http://www.everest-h2020.eu

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

D4.2 — Intermediate 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	36 months
Project Website	http://www.everest-h2020.eu

Copyright

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

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

Scientific Coordinator: Christian Pilato – Politecnico di Milano

Disclaimer

Deliverable Information

Work-package	WP4
Deliverable No.	D4.2
Deliverable Title	Intermediate report of the compilation framework
Lead Beneficiary	TUD
Type of Deliverable	Report
Dissemination Level	Public
Due Date	01/10/2020

Document Information

Delivery Date	04/04/2022
No. pages	41
Version | Status	0.4 | Final
Responsible Person	Jeronimo Castrillon (TUD)
Authors	Jeronimo Castrillon (TUD), Karl Friebel (TUD), Felix Wittwer (TUD), Burkhard Ringlein (IBM), Michele Fiorito (PDM), Fabrizio Ferrandi (PDM), Donatella Sciuto (PDM), Christian Pilato (PDM), Stephanie Soldavini (PDM)
Internal Reviewer	Christoph Hagleitner (IBM)

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

Revision History

Date	Ver.	Author(s)	Summary of main changes
12.01.2022	0.1	Jeronimo Castrillon (TUD)	Initial structure
17.03.2022	0.2	Multiple contributors	Major content in place
18.03.2022	0.3	Jeronimo Castrillon (TUD)	Finish overviews, simplified table of contents, fixed consistency issues.
31.03.2022	0.4	Jeronimo Castrillon (TUD)	Clean up, reacted to comments by internal reviewer.

Quality Control

Approved by Internal Reviewer	April 4, 2022
Approved by WP Leader	April 4, 2022
Approved by Scientific Coordinator	April 4, 2022
Approved by Project Coordinator	April 4, 2022

Table of Contents

1 Executive Summary

The EVEREST project proposes a platform and system development kit to deploy demanding workflows to suitable high-performance or edge hardware [17]. This document provides an intermediate description of the compilation framework, midway through the project. 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. Details on how to use the tools are provided in Deliverable D4.3. 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, source-to-source high-level transformations and hardware-oriented transformations for computational motifs that are representative of those found in the use cases of the EVEREST project. Given the complexity of the use cases, some of which are further developed during the project, this document focuses on individual compilation flows and how they converge prior to execution on the EVEREST target platform.

Deliverable highlights

No.	Highlight	Section(s)
1	Coherent framework design, integrating multiple software components	Section 2 (Figure 1)
2	Language support for the different domains of the EVEREST project	Section 3.1
3	Initial plan for convergent intermediate representation and optimization	Section 3.2.1, Section 4.1 (Figure 4)
4	Infrastructure for transformation, optimization and variant-generation for the different domains of the EVEREST project	Section 4
5	Generalized HW generation flow with support for multiple downstream HLS-flows	Section 5 (Figure 14)
6	Seamless connection between software compilation and hardware transformations for kernel-level optimizations, memory optimizations and system integration	Section 4.1-Section 5.3
7	Initial integration with the runtime system	Section 6 (Figure 21)

1.1 Structure of this Document

This document starts with an overview of the compilation framework in Section 2. The overview lists all involved software components of the framework, which are then described in the remaining sections. In particular, the added language support to react to the use case requirements Deliverable D2.2 is described in Section 3. The section also describes the compiler-internal representations and the interfacing to the hardware generation flow. Section 4 describes transformations and optimizations, leveraging the high-level intermediate representations introduced in Section 3. Hardware programming and integration flows are described in Section 5. This includes extensions to High-Level Synthesis (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 will be integrated with the dynamic runtime environment described in Deliverable D5.1. This deliverable finishes with conclusions and next steps in Section 7.

