indices [2 5]
      %6 = cfdlang.prod %1, %5 : [11 11], [11 11 11]
      %7 = cfdlang.cont %6 : [11 11 11 11 11] indices [2 5]
       cfdlang.yield %7 : [11 11 11]
    }
    cfdlang.define @r : [11 11 11] {
      %0 = cfdlang.eval @t : [11 11 11]
      %1 = cfdlang.eval @D : [11 11 11]
      %2 = cfdlang.mul %1, %0 : [11 11 11], [11 11 11]
       cfdlang.yield %2 : [11 11 11]
    }
    cfdlang.output @v : [11 11 11] {
      %0 = cfdlang.eval @r : [11 11 11]
      %1 = cfdlang.eval @S : [11 11]
      %2 = cfdlang.prod %1, %0 : [11 11], [11 11 11]
      %3 = cfdlang.cont %2 : [11 11 11 11 11] indices [1 5]
      %4 = cfdlang.prod %1, %3 : [11 11], [11 11 11]
      %5 = cfdlang.cont %4 : [11 11 11 11 11] indices [1 5]
      %6 = cfdlang.prod %1, %5 : [11 11], [11 11 11]
      %7 = cfdlang.cont %6 : [11 11 11 11 11] indices [1 5]
       cfdlang.yield %7 : [11 11 11]
    }
  }
}
```

```
; ModuleID = '<stdin>'
source_filename = "LLVMDialectModule"

@__constant_11x11x11xf64 = private constant [11 x [11 x [11 x double↩
       ]]] zeroinitializer

declare i8* @malloc(i64)

declare void @free(i8*)

define void @kernel(double* %0, double* %1, i64 %2, i64 %3, i64 %4, ↩
       i64 %5, i64 %6, double* %7, double* %8, i64 %9, i64 %10, i64 ↩
       %11, i64 %12, i64 %13, i64 %14, i64 %15, double* %16, double* ↩
       %17, i64 %18, i64 %19, i64 %20, i64 %21, i64 %22, i64 %23, i64 ↩
       %24, double* %25, double* %26, i64 %27, i64 %28, i64 %29, i64 ↩
       %30, i64 %31, i64 %32, i64 %33) {
  %35 = call dereferenceable_or_null(10648) i8* @malloc(i64 10648)
  %36 = bitcast i8* %35 to double*
  br label %37

37:                                               ; preds = %63, %34
  %38 = phi i64 [ %64, %63 ], [ 0, %34 ]
  %39 = icmp slt i64 %38, 11
  br i1 %39, label %40, label %65

40:                                               ; preds = %37
  br label %41

41:                                               ; preds = %61, %40
  %42 = phi i64 [ %62, %61 ], [ 0, %40 ]
  %43 = icmp slt i64 %42, 11
  br i1 %43, label %44, label %63

44:                                               ; preds = %41
  br label %45

45:                                               ; preds = %48, %44
  %46 = phi i64 [ %60, %48 ], [ 0, %44 ]
  %47 = icmp slt i64 %46, 11
  br i1 %47, label %48, label %61

48:                                               ; preds = %45
  %49 = mul i64 %38, 121
  %50 = mul i64 %42, 11
  %51 = add i64 %49, %50
  %52 = add i64 %51, %46
```

Figure 11 – Example lowering from `cfdlang` to LLVM-IR (excerpt).

4.6 Connecting to HLS

As discussed in Deliverable D4.2, we guide HLS at a low-level abstraction, such as polyhedral descriptions in the `affine` dialect. During our higher level MLIR transforms, we can establish resource estimates in terms of memory bandwidth. On the polyhedral level, we use this to reschedule sub-kernel regions for HLS, inserting annotations. This means that we output vendor-specific pragmas, such as Xilinx's `#pragma HLS <?>`, into an interchange format, such as C99. Flows with better MLIR integration, such as Bambu, remove the interchange format in favor of passing this information directly in MLIR. As discussed in Deliverable D4.2, we retain the possibility of generating different variants of a kernel or function. This is enabled by expression rewriting and different polyhedral schedules (to generate different pipelines with intermediate buffers). These variants can be used for runtime selection.

4.7 Machine Learning Abstractions and Optimization

The ML flow to compile Deep Neuronal Networks (DNN) to FPGAs is implemented in the tool `DOSA` [28] using Operation Set Architectures [31]. In the following, we report on considerable progress with respect to the status from Deliverable D4.2. In the past few years, a large number of DNN-to-FPGA compilation flows have been proposed and implemented (see Section 4.7.1), all of them with limited application scope and scalability. The goal of DOSA was not to create yet another specialized compiler, but to create an organic compiler, which is capable of harnessing the core capabilities of existing ML-to-FPGA compiler frameworks. Today, DOSA provides interfaces to the most mature ML-to-FPGA compilers, but can be easily extended to include more and/or future compilers as well (see Section 4.7.2).

4.7.1 Operation Set Architectures

The FPGA community has been researching the implementation of neural networks on FPGAs for nearly 30 years, resulting in a *Cambrian explosion* [12] of DNN-to-FPGA tools[1] that scale from Edge to Cloud and target a wide variety of applications. Despite this variety, the architectures generated by all these existing frameworks can be sorted into two categories: **Engine-type** and **streaming-type** architectures, as depicted in the lower half of Figure 12.

The engine-type (see 4a in Figure 12) consists of one or multiple custom-designed processing units (i.e., engines) that can execute domain-specific instructions and are often referred to as *NPU* or *xPU*. These processing engines frequently contain dedicated units for matrix multiplication, vector processing, and non-linear functions, since these are the mathematical foundations of today's DNNs. Consequently, a DNN is broken down by a compiler into instructions that those processing engines can handle. These instructions are issued by a control unit at run-time and scheduled based on memory dependencies and processing unit availability. Although this pattern is simple, the design space is huge: For example, the processing elements can contain various specialized units with different data sizes or types. Examples of this type of architecture are TVM's VTA [18], Xilinx's Vitis AI [47] and Microsoft's Brainwave [12].

The streaming-type architecture (see 4b in Figure 12) integrates all the application-specific operations in the FPGA logic, so that the data *just* streams through the fabric at run-time. This type of architecture can achieve a higher throughput with lower latencies at the cost of higher resource usage than the engine type. The design space of this template is also huge. Starting with data types of different precision per operation to a variety of unrolling of loop parallelisms and pipelining options. Example frameworks that generate this type of accelerators are hls4ml [8], Haddoc2 [1], and FINN [3].

Both architecture *templates*, streaming and engine, are well justified for different reasons. The streaming template is best used for DNNs that require high throughput and/or low latency. The engine-type accelerators are better for large DNNs in latency-relaxed environments or if there are insufficient FPGA resources to implement the streaming type.

Throughout the project, we advocated for deploying mixed architectures built upon a combination of specialized streaming- and engine-type accelerators.

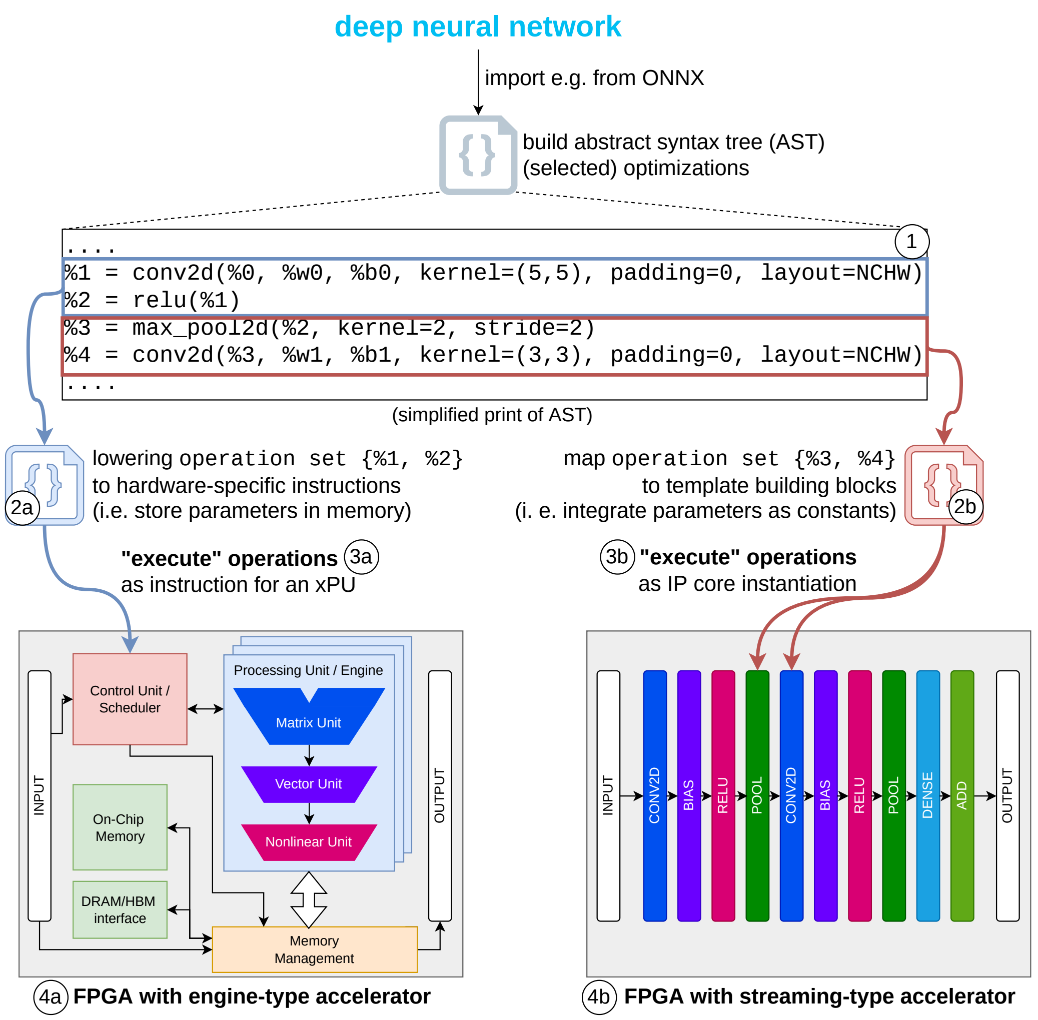