2 Overview of the Compilation Framework

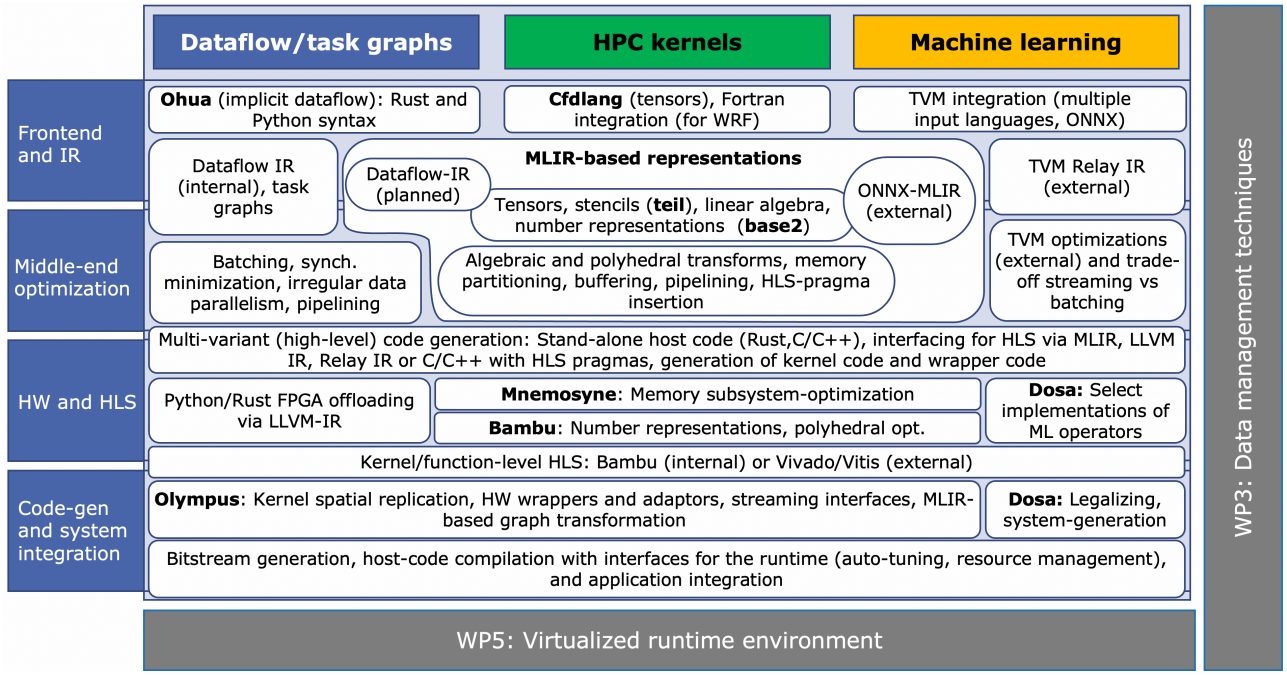


Figure 1 – Overview of the compilation flow.

Following from Figure 1 in Deliverable D4.1, a more detailed representation of the languages, components, tools, interfaces and methods of the compilation framework is shown in Figure 1. From the high-level point of view, the figure reflects the heterogeneity of the requirements identified early in the project. This is reflected on the multiple types of applications (dataflow/task graphs, HPC kernels and machine learning), the different languages (Fortran, C/C++, ONNX, Rust, Python and Domain-Specific Languages (DSLs)), the rich set of interfaces (via intermediate code in LLVM or MLIR, via generated C/C++ or Rust code and other metadata for variant generation), and the integration with multiple external tools and frameworks (WRF build system, TVM, the MLIR infrastructure, Vivado/Vitis HLS tools. As described in the proposal, one the goals of the EVEREST SDK and its compilation framework is to work on *unification* of methods, so as contribute to the convergence of data-centric applications involving HPC, Big Data and machine learning. At the intermediate level, Figure 1 shows unification around the MLIR framework. This is a considerable advancement with respect to the decoupled flows planed for simplicity in Deliverable D4.1. As the programming flows advance towards the HW of the EVEREST platform, the flows converge ending at the final bitstream and code generation step. For generality, the HW generation flow supports both vendor-tools (e.g., Xilinx Vivado/Vitis) as well as open-source alternatives (Bambu).

Figure 1 also shows that the data management techniques from WP3 inspire the implementations in the methods and tools developed within WP4. The code produced by the compiler can run standalone on the different nodes of the EVEREST platform, with only CPUs, or with bus or network-attached Field Programmable Gate Arrays (FPGAs). For final deployment, the code adheres to the Application Programming Interfaces (APIs) of the EVEREST runtime environment of WP5.

From top-to-bottom, the four major phases of Figure 1 are described in this deliverable. The language support (front end) and the internal compiler representations are described 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; the basic Cfdlang DSL for tensor kernels in HPC, like in Computational Fluid Dynamics (CFD) applications in Deliverable D2.1; 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 support primarily the HPC kernels. However, as shown in the figure, we plan to provide a dataflow IR in MLIR, equivalent to the internal one used in the Ohua compiler. That way and with the ONNX-MLIR import a unified representation for all types of application classes can be readily obtained.