```
....
%1 = conv2d(%0, %w0, %b0, kernel=(5,5), padding=0, layout=NCHW)
%2 = relu(%1)
%3 = max_pool2d(%2, kernel=2, stride=2)
%4 = conv2d(%3, %w1, %b1, kernel=(3,3), padding=0, layout=NCHW)
....
```
deep neural network
import e.g. from ONNX
build abstract syntax tree (AST)
(selected) optimizations
1
(simplified print of AST)
2a
lowering operation set {%1, %2}
to hardware-specific instructions
(i.e. store parameters in memory)
map operation set {%3, %4}
to template building blocks
(i. e. integrate parameters as constants)
2b
"execute" operations 3a
as instruction for an xPU
3b "execute" operations
as IP core instantiation
INPUT
Control Unit / Scheduler
Processing Unit / Engine
Matrix Unit
Vector Unit
Nonlinear Unit
On-Chip Memory
DRAM/HBM interface
Memory Management
OUTPUT
INPUT
CONV2D
BIAS
RELU
POOL
CONV2D
BIAS
RELU
POOL
DENSE
ADD
OUTPUT
4a FPGA with engine-type accelerator
4b FPGA with streaming-type accelerator

Figure 12 – Basic principle of the Operation Set Architecture (OSA). An operation can be *executed* either by lowering it to an instruction (for engine-type accelerators, left-hand side) *or* by implementing it as parameterized hardware IP core (for streaming-type accelerators, right-hand side).

To compare different mixtures of architecture templates and to combine them into optimal solutions, we propose the operation set architecture (OSA) principle. The levels of abstraction of these operation sets are selected to allow the compiler to make meaningful comparisons of performance and resource metrics between totally different implementations.

With respect to the intermediate representation (IR) snippet of an abstract syntax tree (AST) in Figure 12 (1), the `conv2d`, `relu`, and `max_pool2d` constitute components of such operation sets. In essence, the operation sets are groups of those components. We selected this level of abstraction after we revisited the long list of specialized frameworks, existing domain-specific languages (DSLs), IRs, and optimization techniques of existing tool flows for DNNs. We observed that compilers of frameworks that target engine-type accelerators apply the heaviest optimizations. This results from the fact that DNNs are compiled into engine-type instructions, for which well-known optimization methods from classical CPU and GPU compilers can be used. Such optimizations include constant folding, dead code elimination, operator fusion, elimination of common sub-expressions, simplify paddings, and simplify the data flow graph for inference. Surprisingly, frameworks targeting streaming-type accelerators perform either no or only a few hardware-specific optimizations [1, 45, 49, 20] and even leave constant folding to the synthesis tools in most cases.

Therefore, we concluded that the level of abstraction for comparing different implementations should be chosen such that the decision between engine-type and streaming-type accelerators happens after these “basic” optimizations but before the program gets lowered further, since many of these optimizations would help the streaming-type accelerators, too. Following this path, the compiler can optimize a DNN graph above or at this level of abstraction (cf. (1) in Figure 12), detached from the lower-level details of the execution of one operation. This level is similar to the abstraction levels used by popular DSLs like RelayIR [18] [37] or

ONNX [44]. Only after that, the compiler decides between engine and streaming implementations of each operation. Therefore, such a higher-level operation is either lowered to engine/NPU/xPU-specific instructions or synthesized as a parameterized IP block for streaming accelerators. Engine operations are lowered into engine/NPU/xPU specific instructions (2a), and streaming operations are synthesized as a parameterized IP block (2b).

From an AST ① point of view, all operations all operations are *somehow* available specialist frameworks, illustrated with the arrows (3a) and (3b). How the operations will be *executed* in detail is up to the different frameworks (cf. (2a) and (2b)). Thus, each of these frameworks supports different a *set of operations* through its chosen target architectures (cf. (4a) and (4b)). However, the AST can be optimized and transformed without considering these details. In the example of Figure 12, the first two higher-level operations, `conv2d` and `relu`, are executed on an engine-type accelerator, while the following two operations, `max_pool2d` and `conv2d`, are then *executed* by implementing IP cores in a streaming way.

The method ① — ④ of Figure 12 is what we refer to as *"Operation Set Architecture"*, because it provides a meaningful unified abstraction level and utilities to compare and optimize different implementations of similar sets of operations.

4.7.2 DOSA: Distributed Operation Set Architectures using organic compiler principles

Our goal is to develop an *organic compiler* that analyzes a given DNN and selects the best-possible template-type and implementation offered by several frameworks for each arithmetic operation of this DNN. This compiler should understand a conventional DNN exchange standard and provide the user with insights on achievable performance, possible bottlenecks, and how the user's constraints influence the architectural decision. The DSE by this compiler should also consider partitioning, with model- and device-parallelism as options. They should ideally take only a few seconds to allow frequent iterations and optimizations with a user in the loop. Therefore, this compiler must be able to predict performance and resource consumption for each possible implementation quickly and reliably and must use meaningful criteria to compare these estimates.

Figure 13 shows the flow diagram of our organic compiler. Such an organic compiler requires a DSE phase that can analyze the DNN and that knows about the characteristics of the available frameworks. Consequently, an organic compiler has four inputs: The DNN Ⓐ, specified in a community standard as, e.g., ONNX [43], the targeted performance and resource constraints Ⓑ, the description of the targeted devices Ⓔ, and the available specialist frameworks as library Ⓓ.

The flow starts ① with the import of the DNN and the execution of straightforward optimizations, such as constant folding, dead code elimination or operator fusion. Also, an AST of the DNN is built. In parallel, the library of specialist DNN-to-FPGA frameworks Ⓓ and the library of available target platforms Ⓔ are imported ② and prepared for the DSE Ⓒ.

In the next step ③, the characterizations of step ② are then used to annotate the AST of the DNN operation-wise using a roofline-like analysis, together with the library of available platform characterizations.

Afterward, having a detailed AST annotation, the DSE phase starts with partitioning the DNN ④, if required by the size or throughput requirements of the DNN. The partitioning is based on the roofline analysis and bandwidth analysis. Next, based on an updated roofline analysis and the estimated latencies between nodes, high-level architectural decisions are made ⑤. Foremost, this decision involves deciding if weights of the operations of the DNN can be stored in an off-chip memory or if it has to stay on-chip because the available bandwidth would not allow to load them fast enough (cf. [31, 28]). Also, the best available specialist framework that can implement the decided micro-architecture with the derived performance requirements is selected. Next, ⑥, if multiple target devices are available, the best candidates are selected in this step. This step also includes calculating the resources necessary for the *glue logic* between the selected accelerator blocks. Here, it could be that this glue logic consumes more resources that are left on some devices. In that case, this result is annotated, and the compiler continues with another partition step. After a valid solution is found, or

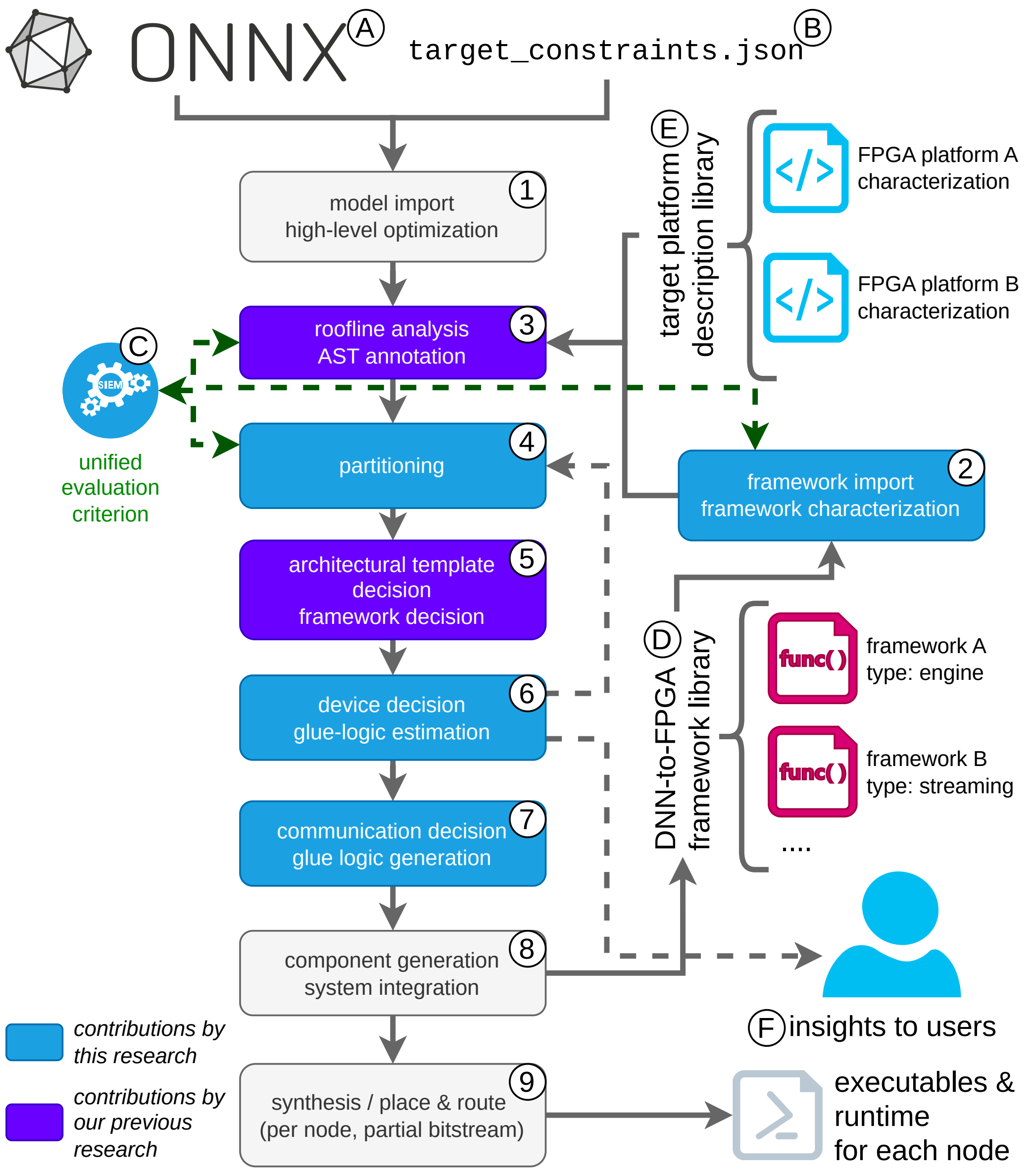

ONNX Ⓐ
target_constraints.json Ⓑ
model import high-level optimization ①
roofline analysis AST annotation ③
partitioning ④
architectural template decision framework decision ⑤
device decision glue-logic estimation ⑥
communication decision glue logic generation ⑦
component generation system integration ⑧
synthesis / place & route (per node, partial bitstream) ⑨
Ⓒ unified evaluation criterion
Ⓔ target platform description library
FPGA platform A characterization
FPGA platform B characterization
framework import framework characterization ②
Ⓓ DNN-to-FPGA framework library
framework A type: engine
framework B type: streaming
....
Ⓕ insights to users
executables & runtime for each node
contributions by this research
contributions by our previous research

Figure 13 – Flow of an organic compilation to (distributed) FPGAs.

if the compiler fails to find one, the user is informed about the resulting performance, resource footprints, and potential bottlenecks Ⓕ.

As the seventh step ⑦, the communication details between FPGA nodes are decided if the solution consists of multiple nodes. This involves finding the best synchronization pattern and deciding the type of serialization of multi-dimensional tensors. These decisions influence the latency between FPGA nodes only minimally since they are all implemented in a data-flow architecture as part of the network stack of the FPGA logic. For the inter-node communication, DOSA builds on the ZRLMPI framework [33, 32].

5 Hardware Generation Flow

The EVEREST compilation flow includes a flow to automate the generation of complex hardware architectures on FPGA. Our hardware generation flow aims at optimizing the computation of the kernel implementations produced by the fronted compilers and the data transfers with local and remote memories. Also, it supports multiple backends due to the different types of nodes envisioned in the EVEREST target platform.

The flow starts from the code produced by the frontend compiler, metadata for on-chip memory optimization, and a JSON file that includes platform details (e.g., type of the target FPGA, available resources, number and bandwidth of memory channels). It also requires the MLIR description of the kernel connectivity (Olympus dialect, cf. Section 4.5). The flow performs the following steps:

- it applies hardware-oriented optimizations and produces the hardware description of the kernel obtained from the compiler (Section 5.1 or Section 5.2 in the case of ML operators);
- it optimizes the on-chip memories by searching for sharing opportunities (Section 5.3);
- it creates the system-level description of the hardware architectures (Section 5.4). Based on the characteristics of the target FPGA node (either network- or bus-attached), the system generation part can perform communication optimizations or replicate the kernels to operate in parallel while coordinating the associated data transfers. In this step, it also creates the necessary files to interface with the proper synthesis tools and generate the bitstreams.
- it generates the specific implementations of the host code functions that reflect the transformations applied during the creation of the hardware architecture, along with interfaces with the runtime (Section 6).

In particular, this flow allows us to decouple the optimizations of the kernel and the system. The compiler produces the high-level description of the kernel, and the flow supports different HLS tools (e.g., Xilinx Vivado/Vitis HLS or Bambu) to create the corresponding hardware description. The system specification and the corresponding HLS depend on the synthesis flow used to target the specific FPGA node. For example, we use Vivado HLS 2019.2 for IBM cloudFPGA nodes and Vitis HLS 2021.1 for the Xilinx Alveo nodes (see Section 5.4.2 for more details).

5.1 Hardware-Oriented Optimizations and Kernel Generation

The EVEREST SDK uses a combination of HLS tools and hardware generators to create the hardware descriptions of the kernels identified by the compiler. As input, the kernel generation part supports C/C++ (synthesized with commercial HLS tools or Bambu), LLVM bitcode (supported by Bambu), and convolutional models (currently supported by DOSA through 3rd-party libraries). Each of these flows includes specific hardware-oriented optimizations to improve the hardware generation.

When the compiler flow emits C/C++, we currently use Bambu or Xilinx HLS tools to synthesize the corresponding hardware descriptions. In both cases, the `ap_fixed` library is used to specify custom data types so that they can be automatically synthesized. Table 2 summarizes the main differences in the features supported by Bambu and the Xilinx HLS tool, which can be used to assess which tool is the best choice in different situations. For example, applications from the traffic simulation use case in EVEREST that heavily rely on C++ with complex data structures and Standard Template Library containers should use Bambu, because making them compatible with Vitis HLS would require extensive manual rewriting of the code. The optimization features currently missing in Bambu (array partitioning, dataflow) can be implemented through MLIR as it has been done for loop pipelining, exploiting direct synthesis of MLIR through the Clang-LLVM frontend.

The PandA Bambu HLS tool is used within the EVEREST SDK to experiment with new features within the HLS flow. The HLS flow starts from a high-level description of the application and is able to generate an equivalent RTL design for a given target FPGA. The only constraints on the input description are about recursive functions and memory allocation. Only tail recursive functions are allowed for the HLS flow to complete successfully. Furthermore, dynamic memory allocation is supported but strongly discouraged, since it is quite

Feature	**Bambu**	**Vitis HLS**
Input languages	C/C++/LLVM IR	C/C++
Target hardware	Xilinx/Intel/Lattice/NanoXplore FPGAs, ASICs	Xilinx FPGAs
Codebase	Open-source	Proprietary
Loop optimizations	Unrolling, pipelining (through MLIR)	Unrolling, pipelining
Array partitioning	No	Yes
Dataflow support	No	Yes
Custom floating-point formats	Yes	No
C++ support	Complete (STL through ETL library)	No struct arguments, no pointer-to-pointer, no STL

Table 2 – Feature support in Bambu and the Xilinx HLS tool.

inefficient when implemented on an FPGA target. The input formats accepted by the HLS tool are both C/C++ descriptions and LLVM IR descriptions. This is possible since Bambu exploits as front-end of the HLS flow standard compilers such as GCC and Clang whose intermediate representation is then converted to the internal Bambu IR. This means any input description which is supported by the exposed front-end compilers can be fed to the HLS flow of PandA Bambu. In the EVEREST design flow, the MLIR description generated after the optimizations covered in previous sections is lowered into the LLVM IR dialect and mapped to an equivalent LLVM IR description passed to Bambu for HLS. Apart from the application description, the HLS flow also requires as inputs some metadata to guide the hardware generation process. A top level interface has to be defined to specify how parameters are exchanged between the accelerator and the host and specific memory interface types may be defined too, such as AXI interfaces. A target board and clock frequency must be set, so that the back-end of the HLS flow is able to generate a target specific RTL description and a proper scheduling of the operations to accommodate the required clock period. This information is extracted from the platform description file and passed to the tool. Finally, as a result of the HLS flow, an RTL description equivalent to the input application description is generated. The accelerator design will expose the required I/O interface and will implement a target optimized architecture to run the input application. The generated RTL description can be then passed to the subsequent system integration step as a custom black-box.

Besides PandA Bambu, the EVEREST SDK can also invoke 3rd-party tools or use domain-specific libraries to generate Hardware Description Language (HDL) code, especially for ML applications, if the ML compilation flow detects that the usage of such libraries would produce the better result. One example is the usage of the `Haddoc` library [1] for specific convolutions. In this case, DOSA generates the required Tool Command Langauge (TCL) scripts or meta-data represented in JavaScript Object Notation (JSON) to invoke those domain-specific 3rd-party tools.

In the following, we detail how the computation-related optimizations described in Deliverable D3.2 are integrated into the EVEREST compilation flow.

5.1.1 Loop Pipelining

The approach we proposed in [7] aims to leverage high-level code optimizations to provide a hardware-oriented input description to the HLS. Figure 14 shows the main steps and tools involved. The input MLIR code, which may contain loops to be pipelined, is first passed to a *scheduler* to obtain a loop iteration schedule. *Code transformations* are applied to the input code to reorganize the by improving the instructions parallelism. The resulting code is finally passed to the *HLS tool* to generate an accelerator description in Verilog/VHDL.

Loop pipelining requires a scheduling phase and a code generation phase. A single iteration of a loop may contain many operations which must be serialized because of data dependencies, thus they can not be run in parallel. Loop pipelining allows to schedule operations from different original iterations together: as these operations would not depend on each other, they could be executed in parallel without constraints. By overlapping original iterations, loop pipelining eliminates the parallelization constraints: all operations within the same iteration are independent now, since they belong to different iterations of the loop, so they can be executed in parallel.

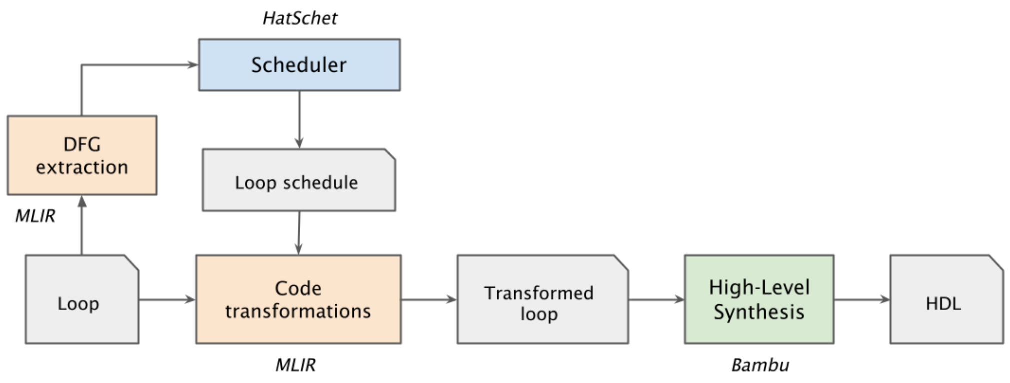


Figure 14 – Overview of the optimization flow for synthesis-oriented loop pipelining starting from MLIR description.

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

5.1.2 Custom Precision Floating-point Data Types

Custom floating-point data types are available within the EVEREST SDK and are implemented by the PandA Bambu HLS framework. They may be used by feeding specific flags to the HLS tool along with the input description of the application or they can be used directly within the application description language as a library through a C API.

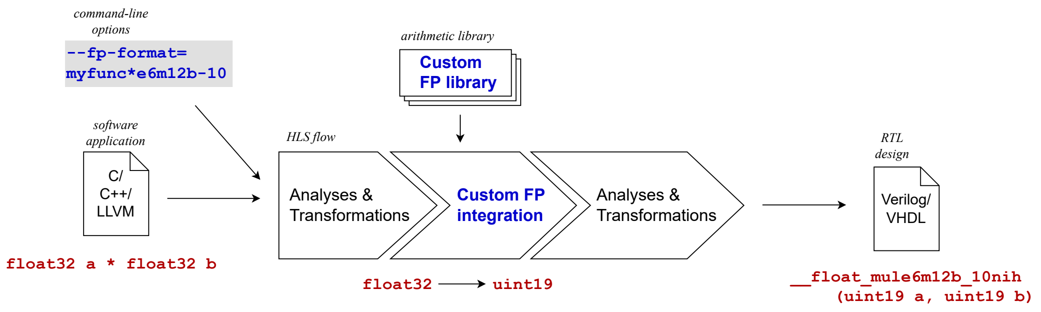


Figure 15 – Sample flow of custom floating-point application implementation through PandA Bambu HLS starting from generic input representation

The former case does not require any modification of the input description which is using standard floating-point data types. These types will be converted by the HLS tool following function-scope directives fed to the tool as command line options: each directive may require a custom floating-point format to be applied to a single function or function tree (a top function and all those called by it). Conversion from and to custom data types is handled internally by the synthesis flow in this case, providing fully automated translation of the input description, as shown in Figure 15. Conversely, the latter case leaves complete freedom to the upper levels of the EVEREST SDK to exploit the templatized floating-point functional units offered by the HLS component library. This is the case for the `base2` dialect introduced in previous deliverables. The input application is converted to `base2` dialect and custom data types are integrated at MLIR-level into the intermediate representation. Standard floating-point types are converted to custom types before the IR is fed into the HLS tool and operations involving floating-point custom types can be converted into function calls to corresponding templatized

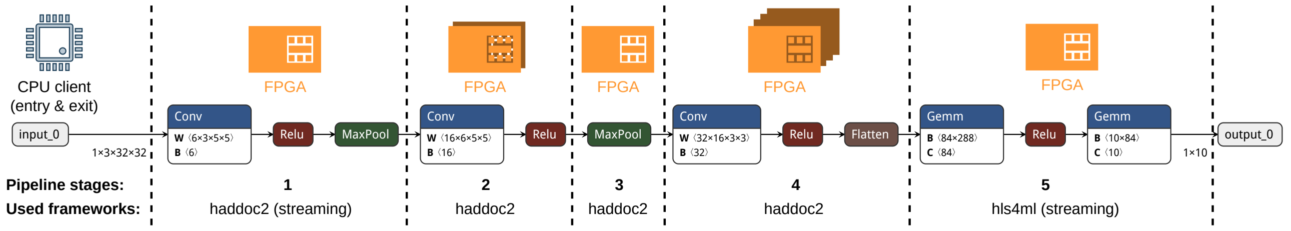


Figure 16 – One example of a DNN spread (PTTDNN) across 9 FPGAs. The CPU client just performs send and receive calls [28].

functions from the Bambu HLS library. Instead, when quantization from floating-point to fixed-point is applied, floating-point operations and types are replaced with integer ones during the lowering process from MLIR to LLVM IR. Anyhow, both cases will benefit from the rich set of inter-procedural transformations and optimizations offered by the HLS flow. Custom floating-point support is enabled by an internal C library available within the PandA Bambu HLS tool: the library implements templatized floating-point functional units for basic arithmetic operations, comparisons, and standard-to-custom and custom-to-custom type conversion. Since templatized cores have been implemented as a library, separately from the HLS tool, they can be integrated into a generic application at any level without issues. These operators are then integrated, following one of the two flows just defined, into the application description that is then further optimized along the synthesis flow. The end result is thus featuring custom floating-point functional units for each of the required data types. Furthermore, floating-point cores, both standard or custom precision, are commonly implemented by state-of-the-art approaches as generic functional units from an RTL component library, thus they cannot be considered by the HLS flow nor optimized during its execution. Conversely, the offered implementation integrates the actual functional units IR into the input application IR enabling a fine grained optimization of their design. With this novel approach, the architecture of the floating-point functional units is optimized along with the whole application description resulting in ad-hoc improvements on the standard functional units design. The resulting accelerator design then features custom precision floating-point computation with application-specific implementations of the required arithmetic operators and mathematical functions.

Adoption of custom floating-point data types may result in many benefits for the generated hardware accelerator such as lower computational latency, lower resource usage and power consumption, and faster memory access due to the reduced bitwidth, as already discussed in Deliverable D3.1.

5.2 Partitioning to Distributed FPGAs

Once the AST is optimized and each node, i.e., a Relay IR instruction or operation, is annotated with its potential implementations and the required performance, DOSA has to perform the DSE and select the best possible implementation.

The DSE is done multiple times, each with different hyper-parameters. Such hyper-parameters are, for example, if the switching costs should be considered early or late, how many operations the DSE should consider for its current decision, or which target hardware to assume first, if multiple are available. One pass of the DSE consists of eight steps: In the first step, (cf. box ④ in Figure 13), for each operation, the implementation candidates (c.f. (2a) and (2b) in Figure 12) are sorted by performance, latency, or resource, depending on the overall optimization goal. Then, the best engine-type and streaming-type candidate are are analyzed for violations of any of their roofline borders. If only one type is possible, this one is selected as implementation *type* for this operation. Otherwise, the engine type is selected tentatively, since engines are in general more efficient. Please note, at this step, only the type of the micro-architecture is selected, not the concrete implementation itself.

Second, a greedy algorithm now goes through the AST and selects for each operation the contract with the lowest resource consumption that still fulfills the performance requirements. Here, this decision could be influenced by the consideration of *switching-costs*, as configured by the hyper-parameters. Switching-costs are the FPGA resources that must be spend to connect two *different* frameworks with adapters for the data-path. Hence, selecting two sequential operations with the same OSG could save FPGA resources. At the contrary, the second operation could have better available implementations, therefore it could be *worth the costs*. To compare all possible solutions, different DSE passes consider different numbers of sequential operations. If no implementation is available that fits within the resources of one FPGA, the operation is parallelized at this step. For example, a convolution that produces 64 output feature maps can be split in two with 32 output feature maps each. Those operations are later scheduled on different nodes.The third step then splits the FPGA nodes *horizontally*, if the selected contracts including switching costs exceed the targeted utilization of the FPGA nodes. Due to routing difficulties, it is not recommended to utilize more than 70 – 80% of specific FPGA resources. Horizontal splitting refers here to scheduling the operation that exceeds the resource budget and all its following operation on a new FPGA node that comes behind the current FPGA node. Hence, *horizontal* FPGA nodes form a pipelined model parallelism.

Fourth, the DSE pass ensures that all operations scheduled on the same node have the same parallelization factor. For example, if the first operation on the node is a split convolution, the following activation must be split as well. This is called *vertical* splitting, since the same pipeline step shared across multiple FPGAs and is another form of model-parallelism. As one example, Figure 16 visualizes these two different forms of model parallelism for a six layer CNN, distributed across five horizontal pipeline stages using nine FPGAs in total. Next, as fifth step, nodes that do not fulfill performance goals or over-saturate the network bandwidth are instantiated multiple times and the incoming requests are later distributed using a round-robin procedure. This is equivalent to (partial) data parallelism of the DNN model. Therefore, these parallel nodes only need to compute $1/X$ of the data, hence the required performance and bandwidth is reduced by a factor of X. The sixth step traverses the AST once more and selects for each FPGA node the best FPGA device, if multiple hardware targets are still possible. This step is using a roofline analysis as a basis to minimize the waste of unused bandwidth or logic resources, i.e. trying to select the device with the lowest, but still fitting *roof*. The seventh step combines sequential contracts, i.e. implementation candidates, of the same OSG to avoid the generation of unnecessary wrappers and data-path adapters. The last step of each DSE pass updates all performance annotations and checks if the performance goals are fulfilled at all levels.

Finally, after performing multiple DSE passes with different initial conditions and hyper-parameters, the draft with the lowest resource costs is selected, in case multiple drafts fulfill the performance goals. If no valid implementation could be found, the user is notified about the reasons. For example, the resource budget is too low, or one operation is not supported by any available framework. In the case the optimization goal is to minimize the resource footprint, i.e. using as few FPGA nodes as possible, some of the steps above are skipped.

5.3 Memory-Related Optimizations

In this section, we describe how the data management techniques described in Deliverable D3.2 are implemented and included into the EVEREST SDK. Such optimizations and the associated hardware generation process can be easily adapted to many tensor-based kernels like the ones present in the EVEREST use cases. Also, the same optimizations are valid for all variants of the target architecture, only with different parameters (e.g., the number of memory channels, the number of FPGA resources, the bus bit-width, etc.)

On-Chip Memory Sharing. We run Mnemosyne on the metadata produced by the compiler to generate the RTL of the on-chip memory architecture associated with each kernel. In particular, Mnemosyne uses the buffer compatibility graph to identify opportunities for sharing the physical on-chip memory banks without performance overhead [26]. Sharing opportunities can be exploited when distinct internal buffers have no overlapping lifetime and so they can share the same physical banks. Such memory architecture implements the logic to access the same memory banks from different kernel interfaces [14, 26]. Mnemosyne wraps the RTL kernel description (produced by HLS) with the resulting RTL description of the kernel memory architecture to expose only input and output ports to the computational units. This conceptual interface is then used for integration of the kernel into the Computational Unit (CU) in a transparent way.

Host-FPGA Double Buffering. This optimization requires changes in the CU wrapper to determine on which memory channel the CU should operate at each time. Based on the type of target architecture, it may be required to change also the configuration file (e.g., in the case of the Alveo boards) to specify how to attach more channels to the same CU. Finally, the host code must be updated to target the proper channel in each data transfer. Additionally, since we use two channels to implement double buffering, this can limit the number of outstanding memory transactions and, in turn, the maximum number of parallel CUs. However, in case of many channels and few CUs, Olympus (cf. Section 5.4.2) also separates input and output channels to simplify the control logic and improve logic connectivity of the FPGA resources.

Bandwidth Optimization. In the case of large channel busses, the hardware generation flow modifies the host code to interleave the input for the multiple elements before sending it to channels and de-interleave the output after receiving the results. The optimization only needs information on the bus bitwidth (e.g., 256 bits for the AXI links of the Alveo) and the data type bitwidth (i.e., 32, 64, or custom bits based on the data types). Both parameters are available from the user-supplied board specification and the compiler-supplied array information, respectively. From this, Olympus generates the CU *Read* and *Write* functions to split and aggregate the data into the appropriate number of lanes. The overall CU structure is then created by composing the *Read*/*Write* functions with multiple instances of the kernels. Similarly, the data reorganization portion of the host code can be generated with the same information by specializing the allocation functions of the host application.

Dataflow Optimization. This optimization is enabled by the compiler generating a kernel using subfunctions using streams, instead of one flat kernel function. The exact scheduling of the stages may not be straightforward, as the compiler has freedom to optimize the grouping for the best performance. Olympus then creates data streams among the subkernels for data communication. In order to stream data between the subkernels, data must be buffered when the subkernel does not operate on it in the same order that it is streamed or when the same values are reused multiple times inside the same subfunction. In most cases, this means that data streamed in gets stored in an internal buffer, then the data can be operated on using random access, and as each result is computed, it is streamed out. Data structures reused across multiple blocks must be streamed through these blocks and buffered inside them to maintain a consistent structure and avoid multiple hardware modules accessing the same data concurrently. This optimization does not require any changes in the host code. All optimizations are implemented as graph transformations on a connectivity graph that is extracted, optimized, and implemented inside Olympus.