In the compiler middle end (optimization in Figure 1) the EVEREST compilation framework integrates a series of analysis and optimizations for the different application cases. 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. Bambu's polyhedral analysis seamlessly interoperates with the representations used in the middle end. Custom number representations supported by the language for HPC kernels are compatible with those supported for Bambu for trade-off analysis (area, performance and accuracy). Information about lifetime of variables, data formats and data allocation (cf. Deliverable D3.1) can also be exported for custom memory subsystem generation using Mnemosyne. Mnemosyne then creates private local memories and the logic to bring the data to and from the right accelerator. 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) implementaitons 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. At this level, Olympus performs several optimizations to effectively use the area in the FPGA and balance the computation and communication time (specially relevant for systems with multiple memory channels). This phase can profit from a further MLIR integration in the second half of the project. In this way, operator scheduling and graph pipelining information from the upper compilation stages can be passed to the system integration.

3 Language Support and Intermediate Representations

This section discusses the language support (Section 3.1) and intermediate representations (Section 3.2 of the compiler framework. As specified in the project plan and Deliverable D4.1, we provide support for kernels in HPC and machine learning as well as for tasks and dataflow graphs. We had originally planned to support particle-based simulations in the compilation framework, but an explicit need was not identified in the requirement analysis (cf. Deliverable D2.2).

3.1 Language Support

3.1.1 CFDlang Extensions

Our starting point for the EVEREST kernel DSL was the CFDlang language [20], originally designed for CFD applications with primitives extended to cross-domain tensor expressions in [24]. It adopts an operator-based syntax for tensor expressions, as opposed to index-based expressions in languages such as TensorComprehensions [31]. This provides the freedom needed to implement data management techniques as described in Deliverable D3.1. Various extensions have been made to the implementation of this language in order to bring the EVEREST technology to the Inverse Helmholtz kernel use case. See Figure 2 for our driving inverse Helmholtz operator example in current DSL syntax.

At present, most extensions to CFDlang are related to the compiler implementation, and will be described in Section 3.2. This allows us to make major adjustments to the language itself very quickly, which was one of the prerequisites to achieve the use-case provider-centric design evolution targeted in Deliverable D2.2. We are currently at a point in language design where the application requirements have been identified and concrete implementations have been planned, but there is no coherent solution for all use cases yet. In this prototype phase, we have identified a few inconsistencies in the language and began to improve the integration into the EVEREST flow. For example, this includes changes to the way declarations are made, so that the DSL reflects the user's view of the application better, and forms a closer link to the EVEREST runtime (discussed later in Section 6).

```
var input S   : [11 11]
var input D   : [11 11 11]
var input u   : [11 11 11]
var output v  : [11 11 11]
var t         : [11 11 11]
var r         : [11 11 11]

t = S#S#S#u . [[1 6][3 7][5 8]]
r = D * t
v = S#S#S#r . [[0 6][2 7][4 8]]
```

Figure 2 – The inverse Helmholtz operator kernel in CFDlang DSL.

Future extensions will be necessary to achieve the targets set for the DSL in Deliverable D2.2 in a transparent and transferable fashion. Our current changes to the EVEREST kernel DSL and it's design include:

- Index-based expressions: While the absence of index-based expressions makes CFDlang free of Undefined Behavior (UB), we will relax this constraint to provide better support for use cases. In particular, a restricted introduction of implicitly ranged indices will allow mapping a larger subset of linear algebra, without introducing UB, while achieving a more readable syntax at the same time.
- Stencil application: Stencils are an important part of many numerical physics simulations, and are often prime candidates for the kind of optimization we pursue. With our interoperable pipeline based on MLIR, we now include language features that we can delegated to third-party components. With the OpenEarthCompiler [13], stencils are a logical first step in that direction, as they can also be of use to the exemplary EVEREST applications.
- Annotations: Although a maximum degree of automation is desired, it is still prudent to allow for external

guidance through DSL annotations. As a first step, this gives us the tools to test and design more of our internal annotations that are used in the lowering process. For example, annotations that clarify intent and extra functional requirements, such as quantization information, give useful insight to the compiler that only the domain expert could provide.

3.1.2 Fortran Integration for WRF

One part of unifying the view the runtime and the user have on the application through the DSL is the method with which it is integrated into the existing codebase. In case of WRF, we have a large legacy codebase which makes a DSL integration particularly challenging. These challenges affect the EVEREST kernel DSL itself, its runtime library, and the "host application" (i.e. WRF in this case).