Interface Modification for Supporting Custom Precision. Using the data representation that is defined in the previous HLS steps of EVEREST as an input, the data types are automatically changed in the implementation. Based on the HLS tools used for the kernel generation, there are different ways to specify custom data types. For example, in the case of Xilinx Vivado/Vitis, fixed-point implementations only require a redefinition of the data types before HLS using the given arbitrary-precision libraries. In the case of Bambu, custom floating-point implementations are specified in the exchange format between the compiler and the tool, and automatically synthesized by the tool. The conversion from/to double is generally implemented in the host code to save hardware resources. However, this requires to adapt the data allocation functions, which receive the input values in double but need to write fixed-point values in the FPGA buffers, and the functions to retrieve the results that must implement the opposite conversion.

5.4 System Integration

5.4.1 Network-attached FPGAs

Besides the roofline-based DSE, the ability to interface with all different kinds of specialist frameworks via the OSG infrastructure, linking all components together is another key ingredient for an organic compiler framework (cf. step ⑦ in Figure 13). Two levels of communication must be handled: first, the communication between different accelerator cores on one FPGA node, and second, the communication between nodes.

For intra-FPGA communication, the bandwidth of each interface is calculated, and based on this, interconnection FIFOs with a bit width of 64, 256, or 1024 are generated. The FIFOs are implemented via TCL scripts. Consequently, to connect to these FIFOs, the OSGs are required to generate wrappers to connect the individual frameworks with this standardized communication interfaces. Additionally, some frameworks may require a different serialization of the data. We support both the two alternative cases of grouping data (pixels) to further serialize them. First, all pixels of one channel, e.g., three channels for RGB, are grouped together, i.e., the tensor is transferred frame by frame. In the second case, the pixels of the same index of all channels are grouped together. Depending on the framework, these wrappers may need to transform the tensors and hence the resulting switching costs are taken into account during the DSE. DOSA is linking all components together in a global HDL file. Hence, the OSGs need to emit HDL code for instantiating their accelerators and wrappers in this HDL top module. These HDL code snippets must contain defined placeholders to replace the interface signals with the correct input and output. After all interfaces and modules have been created, the placeholders are replaced to connect the accelerator cores in the correct order.

For the inter-node communication, DOSA builds on the ZRLMPI framework [33, 32]. ZRLMPI is a one-click solution for compiling, optimizing, and deploying MPI applications on heterogeneous FPGA-CPU clusters. The framework provides FPGA cores and CPU software to synchronize FPGA and CPU nodes at run-time by implementing a subset of the MPI standard. Please note that ZRLMPI is used here as a hardware-agnostic heterogeneous synchronous communication layer, not as a programming model. However, the order of messages is important for distributed DNNs, we exploit the implicit synchronization between nodes by using the `MPI_Send` and `MPI_Recv` features of ZRLMPI. For example, if a large convolution is partitioned vertically on two FPGAs, the subsequent FPGA in the pipeline needs to receive the resulting activations in the correct order. Such a situation can be observed in Figure 16 between pipeline stages 2 and 3. Therefore, DOSA instantiates the necessary Message Passing Engines from the ZLRMPI framework and connects them to the accelerator cores within the FPGA node by generating a lightweight wrapper core to connect the different streaming interfaces. Additionally, this network adapter core has the communication plan of the node as table. This communication plan is generated by DOSA and applies all parallelizations that were inserted by the DSE, i.e. horizontal and vertical model parallelism and data parallelism. Here, also the abstraction level of the OSA helps, since the communication pattern of the operations can be derived in a straightforward way at this IR level.

Lastly, the network streams and memory buses are connected to the corresponding interfaces offered by the Shell of the selected target platforms (cf. [32, 29]).

5.4.2 Bus-attached FPGAs

The overall hardware architecture produced by the EVEREST system integration part is described as a block diagram that can be later imported into the toolflow for logic synthesis and bitstream generation. This approach allows us to integrate kernels and components generated with different flows, increasing the interoperability of the EVEREST SDK. The kernels can be indeed generated by Vitis HLS or other HLS tools (such as Bambu [11]), or directly described in HDL. This block diagram is generated by Olympus starting from its MLIR system-level description using the xDSL [5] library to perform transformations on MLIR using Python.

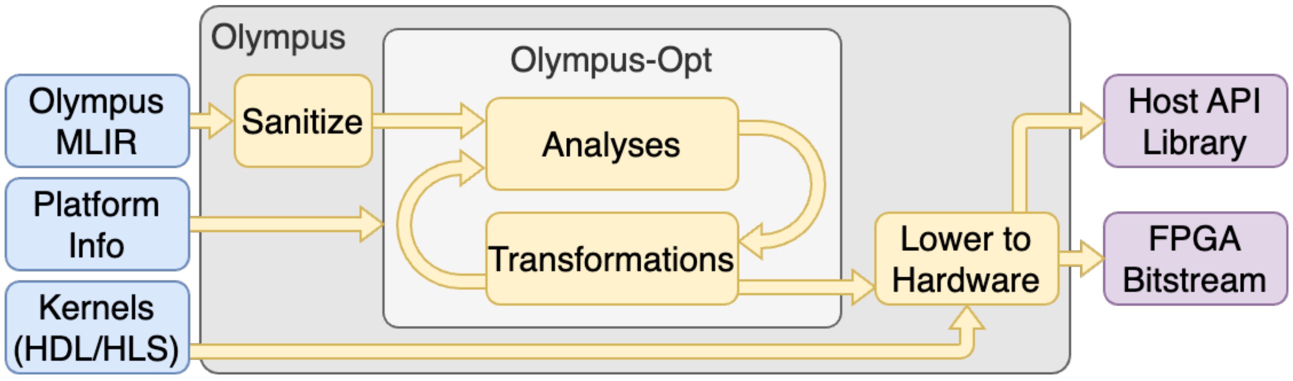
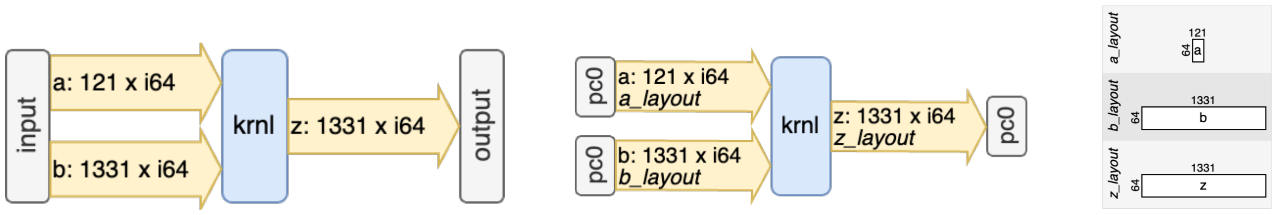


Figure 17 – Olympus flow diagram: starting from an MLIR system level description, platform info, and kernel implementations Olympus generates an optimized hardware architecture implemented as an FPGA bitstream and host API library.

A diagram of the overall flow is shown in Figure 17. The inputs to Olympus, shown on the left in blue are the Olympus MLIR description of the DFG (cf. Section 4.5), the FPGA platform details (including the type and number of FPGAs), and the kernel implementations. Olympus performs sanitation of the input, then iterates over the Olympus-Opt analyses and transformations to optimize the final DFG. Finally, the DFG is lowered to hardware and the output products, shown in purple on the right, are produced for both the host driver API



(a) Original input DFG (b) Sanitized DFG (c) Sanitized input layouts

Figure 18 – Visualization of a DFG with one kernel with two input and one output channel. The original input (a) is sanitized to be with PC nodes (b) and layouts (c) for each channel.

library and the FPGA bitstream.

Sanitize step. The first step is to sanitize the input Olympus MLIR to get a more hardware-oriented description. For example, given the input MLIR shown in Figure 18a, we use this step to derive a form that could immediately be passed to the hardware lowering step to create the system architecture (e.g., with connections to the FPGA memory channels). This allows the user to create the MLIR in a more convenient form without having to add redundant details. First, layouts are created for each channel. The layout is an additional attribute of the channel operators and represents the organization of the data when sent through the channel. The layout created at this stage is simply a width of one element and a depth of the `depth` attribute, as shown in Figure 18c. Additionally, `olympus.pc` nodes are created for each data channel connected to global memory (i.e. not connected to kernels on both sides). These are similar to kernel operations but instead represent the pseudochannel (PC) of global memory and are used as the terminals for data channels to main memory. These operations have one attribute (the `id` of the memory channel) and one operand (the channel connected to this PC). The direction is inferred by whether this channel is an input or output for the kernel it is connected to. In this stage, each channel to global memory is connected to one `olympus.pc` node and all `id` attributes are set to 0 (e.g., all virtual channels are connected to the same physical channel as this guarantees to generate a solution that can be always implemented). After these steps, the IR can be immediately lowered to HDL and synthesized into a working (but inefficient) design (Figure 18b).

Olympus-Opt. The next stage is an iterative series of analyses and transformations to obtain a more optimized system architecture. In addition to the sanitized input MLIR, this stage requires the FPGA target specification including: the number of global memory channels and their widths and the amounts of each available resource. Additionally, a resource utilization limit (default 80%) can be given. The analyses comprise of two main calculations. First, the target PC information and the attributes of each data channel are used to calculate a bandwidth utilization percentage. Second, the total resource availability and the kernel resource utilization are used to estimate an overall utilization. Using the results of these analyses, transformation passes can be chosen to alter the DFG to increase expected performance. These transformations implement some of the solutions proposed in Section 5.3, such as:

- **Channel reassignment:** Data channels connected to PC nodes and data channels of `complex` type are distributed across the channels available on device to increase bandwidth utilization. Figure 19 shows how Figure 18b would be transformed with each PC node being assigned a separate `id` number, to represent a mapping onto separate physical PCs.

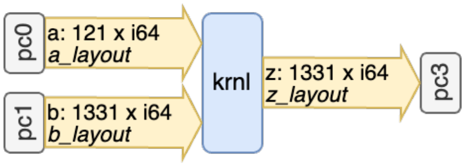


Figure 19 – Sample result of applying channel reassignment to Figure 18b. Each PC node has been given a different `id`.

- **Replication:** If the resource utilization is low, the entire DFG can be replicated for increased parallelism, up to the resource utilization limit. Figure 20 shows how Figure 18b would be replicated twice. Each operator is replicated and given a new identifier. Each replicated PC node is given the same `id`. Replication can gain near ideal speedup, however a high degree of replication reaching near 100% utilization of a resource induces routing congestion and therefore a longer critical path. Replication should be used carefully, utilizing other optimizations for more performance.

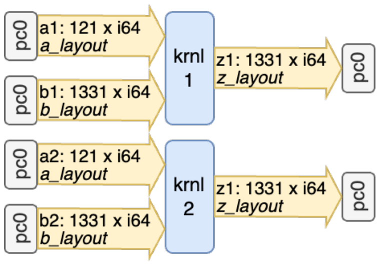


Figure 20 – Sample result of replicating Figure 18b two times.

- **Bus widening:** If data widths are evenly divisible into PC widths, kernels can be replicated such that multiple instances use the full PC. For instance, a kernel with a 64-bit data input using a 256-bit PC can be replicated four times so each kernel's data uses one of four lanes in the PC [39]. Figure 21 shows how Figure 18b would be affected by bus widening for a 128-bit bus.

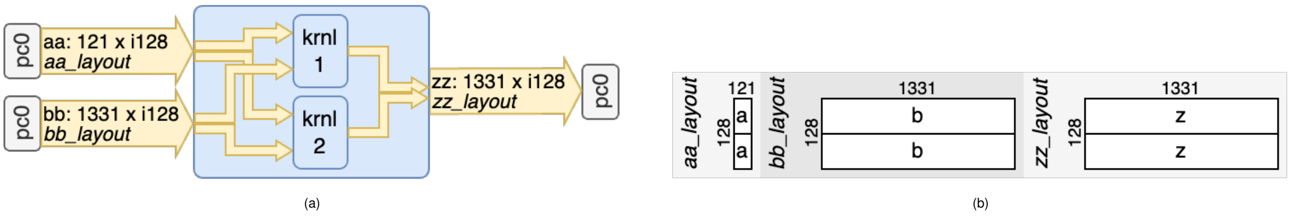


Figure 21 – Sample result of applying bus widening to Figure 18b with a bus width of 128. Each channel has been widened by $2\times$, and two kernels are instantiated. The layouts b have each data array replicated in parallel.

- **Bus optimization:** To increase bandwidth utilization, channels can be grouped to interleave data [40]. The Iris algorithm can split data into smaller chunks and interleave them with other arrays to compact them on a bus with a given width. Figure 22 shows how Iris combines the a and b channels in Figure 18b into a 128-bit bus. In the new single channel, the layout reflects the result of the Iris algorithm with the b array broken up to achieve the most compact result. The Iris algorithm can achieve over 95% bandwidth efficiency for a channel, compared with $\tilde{4}5$% efficiency of a naive layout. Each data channel is made twice as wide and the layout is modified to act as two "lanes". These channels are connected to a super-node encapsulating two kernels. When this is lowered to hardware, the data mover modules separate the "lanes" and send the data to the correct kernels. With sufficient resource availability, this optimization achieves near ideal speedup for the number of replications.

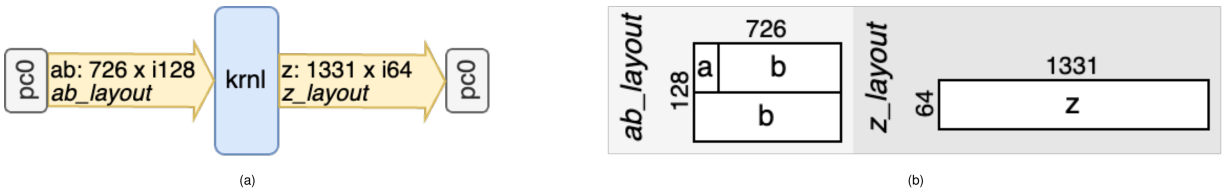


Figure 22 – Sample result of applying the Iris algorithm to Figure 18b to combine the a and b channels on a 128-bit bus. a and b are interleaved in the layout b of the new ab channel.

- **PLM optimization:** If the characteristics of the data accesses are known, the physical memories can be shared for area efficiency [26]. Memories or interfaces can be shared based on spatial or temporal compatibility, respectively. This information can be detected by static compiler analysis and supplied as additional information to enable this optimization. This optimization saves on hardware resources, often to a high enough degree to allow for additional compute unit replication and therefore speedup.

Lower to Hardware. After the Olympus-opt passes, we can effectively generate the hardware system architecture. Channels connected to `olympus.pc` nodes are connected to the PCs on the device. For the Alveos, this is configured in the `*.cfg` file input to the Vitis tool. Data channels with the `stream` type are instantiated as FIFOs of the specified `depth`. `small` type channels are instantiated as Private Local Memory (PLM) in BRAMs so data can be randomly accessed, but does not need to be sent out to global memory. These memories can be shared using Mnemosyne-generated PLM architectures. `complex` type channels are connected to the device PCs so the kernels can use arbitrary pointers to access this data. Channels with Iris-generated layouts are instantiated with adapters generated by the Iris tool to pack or unpack the data in a way the kernels can use. For Xilinx devices, these modules are connected in a Vivado block diagram. One Vitis HLS module is instantiated alongside the kernels to bridge the global memory and the kernels and includes the PLMs and data moving modules. If a kernel is connected to a `complex` channel, this kernel has an AXI port that connects directly to the global memory. Additionally, Olympus generates a host API library for initializing the device, creating on-device data buffers, moving data between host and device memory, and initiating kernel execution (cf. Section 6.1).

6 Code Generation and Runtime Integration

Sections 3-5 describe the languages, intermediate representations, transformations and hardware generation capabilities of the EVEREST SDK. In this section we describe how the final implementation is packaged and deployed as a hand-over to the runtime system. As discussed in previous deliverables, the SDK can generate pure software versions that run on CPUs. Such versions can be used by the runtime as alternative variant if special FPGA resources are not available. In general, the compilation flow can generate multiple software and hardware variants. In the following we describe the interfacing with the runtime and auto-tuning support described in Deliverable D5.1.

6.1 Host Code Generation

For transparent execution, the EVEREST SDK must generate the proper part of the host code that invokes the accelerator. This involves two parts: the compiler (which produces the parts that are still executed in software along with the methods to invoke the accelerator) and hardware generation flow (which has information about how the data must be rearranged based on the applied transformations). This part includes several lowering passes.

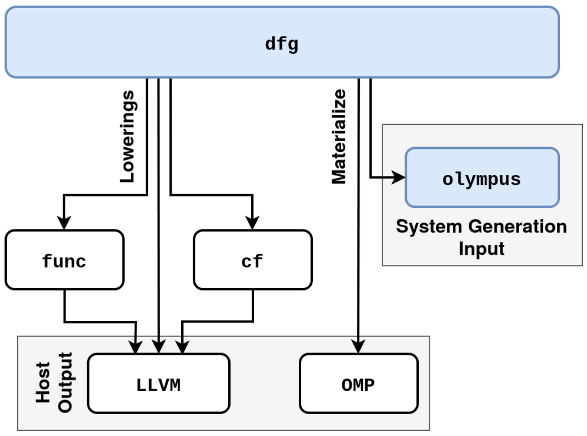


Figure 23 – MLIR dialects used in the compiler and their dependencies between each other to obtain the host code.

Figure 23 gives an overview of the dialects we rely on for the progressive lowering of the initial representation of the graph to LLVM IR. A key aspect of the lowering, however, is that offloading is simply controlled by the keyword `offloaded` during instantiation, as shown in Figure 24. This change will produce a wrapper interacting with the Olympus driver. This offloading choice again is completely transparent and can be changed with only the addition or removal of the `offloaded` keyword.