Our efforts have been three-fold in this regard:

- Fortran-style DSL syntax extensions: We rely on domain experts for the extraction of kernels from the host application. For domains such as weather modeling, this is a sensible approach, since they have lots of experience running simulations. In practice, this does not just mean identifying kernels, but also refactoring the legacy codebase to extract kernel invocations to function calls, so that we may provide an Foreign Function Interface (FFI) interface. In the case of WRF, this is relatively easy because of its inherent disaggregation of the individual driver components, but still requires the user to reason about the Fortran interface with all its peculiarities. By offering a new, optional declaration syntax, we accept a subset of Fortran that is relevant for FFI as shown below. As a result, the compiler can automate the error-prone process of matching the data layouts and transcribing the interfaces.

```
REAL                          :: S(11,11)
REAL, DIMENSION(11,11,11)     :: D,u,v,t,r
```

- C++-Fortran interop runtime library: The unification of the EVEREST flows has highlighted the need for common interface abstraction. Towards such a unification, we started developing a common utility library using the `memref` abstraction of the MLIR as the most general form of argument passing between components.

  A C++-Fortran interop library is also added to the EVEREST kernel DSL runtime (see Interop Layer in Section 6). In the short term, this library assists us in porting the Rapid Radiative Transfer Model for GCM Solvers (RRTMG) driver from its original source, especially when it comes to its integrating testing suite.

- Pre-processing facilities: As a final method of reducing the barrier between DSL and host application, we added pre-processing facilities to our flow. With this scheme, we enable a mixed-source approach for applications like WRF, where the EVEREST kernel DSL can be merged with the original codebase. With regards to long term motivations for projects like ours, we consider it important to enable the application developers (in this context not the users but developers of WRF) to ship a compatible upgrade, even if our DSL can not yet reach that level of interchangeability.

3.1.3 Dataflow

As basis for a language that can express dataflow, we use the Ohua compiler framework [8, 6]. 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 [33]. The new version of the framework, instead, uses a subset of the host language as input. 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 extensions of the compiler to support new languages as 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. Ohua's middle-end is target-agnostic, which means that changes to the core logic of the compiler are usually not necessary to support a new language. Currently, front- and backends are implemented for both Rust and Python, albeit support for the latter is only experimental at the moment. In the following, we will focus on the

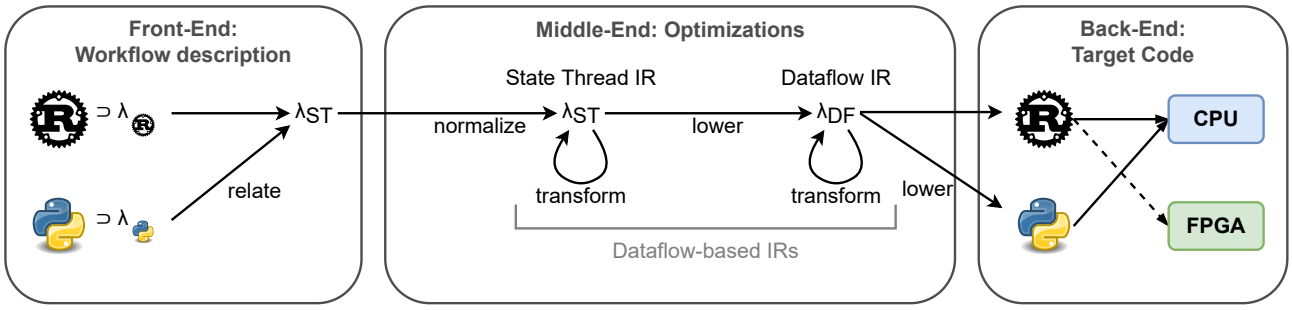


Figure 3 – Current status of the Ohua compilation flow.

Rust integration of our compiler, which is of higher relevance for EVEREST use cases. Figure 3 shows the current flow of the Ohua compiler. The end-to-end flow is explained in more detail in Section 3.2.3.

At its core, an Ohua algorithm (implemented in any supported language subset) is merely a composition of calls to both stateless and stateful functions in a separate file. The definition of such functions is not part of the programming model, i.e., they are not part of Ohua's compilation process. This allows for reuse of existing functions that implement the actual functionality of a program as well as operations on data structures, often referred to as methods. Developers provide a file of valid Rust code to the compiler, which will then in turn produce a file describing the same algorithm in parallel Rust.

An implementation of the *Probabilistic Time Dependant Routing* algorithm from the Traffic Simulation Use Case, would then look like this:

```rust
pub fn delay_profile(
    route: Arc<Route<String>>,
    departure_time: DateTime<Utc>,
    prob_profile: Arc<SegmentsHistoryProbProfile<String, Quartiles>>,
    samples: usize,
) -> Vec<Duration> {
    let no_limit = Arc::new(NoLimitProbProfile::new());
    let free_flow_duration = drive(route.clone(), departure_time, no_limit);

    let mut res = Vec::new();
    for _ in helpers::sample_range(samples) {
        let duration = drive(route.clone(), departure_time, prob_profile.clone());
        let delta = duration - free_flow_duration;

        res.push(delta);
    }

    res
}
```

Deliverable D4.3 details how we arrive at this snippet by transforming the original code base.

In the middle-end, the Ohua compiler operates on a Dataflow Graph (DFG). This directly results in two fundamental restrictions on the input code:

- *Using references is prohibited:* In an Ohua program, each function call in the algorithm definition is regarded a single node in the DFG. After applying transformations to the graph, as outlined in Section 4.2, the resulting graph is serialized into Rust code. The parallel Rust code encapsulates each node of the graph by spawning it as a separate thread, realizing communication between these operators by using unbounded FIFO queues. This enables pipeline parallelism in the graph. But at the same time, this also means that using shared references becomes impossible. A simple solution to this problem is the introduction of reference counted pointers which allow the sharing of data across nodes at a negligible cost. We are looking into methods of automatically wrapping data structures in such pointers where

necessary, e.g., by enhancing the type checker along the lines of stacked borrows [14].

- *Stricter move semantics:* In a typical Rust program, primitive types and also some slightly more complex ones implement the `Copy` trait. This trait relaxes the strict ownership semantics of Rust by ad-hoc *copying* data on the fly when it is used several times in an owned fashion. While purely a quality-of-life improvement, copy semantics help to reduce bloat in the code by repeatedly having to *clone* data or using references as layer of indirection. Ohua is currently unable to support this feature because values generated during the execution of the algorithm are only visible in the scope of the dataflow node. Therefore, copying the result to multiple other nodes is impossible without a deeper understanding of the target language's type system, which is beyond the scope of our compiler implementation. As a result, developers need to resort to manually cloning any value produced during execution of the algorithm. Arguments to the algorithm are not affected by this and still support copy semantics, as the value can simply be copied into the different dataflow node scopes.

Other limitations such as the absence of support for `if` clauses mainly stem from the prototype nature of the compiler.

A key motivation behind using a dataflow-driven DSL in the EVEREST project is that a DFG abstracts over the individual computations that form the algorithm. This comes in handy when deploying such a program onto heterogeneous architectures. The abstraction will allow for a tight integraton for off-loading single nodes of the DFG to FPGAs using HLS, as outlined in Section 5.1.Since the off-loaded functions themselves are not part of the compilation process of Ohua, the main difference between offloaded functions and normal functions is the communication with the nodes. While normal nodes of the DFG communicate with one another using Rust's standard FIFO queues, communication with an off-loaded node requires actual data transfers from and to the accelerator. To notify the Ohua Compiler (`ohuac`) of this change, functions that will be deployed onto the FPGA will be annotated with a macro:

```
#[kernel]
let result = offloaded_computation(/* args */);
```

3.1.4 Machine Learning

ML is a very active field of research, which gave birth to plenty of tools, standards, and frameworks aiming to support users and researchers. The EVEREST project decided to use the community-standard Open Neural Network eXchange (ONNX) as input language, which will be introduced briefly in the following.

The ONNX project [27] aims to enhance the interoperability of ML tools by developing a standardised format to export and import neuronal networks. ONNX is actively developed by a large community, managed by the Linux Foundation. ONNX defines a dataflow computation graph [29], together with built-in operators [28] and data types. Each node in this acyclic graph can have multiple in- and outputs and represents a call to an operator. ONNX is extensible, so that custom operators and data types can be used. The ONNX graph can be exported to `.onnx` files, which are compressed using Protocol Buffers [11]. Thanks to efforts on the ONNX-MLIR integration by the open source community (cf. Figure 1), supporting ONNX contributes to the convergence efforts of the EVEREST project.

An example of a decompressed graph containing a single 2D convolution with activation and pooling layers represented in ONNX is shown in Listing 1. The shown example is developed and trained in the tool Pytorch and exported via ONNX. Similarly, a user of the EVEREST ONNX flow would develop and train a neuronal network in her/his preferred tool and export it into the interoperable community standard ONNX.