```
dfg.instantiate offloaded @mul inputs (%x_rx, %y_rx)
    outputs (%z_sx) : (i64, i64) -> i64
```

Figure 24 – To mark a node as offloaded, the necessary keyword has to be added.

During system integration, Olympus generates a driver API for data movement and kernel execution. The “Organize Data” method will pack all kernel inputs in the layout that the “Read” modules expect. The “Send Data” method transfers this data to the device global memory. The “Start Kernel” method initiates the kernel execution. The “Receive Data” and “Reorganize Data” methods transfer the results from global memory back to the host and unpack them for consumption of subsequent tasks. These methods are compiled into a library for linking into the final executable.

6.2 EVP: The Offloading Linker

Compiler solutions that enable automatic offloading on heterogeneous devices are often faced with increased linking challenges. For programming models that mix host and device code, such as CUDA and SYCL, these are typically addressed with an "offloading linkage" step. Challenges include separating host and device compilation units, mixing diverging ABIs (especially SYCL), and integrating device binaries into the host executable. In EVEREST, we needed to address similar challenges, so we introduced the `evp` tool.

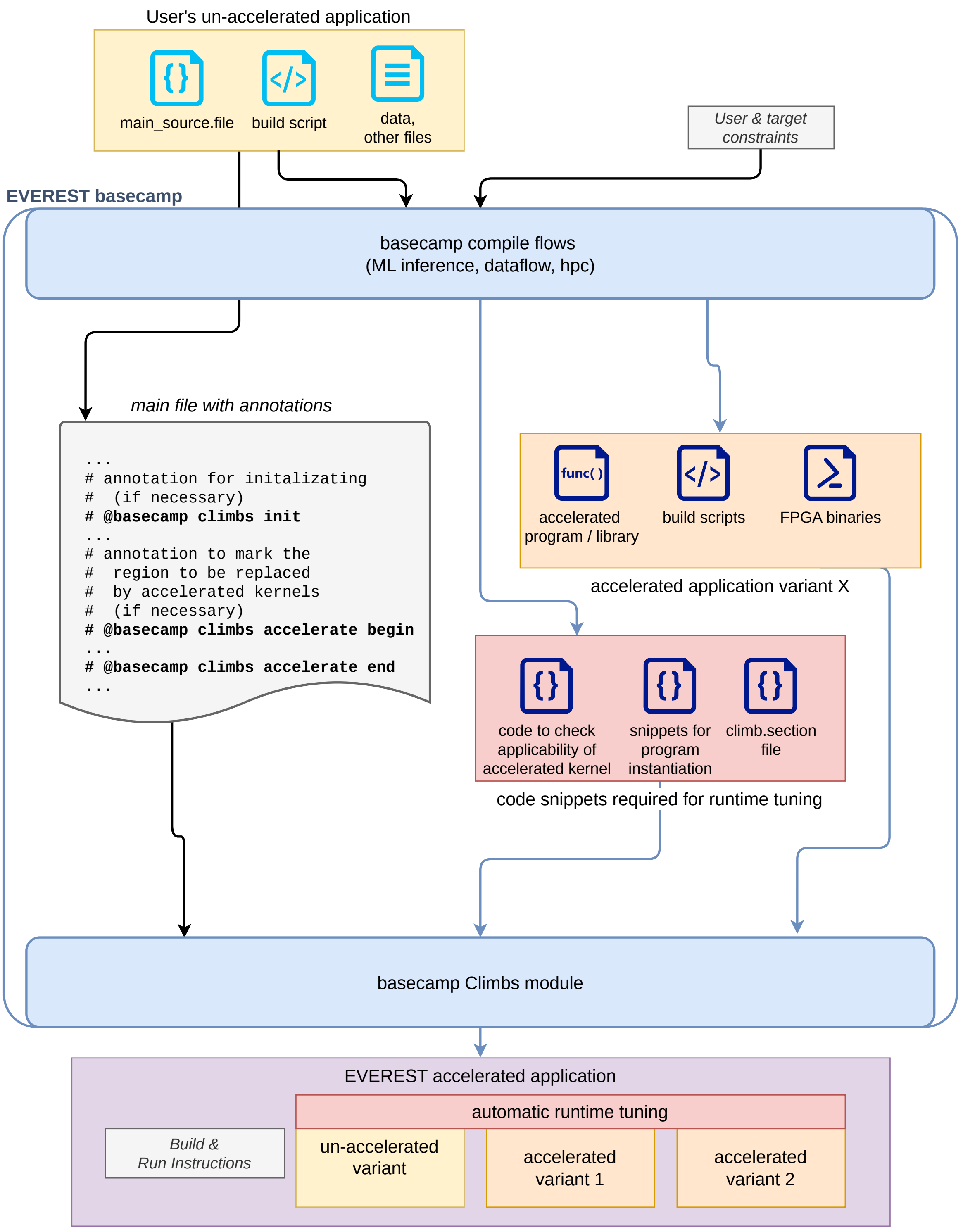
The EVEREST platform dialect `evp` and its associated tool with the same name perform this host/device module instantiation. The `evp` dialect defines offloading primitives that will be implemented by compatible EVEREST backends, e.g., Olympus, in the following compilation stage. The `evp` tool performs rewrites on the host code such that the host/device boundaries in the `dfg` dataflow graph are replaced by these primitives. `evp` outlines the device modules for code generation, and creates the descriptor file for the backend, which includes the relative paths to the device compilation units. The resulting modules can then be compiled using the default toolchain of their respective target, and their artifacts eventually linked using standard C linkage into a complete host executable.

6.3 Bundling Variants with `basecamp Climbs`

As mentioned above, `EVEREST basecamp` is the tool to start all *EVEREST endeavors*, which includes the start of each compilation and optimization flows. But additionally, after these tool flows finished, the resulting accelerated (parts) of the applications need to be bundled together. This includes also the combination of different variants of the same kernel, whereas each variant is optimized for different runtime conditions.

To instantiate the runtime tuner and combine it with the available application variants, basecamp offers the module `climbs`. Where one *climb* represents the combination of different EVEREST tools, so different *parts of the EVEREST endeavor*. How the compile tool flows the and the climbs module interact is shown in Figure 25. At the top of Figure 25, the user interacts with basecamp to trigger the compilation of her application as described in the previous sections of this deliverable. The compilation tool flows can be invoked multiple times to generate different accelerated variants of an application. Once all variants are generated, the user invokes the basecamp climbs module, either via CLI or its python API. One example of using the basecamp climbs python API is presented in Listing 2. In this example, the traffic prediction use case is accelerated using the ML inference flow and DOSA. In the beginning of Listing 2, the user creates a new climb and then adds all necessary files and directories for this unaccelerated (i.e. *original*) application to the climb. The `add_file` function also needs the programming language of the files, so that the right code snippets to instantiate can be identified. Using the `copy` keyword indicates to basecamp that the file does not need to be analyzed. Next, the `add_module` call adds the accelerated variant generated by the basecamp compile tool flow to the climb. This is done by using the path to the `.section` file generated by the basecamp compile flow, in this example by DOSA. Finally, the complete combined application is emitted.

Back to Figure 25, it shows in the center the interaction between the different basecamp modules that enables this automatic bundling of variants. Please note three objects: On the very right-hand side, the orange-colored box shows the FPGA binaries with corresponding drivers and deploy scripts, which form together one accelerated variant of the application. Then, to the left, the internal data structures between the basecamp modules are shown in the red-colored box. To be able to allow a simple interaction for the user, as presented in Listing 2, the compiler also needs to generate code that checks if the accelerated variant can be deployed or executed at runtime, e.g. to check if sufficient FPGAs of the right type are available. Additionally, the compiler needs to provide code snippets in the right programming language to instantiate the execution of the accelerated program at runtime. Both is required to allow the runtime tuner to (i) decide which variant to deploy, e.g. based on availability or on the real-time state and conditions of the application, and (ii) to then call the right driver function. Sometimes, basecamp can not determine the exact parts of the original application that should be part of the runtime tuning. For example, in the ML inference case, DOSA generates an equivalent call for the inference function of a DNN object. However, sometimes, the software version of the ML application needs library specific loading of weights, depending on the requested inference kernel. This loading of weights does not need to be executed in the accelerated version, since the weights are already loaded to the FPGAs at deployment time. Therefore, basecamp offers the option to annotate the original programs with comments like `@basecamp climbs init`, `@basecamp climbs accelerate begin`, and `@basecamp climbs accelerate end`. These annotations enables the user to indicate to basecamp the places where the initalization of the FPGA deployment should happen (best) and which exact parts of the original programm should be replaced by the


User's un-accelerated application
main_source.file
build script
data, other files
User & target constraints
EVEREST basecamp
basecamp compile flows (ML inference, dataflow, hpc)
main file with annotations
accelerated program / library
build scripts
FPGA binaries
accelerated application variant X
code to check applicability of accelerated kernel
snippets for program instantiation
climb.section file
code snippets required for runtime tuning
basecamp Climbs module
EVEREST accelerated application
automatic runtime tuning
Build & Run Instructions
un-accelerated variant
accelerated variant 1
accelerated variant 2