3.2 Intermediate Representations

Our previous definition of the compilation framework had not mandated the use of a particular Intermediate Representation (IR), aside from the interface to Bambu. Still, we expressed our desire for a unified IR that, similar to embedded DSLs, combines both host and device code in a single place. With this alpha release, we

```
graph {
  node {
    input: "input.1"
    input: "conv1.weight"
    input: "conv1.bias"
    output: "5"
    name: "Conv_0"
    op_type: "Conv"
    attribute {
      name: "dilations"
      ints: 1
      ints: 1
      type: INTS
    }
    attribute {
      name: "group"
      i: 1
      type: INT
    }
    attribute {
      name: "kernel_shape"
      ints: 5
      ints: 5
      type: INTS
    }
    attribute {
      name: "pads"
      ints: 0
      ints: 0
      ints: 0
      ints: 0
      type: INTS
    }
    attribute {
      name: "strides"
      ints: 1
      ints: 1
      type: INTS
    }
  }
  node {
    input: "5"
    output: "6"
    name: "Relu_1"
    op_type: "Relu"
  }
  node {
    input: "6"
    output: "7"
    name: "MaxPool_2"
    op_type: "MaxPool"
    attribute {
      name: "kernel_shape"
      ints: 4
      ints: 4
      type: INTS
    }
    attribute {
      name: "pads"
      ints: 0
      ints: 0
      ints: 0
      ints: 0
      type: INTS
    }
    attribute {
      name: "strides"
      ints: 4
      ints: 4
      type: INTS
    }
  }
  node {
    input: "7"
    output: "8"
    name: "Flatten_3"
    op_type: "Flatten"
    attribute {
      name: "axis"
      i: 1
      type: INT
    }
  }
  node {
    input: "8"
    input: "fc1.weight"
    input: "fc1.bias"
    output: "9"
    name: "Gemm_4"
    op_type: "Gemm"
    attribute {
      name: "alpha"
      f: 1.0
      type: FLOAT
    }
    attribute {
      name: "beta"
      f: 1.0
      type: FLOAT
    }
    attribute {
      name: "transB"
      i: 1
      type: INT
    }
  }
  name: "torch-jit-export"
  initializer {
    dims: 36
    dims: 3
    dims: 5
    dims: 5
    data_type: 1
    name: "conv1.weight"
    raw_data: "..."
    }
  initializer {
    dims: 36
    data_type: 1
    name: "conv1.bias"
    raw_data: "..."
  }
  initializer {
    dims: 10
    dims: 1764
    data_type: 1
    name: "fc1.weight"
    raw_data: "..."
  }
  initializer {
    dims: 10
    data_type: 1
    name: "fc1.bias"
    raw_data: "..."
  }
  input {
    name: "input.1"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 1
          }
          dim {
            dim_value: 3
          }
          dim {
            dim_value: 32
          }
          dim {
            dim_value: 32
          }
        }
      }
    }
  }
  output {
    name: "8"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 1
          }
          dim {
            dim_value: 1764
          }
        }
      }
    }
  }
}
```

Listing 1 – A 5x5 Convolution with 3 in channels and 36 out channels followed by a Relu activation, a 2x2 max pooling, a batch flatten and one dense operation. The raw data of the weights and bias are omitted.

have begun embracing the MLIR as the single representation from front-end to system integration, wherever possible. We demonstrate the advantages of this approach in light of arbitrary precision floating-point computations and our now closely integrated Bambu flow. A unified IR promises interdependent optimization of host and device code, which is key to our planned system integration flow, carrying over our previous achievements.

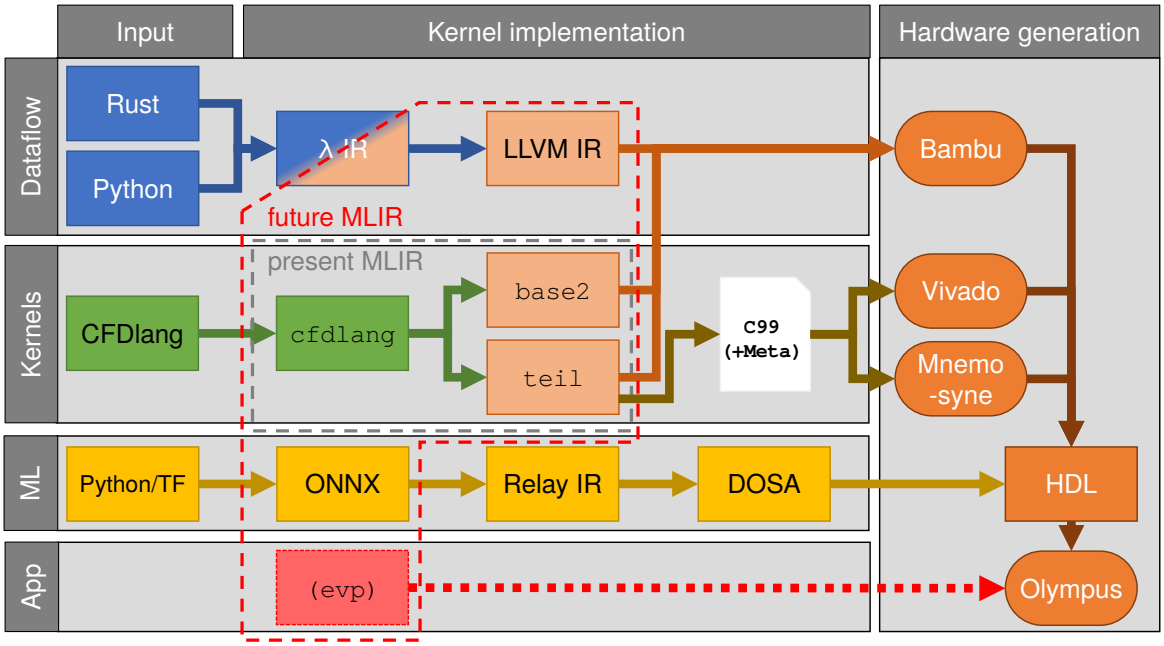


Figure 4 – Overview of current and planned IR collaboration.

Figure 4 shows a collaboration diagram of the different IRs currently used, and the currently planned changes. Depicted is a subview of Figure 1, which starts from the input languages and ends at the system integration level. The three application flows are separated by their front-ends, which we expect to be able to merge with little effort, using existing MLIR infrastructure and our extensions to it. While the contrast in the application domains makes interoperation between middle-end components not very likely, we are still looking to share an MLIR representation here in multiple places. This is especially relevant for the planned extensions to the way Olympus will integrate with applications, for which we imagine a unified application-embedded support runtime with a custom MLIR dialect (indicated by `evp` in Figure 4).

Components which are currently precluded from a fully integrated MLIR infrastructure are our machine learning and dataflow-centric flows. However, we do not view this as problematic, since interoperability can be achieved with currently available third-party components. In addition, we expect future developments in the field of MLIR front-ends to widen the range of supported programs and languages considerably.

3.2.1 Design Rationale

Extending MLIR for a new kind of language, application domain or target device is made easy through its decoupled dialect infrastructure. We are able to develop the DSLs, the runtime and the device drivers mostly independently, and represent them as independent parts within the same IR. A unified flow through MLIR is also eased by its ability to interact with third-party components, which can provide further domain-centric optimizations or languages.

The key aspects of this design are:

- Front-end integration: By connecting the DSL front-end directly within MLIR, we greatly speed up the iterative language design process we are facing. Additionally, we provide tangible benefits to the user of our EVEREST SDK. Through the use of MLIR's facilities and location tracking, we can provide accurate debug information and relate implementation obstacles back to DSL code. Not only does the domain expert receive more valuable error information and performance hints, our dialect design also allows a backward flow that can visualize changes to them as DSL code.

- Separated domain-centric optimizations: To enable truly cross-domain capable optimizations, we must clearly separate them from the front-end tasks related to host code embedding, and make them entirely independent of target-specific transforms. MLIR provides a growing ecosystem of dialects that are designed for this purpose, and we extend it with new ones that fit into this existing hierarchy. We want to give future users the power to plug their own components in these places, which is made possible by MLIR.
- Abstract interfaces: A dependency from an abstract to a concrete dialect is often the easiest way to introduce a new level into the hierarchy, but violates the single responsibility principle on a higher level. We adopt a design where the higher level dialects only provide abstract definitions of e.g. number formats, memory transfers and accelerator resources. This allows us to freely add implementing dialects, such as new target devices, and gives them the highest degree of freedom possible.

We are still using components that are not native to MLIR, which means that interchange formats entering and leaving it are required (cf. Figure 1). For example, we projected that a C99-based transpiler would need to be employed for preparing HLS code. In the unified MLIR flow, we are providing a back-end translation that emits such C99 code, allowing us to interface with vendor tools. Similarly, ONNX can be round-tripped through MLIR using third-party projects, which allows us to plumb our machine learning flow more quickly to a unified front-end. MLIR is also capable of exporting to LLVM-IR, which can be reused by a variety of external tools with outstanding availability and capabilities.

Our new advancements in interfacing with Bambu have shown that MLIR can ultimately substitute all this, with added benefits on top. Bambu accepts LLVM-IR, the process of obtaining which is destructive to metadata we attach to MLIR, cutting Bambu off from any domain knowledge much like in the C99 case. However, Bambu provides its own MLIR input path, which can perform elaborate transformations on the information available there. We are now designing dialects to act as vendor interfaces to such functionality, which has already enabled arbitrary precision floating-point numbers for our numerical kernel flow.