```
...
# annotation for initalizating
#  (if necessary)
# @basecamp climbs init
...
# annotation to mark the
#  region to be replaced
#  by accelerated kernels
#  (if necessary)
# @basecamp climbs accelerate begin
...
# @basecamp climbs accelerate end
...
```

Figure 25 – Interaction between basecamp compilation flows and climb module for the bundling of variants.

accelerated version. Consequently, the explicit information provided by the user combined with the internal information of the basecamp modules allows the automatic generation and instantiation of an accelerated application containing one or many accelerated versions.

```
from ebc import basecamp
e_climbs = basecamp.climbs

# first, create a new Climb
e_climbs.create('accelerated_tpred_simple', 'tpred_integrated_v1/')
# out: Climb tpred_integrated_v1/accelerated_tpred_simple.climb created successfully.

my_climb_file = 'tpred_integrated_v1/accelerated_tpred_simple.climb'
app_dir = 'traffic_prediction/tpred_app/'

# now, we add all the files of the SW only app
# the second argument indicates the programming langauge and it indicates the right snippets of ↩
    the accelerated program to instantiate
e_climbs.add_file(app_dir + 'app.py', 'python', my_climb_file)
e_climbs.add_file(app_dir + 'Dockerfile', 'docker', my_climb_file)
# 'copy' means the files are copied without modification/analysis, also directorys are copied ↩
    recursively
e_climbs.add_file(app_dir + 'client.py', 'copy', my_climb_file)
e_climbs.add_file(app_dir + 'data/', 'copy', my_climb_file)
e_climbs.add_file(app_dir + 'Readme.md', 'copy', my_climb_file)
e_climbs.add_file(app_dir + 'requirements.txt', 'copy', my_climb_file)

# now, we add the DOSA compiled files as 'variant'
# the .section file is generated by DOSA / the basecamp compile flows
e_climbs.add_module('build_dirs/etp_v1/ml_inference.section', my_climb_file)

# finally, emit the Climb
e_climbs.emit(my_climb_file)
```

Listing 2 – Example interaction with the EVEREST climbs module by the example of an ML inference application.

With `basecamp Climbs`, the EVEREST SDK offers a solid and generic infrastructure to trigger the compilation flows to generate, and to manage those variants to interact with an autotuner such as mARGOt. In Deliverable D4.2 Figure 8, we showed how easy it is to get the DSL compiler to export multiple different implementations for a given tensor expression. From EKL, in the context of the radiation module of WRF, we focused on static optimization of the kernels and thus no variants were exported to the runtime system for the final results reported in Deliverable D6.5. The reasons for this are two fold. First, we wanted to deploy the best performing solution to show the potential of our DSL approach. Second, the parameters and constants were known to the compiler so no practically relevant runtime variation was expected that would require adaptive execution of the kernels. That said, nothing prevent further exploration of adaptive execution under more dynamic weather scenarios and workload variations in the system. This, however, remains a matter to be addressed after the end of the EVEREST project. For other use cases, `basecamp Climbs` were used to interact with the mARGOt autotuner. This is the case of ML inference for traffic prediction (see example in Listing 2) and for managing the HW/SW versions of the PTDR kernel in a virtualized environment.

7 Final Assessment

In this section we provide a final assessment as to how the tools described in this deliverable fare with the requirements defined for the language and compilers (Section 7.1), and for the hardware generation tools (Section 7.2). This assessment relates to the EVEREST requirements described in in Deliverable D2.2 and updated in Deliverable D2.5. The section closes with a brief positioning statement with respect to the state of the art in Section 7.3.

7.1 Requirements: Language and Compilers

7.1.1 REQ3.1 – WRF expression abstraction

- **Priority:** Must have.
- **Notes:** Kernel support for WRF simulations. Language support for expressions in numerics (tensors, linear algebra).
- **Assessment:** The abstractions of languages like cfdlang and teil were extended in the `esn` abstraction to better support the WRF radiation module (see Section 3.1). These abstractions effectively fulfill this requirement.

7.1.2 REQ3.2 – WRF Fortran integration

- **Priority:** Must have.
- **Notes:** Expression abstractions should be callable from within Fortran code. Either by annotations or inline code modifications, user can write expressions within Fortran code for WRF.
- **Assessment:** The C++/Fortran interoperability was successfully test performed at start of the project, including a non-seamless integration via text literals tested. A seamless integration with Fortran became irrelevant for the project for two main reasons, namely, the progress of the MLIR Fortran integration and, more importantly, the fact that an integration via the WRF module system turned out to be unavoidable for linkage reasons.

7.1.3 REQ3.3 – ML integration

- **Priority:** Should have.
- **Notes:** Integration with machine learning frameworks. Allow importing models to hook into the code generation process for EVEREST specific transformations.
- **Assessment:** The EVEREST SDK supports the community standard ONNX and pytorch (JIT) to be synthesized to (distributed) FPGAs. Additionally, although this was not a high-priority requirement, we implemented a prototype dialect with `jabbah` (see Figure 8) for frontend integration. In the middle-end, the DOSA framework is integrated with basecamp. It can also produce the MLIR dialect for interfacing with Olympus.

7.1.4 REQ3.4 – Streaming support

- **Priority:** Could have.

- **Notes:** Language support for streaming workflows with highly dynamic loads. Enable compiler reasoning for reconfiguring streaming oriented computations. Expected to support traffic use case. This requirement was under specified in the project, because the applicability to the use cases was not obvious.
- **Assessment:** While this was not explicitly tackled in the project, the Ohua/Condrust compiler does support streaming applications [41]. At the beginning of the project we wanted to connect the frontend with the HyperQueue task manager. Since HyperQueue is no longer needed for traffic use case, this became irrelevant.

7.1.5 REQ3.5 – Integration with compiler frameworks

- **Priority:** Should have.
- **Notes:** For stability, reusability and extensibility, compiler work should build on top of established frameworks (e.g., LLVM and MLIR for numerics, Haskell or alike for dataflow). By contributing to open source frameworks, the results from EVEREST can be used by the community at large. By integrating with these frameworks, EVEREST can reuse and extend existing methods.
- **Assessment:** The integration in MLIR described in Section 4 effectively fulfills this requirement.

7.1.6 REQ3.6 – Compiler transformations for kernels

- **Priority:** Must have.
- **Notes:** At the middle-end, the compiler must include a framework for transformations to manipulate code and optimize for the EVEREST platform. For numerics, this should include affine transformations (polyhedral) with support for stencils and other linear algebra primitives.
- **Assessment:** Algorithmic and polyhedral transformations were demonstrated in [14, 39], effectively fulfilling this requirement.

7.1.7 REQ3.7 – Compiler transformations for dataflows

- **Priority:** Could have.
- **Notes:** For dataflow programs, the compiler should include semantic preserving rewrites for performance and energy optimizations, while retaining determinism. This should extend on previous work on optimization for dataflow programs (including mapping, graph rewrites and I/O batching).
- **Assessment:** The Ohua/Condrust compiler implements semantic rewrites, described in [41]. The connection to more advanced energy-aware optimizations like in [16, 17] remains future work.

7.1.8 REQ3.8 – Multi-target code generation

- **Priority:** Must have.
- **Notes:** The source-to-source compiler should generate code for different targets. Code written in high-level expression abstractions should translate to pure software (C/C++ code), or software with offloading to accelerators (e.g., FPGA).
- **Assessment:** From the abstractions for kernels (Section 3.1) code can be generated automatically for CPU and FPGA. While we have used experimental linearization as C/C++ code, MLIR allowed us to generate code direclty via LLVM. Additionally, MLIR also enables code generation for GPU targets, which could be used with some little additional effort. The `dfg` dialect include annotations for HW offloading. The downstream tool flow from `dfg` and `olympus` allow to transparently deploy different implementations (a pure software one or version with FPGA offloading). This requirement is thus fulfilled.

7.1.9 REQ3.9 – Generation of tunable parameters

- **Priority:** Must have (for adaptable kernels).
- **Notes:** To enable autotuning, the compiler must produce descriptors of solutions to interface with mARGOt. From high-level abstractions, the compiler should extract knobs and parameters that are key to modifying performance and/or energy efficiency.
- **Assessment:** For kernels, we demonstrated the capabilities for variant generation. See for instance variants with varying polynomial degrees in [39]. This was demonstrated in the 4th EVEREST project webinar, “Domain specific languages for heterogeneous and emerging computing systems”. In the particular case of WRF, no adaptable kernels were recognized by the use case providers. Variants with and without hardware acceleration can be generated for ML inference and from the `dfg` abstraction.

7.1.10 REQ3.10 – Interface to HLS

- **Priority:** Must have.
- **Notes:** The compiler should enable a downstream HLS flow. The compiler must export code (or an intermediate representation thereof) to the HLS flow, including behavioral descriptions of the kernels and configuration information for memory modules. The representation must be synthesizable (e.g., no dynamic allocation of memory).
- **Assessment:** Interfacing from high-level languages to the HLS downstream flows, including Bambu and Vivado has been demonstrated in webinars, in [39, 14, 13, 31, 28], and discussed in Deliverable D4.2. This requirement is thus fulfilled.

7.1.11 REQ3.11 – cFDK/OC-Accel software integration and language compatibility

- **Priority:** Must have.
- **Notes:** The software should be compatible with the cFDK/OC-Accel API. The software part of the kernels that are being mapped to the FPGA should be built in a way that allows the seamless integration with the API specifications of cFDK (e.g. C/C++/Python sockets) and/or OC-Accel frameworks (e.g. C/C++/libocxl).
- **Assessment:** The DOSA flow supports the cFDK API completely (see Section 4.7.2). Support downstream from the `olympus` abstraction is in progress and will be finished by the project’s end. The EVEREST SDK offers tools to accelerate key HPC and ML kernels by offloading them to high-performance FPGAs. While the bandwidth between the HPC compute nodes and on the bus-interfaces between the CPU and the FPGA-accelerators within the compute nodes is important, the ability to coherently access address memory across the CPU and accelerator complex was not identified as an important element during the analysis of the use-cases. Furthermore, the OpenCAPI interface, which we were planning to use to link the FPGAs to the CPUs did not achieve the wide-spread adoption we were expecting at the time of the writing of the proposal. Therefore, we chose to implement the FPGA-accelerated HPC system that will be used to demonstrate acceleration of the weather codes based on standard PCIe-attached FPGA-accelerator cards (Xilinx Alveo). This approach will naturally extend to CXL-attached accelerators, which rapidly gain adoption across the industry (cf. Deliverable D6.2).

7.1.12 REQ3.12 – Reasoning about heterogeneous applications

- **Priority:** Must have.

- **Notes:** Compiler optimizations must consider offloaded kernels. The source to source compiler must differentiate between offloaded and host functions to apply parallelizing transformations to host code only.
- **Assessment:** For applications fully described using the abstractions of the SDK (ML, EKL and Ohua/dfg), the tool flow modifies the code on the host to implement different parallelization schemes. A good example of this can be seen in [39]. Similarly, also the developed Operating Set Architecture used by DOSA optimizes the implementation of operations that can not be offloaded to an accelerator [28, 31]. In the particular case of WRF, the Fortran code around the extracted kernels is not modified as moving the entire legacy RRTMG implementation into DSL was out of scope.

7.1.13 REQ3.13 – Glue code generation for heterogeneous applications

- **Priority:** Should have.
- **Notes:** Generate interfacing code for offloaded kernels. The source to source compiler must generate code that allows the interfacing between host and accelerator.
- **Assessment:** `olympus` and `evp` dialects generate the glue code, effectively fulfilling this requirement.

7.1.14 REQ3.14 – Abstractions for offloaded kernels

- **Priority:** Should have.
- **Notes:** Provide abstractions for marking a kernel as *to-be-offloaded* in a high-level algorithm. The high-level dataflow language provides abstractions for marking offloaded kernels, easing development of the application.
- **Assessment:** The syntax extension described in Section 3.3 and Section 4.3 fulfills this requirement.

7.2 Requirements: HLS and Memory Design

7.2.1 REQ4.1 – C/C++ support

- **Priority:** Must have.
- **Notes:** C/C++ support for HLS of descriptions coming from DSL compiler. The HLS tool should support C/C++ code from the use case applications.
- **Assessment:** Bambu provides full coverage of the C/C++ constructs employed in the use case applications, as described in Deliverable D4.6, Section 2.5. The requirement is thus fulfilled.

7.2.2 REQ4.2 – Bambu LLVM IR support

- **Priority:** Must have.
- Notes: Low-level integration with DSL compiler. The HLS tool should support a version of LLVM consistent with the one used by the DSL compiler.
- Assessment: Bambu currently supports up to LLVM 16, which is compatible with the version used by the DSL compiler. The requirement is thus fulfilled.

7.2.3 REQ4.3 – Bambu MLIR dialect support

- **Priority:** Can have.
- **Notes:** Direct synthesis from MLIR dialects. It may improve the final performance raising the abstraction level. At least, it should support integration with the affine dialect.
- **Assessment:** Bambu can synthesize MLIR code lowered and translated to LLVM IR in the frontend. The requirement is thus fulfilled.

7.2.4 REQ4.4 – HLS Verilog output

- **Priority:** Must have.
- **Notes:** HLS generates RTL Verilog code as output. The Verilog code must be synthesizable with the EVEREST backend flow.
- **Assessment:** The EVEREST backend flow for Verilog kernels is currently logic synthesis and implementation with the Xilinx toolchain, which is fully compatible with Bambu. The requirement is thus fulfilled.

7.2.5 REQ4.5 – HLS VHDL output

- **Priority:** Should have.
- **Notes:** HLS generates RTL VHDL code as output. The VHDL code must be synthesizable with the EVEREST backend flow.
- **Assessment:** The EVEREST backend flow for VHDL kernels is currently logic synthesis and implementation with the Xilinx toolchain, which is fully compatible with Bambu. The requirement is thus fulfilled.

7.2.6 REQ4.6 – Top function specification

- **Priority:** Must have.
- **Notes:** The code to be synthesized must be in a stand-alone function that must be specified. The top function could be specified as an annotation to the code, command line parameter, or option files.
- **Assessment:** Bambu synthesizes a kernel starting from a user-defined top function, as described in Deliverable D4.6, Section 2.5. The requirement is thus fulfilled.

7.2.7 REQ4.7 – Block level/Top component interfaces

- **Priority:** Must have.
- **Notes:** The protocol to interface with the top module has to be specified. The start, done, etc. protocol of the top component has to be defined and compatible with the EVEREST platform.
- **Assessment:** The protocol for control signals has been defined, expected signals are `clock`, `reset`, `start_port` (inputs) and `done_port` (output). Bambu correctly generates such signals and Olympus generates their drivers. The requirement is thus fulfilled.

7.2.8 REQ4.8 – Port-Level interfaces

- **Priority:** Must have.
- **Notes:** I/O interface protocols added to the individual function arguments. The definition of the protocol should be defined through code annotations.
- **Assessment:** The protocol to access function arguments can be specified through pragma annotations or an XML file, as described in Deliverable D4.6, Section 2.5. The requirement is thus fulfilled.

7.2.9 REQ4.9 – Bambu Vivado HLS I/O interface interoperability

- **Priority:** Can have.
- **Notes:** Annotations specifying the I/O protocols interface compatibility. Block/port level interfaces should use the same annotations used by Vivado HLS.
- **Assessment:** Protocols and pragma annotations are fully compatible with the ones used by Vivado HLS. The implementation of the `array` and `m_axi` interfaces correspond to the Vitis HLS BRAM and AXIm interfaces, respectively. The requirement is thus fulfilled.

7.2.10 REQ4.10 – Technology options specification

- **Priority:** Must have.
- **Notes:** The HLS tool accepts inputs for optimization, clock constraint and resource constraints. These technology constraints will passed as input options.
- **Assessment:** Bambu exposes options that allow the user to select the target FPGA, clock period, and optimization level, as described in Deliverable D4.6, Section 2.5. The requirement is thus fulfilled.

7.2.11 REQ4.11 – Bambu Data flow annotations

- **Priority:** Should have.
- **Notes:** HLS Data flow support. Dataflow style applications could be specified by code annotations.
- **Assessment:** Only a preliminary design has been defined for the support of dataflow applications in Bambu.

7.2.12 REQ4.12 – Bambu OpenMP support

- **Priority:** Can have.
- Notes: OpenMP for pragma synthesis support. The body of OpenMP parallel loop needs to be in a separate function.
- **Assessment:** Only a preliminary implementation has been defined for the support of OpenMP applications in Bambu.

7.2.13 REQ4.13 – Bambu floating point precision

- **Priority:** Can have.
- **Notes:** Floating point variables may use a custom floating precision data type. Allow optimizations of scientific and machine learning kernels.
- **Assessment:** Bambu supports the customization of floating-point formats through the TrueFloat library, as described in Section 5.1.2. The requirement is thus fulfilled.

7.2.14 REQ4.14 – cFDK/OC-Accel top component interface

- **Priority:** Must have.
- **Notes:** Interface definition of the top component being integrated with cFDK/OC-Accel frameworks. The top-level component of the functionality that will be mapped to the FPGA must be compatible with the cFDK ROLE interface (AXIlite, AXIs, AXIm) and/or OC-Accel Action interface (AXIlite, AXIm).
- **Assessment:** The AXIm protocol is fully supported by Bambu, while AXIlite and AXIs will be integrated in the future. Since the project reacted to market developments, we integrated with the Alveo Shells instead of OC-Accel (cf. REQ3.11). Here, Olympus and Bambu fully integrate the required interfaces. Integration with cFDK/OC-Accel has been defined and partially automated.

7.2.15 REQ4.15 – Memory interfaces

- **Priority:** Must have.
- **Notes:** Standard interfaces for memory accesses. The HLS-generated kernels and the memory modules should have a common interface format.
- **Assessment:** Olympus successfully integrates accelerators generated by Bambu with memories and other communication modules.

7.2.16 REQ4.16 – Software-level support

- **Priority:** Must have.
- **Notes:** Software code to interface with the accelerators. The accelerators should be invoked with custom OS drivers.
- **Assessment:** Olympus generates an API library for invoking the kernel using the underlying platform drivers. This library can be linked to the host executable. Manual process tested via PTDR, automation in progress.

7.2.17 REQ4.17 – Hardware/software data sharing

- **Priority:** Must have.
- **Notes:** Data allocation must be compatible with hardware memory interfaces. The software-level data allocation should be performed in a way that hardware can access the data.
- **Assessment:** Kernel source code must use ETL or primitive data types. Olympus and Bambu understand these types and Olympus' host API library can organize them for correct kernel access. Manually tested via PTDR, automation in progress.

7.3 Advancing the State-of-the-art

Advances with respect to the state of the art have been demonstrated in the several publications cited in this deliverable. Major highlights are summarized in the following:

- High-level programming abstractions: EVEREST contributed to (implicit) stateful dataflow programming [41] and to tensor DSLs. The latter is an evolution of cfdlang [36] and TeIL [35] which, as opposed to many other tensor DSLs, include a formal semantic specification with provable properties. The extensions described in Section 3.1, Section 4.2 and Section 4.1 advance the state of the art with subscripted subscripts and seamless integration of custom data representations (compared to `linalg`) via `base2` [13] (compared to `tosa`).
- Weather modeling: Improving and accelerating the large legacy code base at the core of today's weather and climate models is the matter of plenty of efforts and is known to be extremely hard. There are two lines of research when it comes to acceleration and improved scaling of weather and climate codes on modern, heterogeneous hardware. The first approach is to re-compile the legacy code base predominantly written in Fortran for, e.g., modern GPU accelerated backends (`https://www.research-collection.ethz.ch/handle/20.500.11850/644620`), but this approach has inherent limitations because the low-level Fortran code does – in many cases – not expose the high-level data and control structure required to optimize code for modern hardware. The second approach focuses on building parts or the complete models from scratch with DSLs (https://www.pasc-ch.org/projects/2021-2024/kilos/index.html, https://gmd.copernicus.org/articles/16/2719/2023/) and developing new compilers which can target heterogeneous hardware backends, e.g., the Open Earth Compiler [15], GridTools [2], and GT4Py [23]. In EVEREST, with considerable effort, we showed that it is possible to gradually inject DSL expressions into the complex existing WRF code base. We tackle the radiation module, which experts acknowledge as one of the most challenging code bases. Our implementation can also be used from more modern initiatives such as ecRAD[2]. Apart from integrating into WRF, we also address the challenge of deploying components into accelerators on FPGA.
- MLIR ecosystem: We have actively contributed to the MLIR community, leading to upstreaming efforts (`ub` and `base2` from Figure 8) and uptake by project partners. We also proposed a novel approach to handle dependent types in MLIR to support the language extensions for tensors (see Section 3.1). In the area of dialects for FPGA design, we propose the simple, yet effective, dataflow `dfg` dialect that offers a higher-level abstraction to reason about resource sharing (compared to CIRCT `https://github.com/llvm/circt`). It offers a transparent, vendor-independent device model (vs. Noctua, ESSPER [38]).
- HLS and system-level design: EVEREST builds on top of established HLS tools (Bambu, Vivado/Vitis HLS) to introduce new methodologies (Section 5.1) and advanced system design features (Section 5.3 and Section 5.4). The introduction of MLIR-based loop pipelining [7] stems from a collaboration with the SODA project [4], and the code has also been integrated into the `soda-opt` repository. Unlike `soda-opt` and ScaleHLS [48], which only present simulation results, in EVEREST, we can deploy HLS kernels on actual FPGA boards, identifying bottlenecks that only arise when the kernel is integrated with High Bandwidth Memory (HBM) memory and a host controller. Olympus alleviates such bottlenecks by introducing dedicated optimizations. Also, Olympus automates several optimizations that, currently, must be manually introduced in FPGA architectures as block diagrams or code rewritings. For this reason, this tool is intended to complement (and not replace) commercial frameworks for FPGA design, like AMD Vitis. In conclusion, we can consider the hardware generation flow as an extension towards higher levels of abstraction of the current FPGA design methods.
- Inference on FPGAs: One of our main goals is lowering the barrier for the deployment of FPGAs by non-FPGA experts. DOSA has a simple command line interface to automatically compile, build and deploy a DNN on a heterogeneous, distributed cluster. Due to the roofline, bandwidth, parameter, and device-compatibility analysis of DOSA, the scheduling and partitioning is possible completely without any involvement of the user. Alternative frameworks like FINN or VitisAI require the user to program in Python,

[2] `https://www.ecmwf.int/en/elibrary/79850-ecrad-new-radiation-scheme-ifs`

C++, and HLS to adapt the accelerator to the users needs. Hence, without DOSA, a user who wants to deploy such a DNN is required to manually identify which part is supported by which framework, partition the DNN, generate the partial designs, and depending on the framework write the necessary glue logic manually. DOSA automates this completely and creates required build and deploy scripts. Furthermore, DOSA supports the very popular community standard ONNX. In summary, DOSA significantly improves the user's productivity and offers better coverage of ONNX than comparable frameworks. For further details, please refer to [28].

8 Conclusions

In this deliverable we described the final status of the compilation framework. Apart from minor adaptations, expected given the challenging goals of the EVEREST project, the framework follows the design from Deliverable D4.1. We explained how we achieved convergence of abstractions, with several key intermediate languages as described in Figure 8, which was one of the key goals of the EVEREST SDK. We believe that these abstractions, the transformations and lowerings, and the connection to system-architecture exploration can serve to build more complex tool flows in the future. Leveraging the MLIR infrastructure, users of the SDK can now connect different frontends (similar to the ONNX import for other DSLs) or implement lowerings to other target systems (e.g. to GPUs or the recent AMD AI engines).

Compared to the intermediate report in Deliverable D4.2, this deliverable describe several new additions to the SDK, including, (1) extended language support, e.g., for subscript expressions required for the radiation module of WRF, (2) an end-to-end dataflow programming with annotations for off-loading supported by a rich type system, (3) an end-to-end ML flow based on operation set architectures for distributed execution of large models across FPGAs, (4) a system integration flow that leverages MLIR-based transformations to create the overall hardware system and the corresponding driver API, (5) a clean interface via basecamp to access the tools and wrap the implementations into deployable climbs for the runtime system. Details on the commands to access the tools are provided in Deliverable D4.6. The transformations and optimizations, demonstrated in Deliverable D4.2 based on data policies identified in Deliverable D3.1, represent a solid basis for performance engineers and for extensions after project end. The role the tools play in the use cases is described in Deliverable D6.3, with evaluation results in Deliverable D6.5.

As detailed in Section 7, overall, the work described in this and related deliverables fulfills to a great extent the requirements of the project described in Deliverable D2.2 and refined Deliverable D2.5.

Acronyms

ABI Application Binary Interface. 12, 13, 49

API Application Programming Interface. 7, 37, 42, 49

AST Abstract Syntax Tree. 13, 16, 49

CFD Computational Fluid Dynamics. 9, 11, 49

CKD Correlated K-Distribution. 11, 13, 49

CLI command line interface. 7, 9, 37, 49

CU Computational Unit. 30, 31, 49

DFG Dataflow Graph. 13, 32, 33, 49

DNN Deep Neuronal Networks. 22, 37, 49

DSE Domain-Space Exploration. 16, 24, 49

DSL Domain-Specific Language. 6, 7, 9, 11–13, 15, 39, 47, 49

EKL EVEREST Kernel Language. 11, 12, 16, 49

ESN Einstein Summation Notation. 11, 16, 17, 49

FFI Foreign Function Interface. 12, 13, 49

FPGA Field Programmable Gate Array. 4, 7, 10, 13, 14, 26, 27, 30–32, 36, 37, 41, 42, 47, 49

GEMM general matrix-matrix multiply. 49

GPU graphical processing unit. 21, 23, 41, 49

HBM High Bandwidth Memory. 47, 49

HDL Hardware Description Language. 27, 28, 33, 49

HLS High-Level Synthesis. 4–7, 9, 13, 22, 26–31, 42–47, 49

HPC High-Performance Computing. 6, 7, 9, 11, 42, 49

IR Intermediate Representation. 6, 16, 27–29, 33, 49

JSON JavaScript Object Notation. 27, 49

ML Machine Learning. 14, 15, 22, 27, 42, 49

MLIR Multi-Level Intermediate Representation. 6, 7, 9, 11, 13, 15–17, 21, 22, 26–28, 32, 33, 40, 41, 47, 49

MoC Model of Compute. 19, 49

OI Operational Intensity. 49

ONNX Open Neural Network eXchange. 14, 49

OSA Operation Set Architectures. 49

PC pseudochannel. 33–35, 49

PLM Private Local Memory. 35, 49

PTQ post-training quantization. 14, 49

RRTMG Rapid Radiative Transfer Model for GCM Solvers. 49

RTL Register Transfer Level. 9, 26, 27, 29, 30, 49

SDK System Development Kit. 6, 7, 9–11, 15, 19, 36, 43, 49

SIMD Single Instruction Multiple Data. 49

SLP Superword Level Parallelism. 49

TCL Tool Command Langauge. 27, 49

UB Undefined Behavior. 49

References

[1] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, Cédric Bourrasset, François Berry, and Jocelyn Serot. Tactics to directly map cnn graphs on embedded fpgas.

[2] Anton Afanasyev, Mauro Bianco, Lukas Mosimann, Carlos Osuna, Felix Thaler, Hannes Vogt, Oliver Fuhrer, Joost VandeVondele, and Thomas C Schulthess. Gridtools: A framework for portable weather and climate applications. SoftwareX, 15:100707, 2021.

[3] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks.

[4] Nicolas Bohm Agostini, **S. Curzel**, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9, 2022.

[5] Nick Brown, Tobias Grosser, Mathieu Fehr, Michel Steuwer, and Paul Kelly. xdsl: A common compiler ecosystem for domain specific languages.

[6] PyTorch Community. TORCHSCRIPT.

[7] Serena Curzel, Sofija Jovic, Michele Fiorito, Antonino Tumeo, and Fabrizio Ferrandi. Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design. In 12th International Workshop on Polyhedral Compilation Techniques (IMPACT), pages 1–10, 2022.

[8] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. Fast inference of deep neural networks in FPGAs for particle physics. Journal of Instrumentation, 13(7), 2018.

[9] Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. Supporting fine-grained dataflow parallelism in big data systems. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), PMAM'18, pages 41–50, New York, NY, USA, February 2018. ACM.

[10] Sebastian Ertel, Christof Fetzer, and Pascal Felber. Ohua: Implicit dataflow programming for concurrent systems. In Proceedings of the Principles and Practices of Programming on The Java Platform, pages 51–64. 2015.

[11] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC), pages 1327–1330, 2021.

[12] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-Scale DNN processor for real-Time AI. Proceedings - International Symposium on Computer Architecture, pages 1–14, 2018.

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

[14] Karl F. A. Friebel, Stephanie Soldavini, Gerald Hempel, Christian Pilato, and Jeronimo Castrillon. From domain-specific languages to memory-optimized accelerators for fluid dynamics. In IEEE International Conference on Cluster Computing (CLUSTER), pages 759–766, 2021.

[15] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation. ACM Transactions on Architecture and Code Optimization (TACO), 18(4):1–23, 2021.

[16] Robert Khasanov and Jeronimo Castrillon. Energy-efficient runtime resource management for adaptable multi-application mapping. In Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE), DATE '20, pages 909–914. IEEE, March 2020.

[17] Robert Khasanov, Marc Dietrich, and Jeronimo Castrillon. Flexible spatio-temporal energy-efficient runtime management. In 29th Asia and South Pacific Design Automation Conference (ASP-DAC24), page 8pp. IEEE, January 2024.

[18] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware–software blueprint for flexible deep learning specialization. IEEE Micro, 39(5):8–16, September 2019.

[19] Stephen Neuendorffer and Lana Josipovi. handshake_dialect, 2020.

[20] Jonas Ney, Dominik Loroch, Vladimir Rybalkin, Nico Weber, Jens Krüger, and Norbert Wehn. HALF: Holistic Auto Machine Learning for FPGAs. In Proceedings of the 31st IEEE International Conference on Field-Programmable Logic and Applications (FPL), pages –. IEEE, 2021.

[21] John G Papastavridis. Tensor calculus and analytical dynamics. Routledge, 2018.

[22] Alessandro Pappalardo. Xilinx/brevitas, 2023.

[23] Enrique G Paredes, Linus Groner, Stefano Ubbiali, Hannes Vogt, Alberto Madonna, Kean Mariotti, Felipe Cruz, Lucas Benedicic, Mauro Bianco, Joost VandeVondele, et al. Gt4py: High performance stencils for weather and climate applications using python. arXiv preprint arXiv:2311.08322, 2023.

[24] Christian Pilato, Subhadeep Banik, Jakub Beránek, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Radim Cmar, Serena Curzel, Fabrizio Ferrandi, Karl F. A. Friebel, Antonella Galizia, Matteo Grasso, Paulo Silva, Jan Martinovic, Gianluca Palermo, Michele Paolino, Andrea Parodi, Antonio Parodi, Fabio Pintus, Raphael Polig, David Poulet, Francesco Regazzoni, Burkhard Ringlein, Roberto Rocco, Katerina Slaninova, Tom Slooff, Stephanie Soldavini, Felix Suchert, Mattia Tibaldi, Beat Weiss, and Christoph Hagleitner. A system development kit for big data applications on FPGA-based clusters: The EVEREST approach. In Proceedings of the 2024 Design, Automation and Test in Europe Conference (DATE), DATE'24, page 6pp, March 2024.

[25] Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, Gianluca Palermo, Michele Paolino, Antonio Parodi, Lorenzo Pittaluga, Daniel Raho, Francesco Regazzoni, Katerina Slaninova, and Christoph Hagleitner. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1320–1325, 2021.

[26] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. System-level optimization of accelerator local memory for heterogeneous systems-on-chip. IEEE Transactions on CAD of Integrated Circuits and Systems, 36(3):435–448, 2017.

[27] Robert Pincus, Eli J Mlawer, and Jennifer S Delamere. Balancing accuracy, efficiency, and flexibility in radiation calculations for dynamical models. Journal of Advances in Modeling Earth Systems, 11(10):3074–3089, 2019.

[28] B. Ringlein, F. Abel, D. Diamantopoulos, B. Weiss, C. Hagleitner, and D. Fey. Dosa: Organic compilation for neural network inference on distributed fpgas. In Proceedings of the 2023 IEEE international conference on edge computing & communications (EDGE 2023)), page 4350, Chicago, Illinois, July 2023. IEEE. Citation Key: Ringlein2023.

[29] B. Ringlein, F. Abel, D. Diamantopoulos, B. Weiss, C. Hagleitner, M. Reichenbach, and D. Fey. A case for function-as-a-service with disaggregated fpgas. In Proceedings of the 2021 IEEE 14th international conference on cloud computing (CLOUD 2021), page 333344, Virtual Conference, September 2021. IEEE. Citation Key: Ringlein2021.

[30] Burkhard Ringlein, Francois Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Advancing compilation of dnns for fpgas using operation set architectures. IEEE Computer Architecture Letters, 22(1):9–12, 2023.

[31] Burkhard Ringlein, Francois Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Advancing compilation of dnns for fpgas using operation set architectures. IEEE Computer Architecture Letters, 22(1):912, January 2023. Citation Key: Ringlein2022.

[32] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Programming reconfigurable heterogeneous computing clusters using mpi with transpilation. In 2020 IEEE/ACM international workshop on heterogeneous high-performance reconfigurable computing (H2RC), page 19. IEEE, November 2020. Citation Key: Ringlein2020a.

[33] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. ZrlmpI: A unified programming model for reconfigurable heterogeneous computing clusters. In 2020 IEEE 28th annual international symposium on field-programmable custom computing machines (FCCM), Proceedings of the 28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), page 220, Fayetteville, Arkansas, May 2020. IEEE. Citation Key: Ringlein2020 ISSN: 2576-2621 tex.eventdate: 3-6 May 2020.

[34] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Dosa: Organic compilation for neural network inference on distributed fpgas. In 2023 IEEE International Conference on Edge Computing and Communications (EDGE), pages 43–50, 2023.

[35] Norman A. Rink and Jeronimo 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), ARRAY 2019, pages 57–68, New York, NY, USA, June 2019. ACM.

[36] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. Cfdlang: High-level code generation for high-order methods in fluid dynamics. In Proceedings of the 3rd International Workshop on Real World Domain Specific Languages (RWDSL 2018), RWDSL2018, pages 5:1–5:10, New York, NY, USA, February 2018. ACM.

[37] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new IR for machine learning frameworks. In Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2018, pages 58–68, New York, NY, USA, 2018. ACM.

[38] Kentaro Sano, Atsushi Koshiba, Takaaki Miyajima, and Tomohiro Ueno. Essper: Elastic and scalable fpga-cluster system for high-performance reconfigurable computing with supercomputer fugaku. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPCAsia '23, page 140150, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Stephanie Soldavini, Karl Friebel, Mattia Tibaldi, Gerald Hempel, Jeronimo Castrillon, and Christian Pilato. Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics. ACM Transactions on Reconfigurable Technology and Systems, 16(2):1–34, 2023.

[40] Stephanie Soldavini, Donatella Sciuto, and Christian Pilato. Iris: Automatic generation of efficient data layouts for high bandwidth utilization. In Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC 2023), 2023.

[41] Felix Suchert, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel. ConDRust: Scalable deterministic concurrency from verifiable rust programs. In Karim Ali and Guido Salvaneschi, editors, 37th European

Conference on Object-Oriented Programming (ECOOP 2023), volume 263 of Leibniz International Proceedings in Informatics (LIPIcs), pages 33:1–33:39, Dagstuhl, Germany, July 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[42] Adilla Susungi, Norman A Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. Meta-programming for cross-domain tensor optimizations. ACM SIGPLAN Notices, 53(9):79–92, 2018.

[43] The Linux Foundation. Open Neural Network Exchange (ONNX), 2022.

[44] The ONNX community. Operator Schemas, 2022.

[45] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pages 65–74, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Felix Wittwer. Ohua as an stm alternative for shared state applications. Master's thesis, TU Dresden, August 2020.

[47] Xilinx Inc. Vitis AI User Documentation, 2021.

[48] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations. In Proceedings of the 59th ACM/IEEE Design Automation Conference, 2022.

[49] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator. In Proceedings of the 39th International Conference on Computer-Aided Design, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.