3.2.2 MLIR Language Stack for Numerical Computations

Our most tightly MLIR-integrated flow yet is the one for numerical kernels, which currently uses the CFDlang DSL as input. Following the guidelines from Section 3.2.1, we have implemented a set of dialects that implement the desired separation. Figure 5 shows the dialect hierarchy from abstract on the left to concrete on the right (following from the specification in Figure 8 of Deliverable D4.1). In this flow, `cfdlang` takes the role of the front-end dialect, which subsumed all the previous diverging AST and expression tree IRs. The `teil` dialect is our proposal for a cross-domain tensor expression optimization framework, and is implemented more conventionally with a direct dependency on the existing `linalg` dialect. It provides an abstract number representation interface, which allows an implementation dialect, such as our own `base2` dialect, to implement tensor scalar types.

The compilation flow is depicted in Figure 6, which shows when these dialects are used. The `teil` dialect exists transiently as a means to perform our domain-centric optimizations, such as tensor factorization. Using the `base2` dialect, the scalar type is then implemented, which continues to coexist with the rest of the program until it is passed to synthesis. After scalar type implementation, we enter the target-specific part of this flow, where `teil` can also be used effectively, but further steps are only downward in the dialect hierarchy. This proceeds until a synthesizable artifact is obtained, at which point MLIR is either exited towards a vendor tool, or passed to the Bambu input pipeline.

One advantage of this flow is that it directly enables us to target CPUs, and also puts GPUs in range of future extensions. As an example, Figure 7 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 a multitude of different CPU architectures.

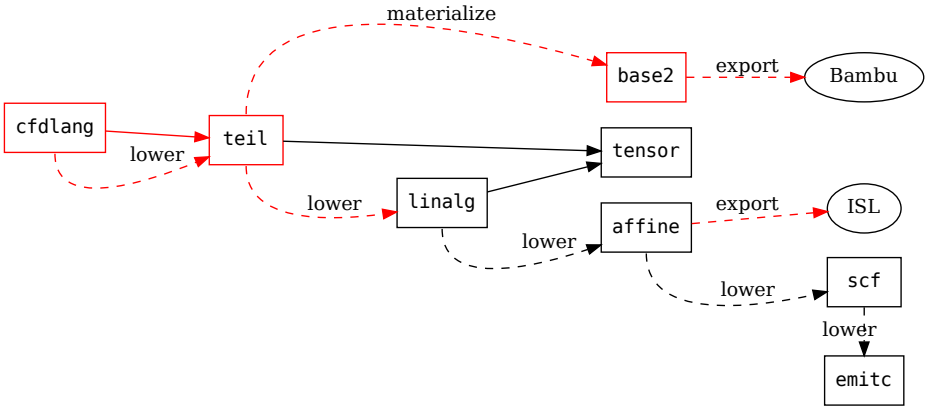


Figure 5 – MLIR dialect dependencies for numerical computations. EVEREST additions marked in red.

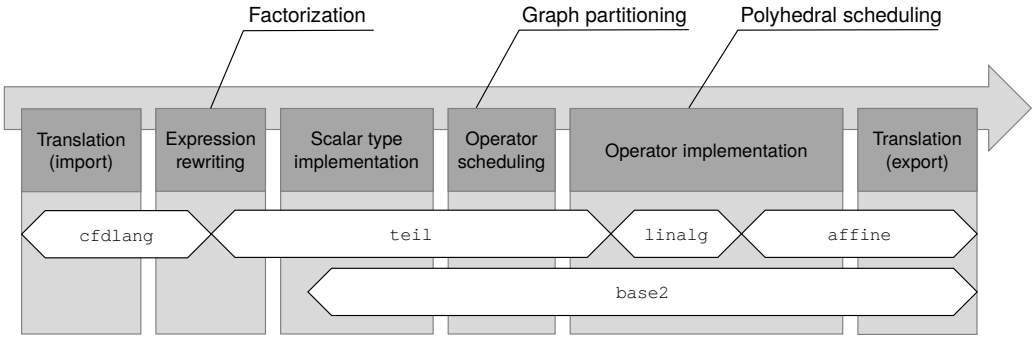


Figure 6 – Dialects in the MLIR flow for numerical kernels.

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

3.2.3 Dataflow

The compilation process for heterogeneous dataflow applications involves two different types of intermediate representations. One is the internal representation of Ohua, which is used to reason about the dataflow within a program. The other IR is used to communicate with the kernel compilation phase.

3.2.3.1 Internal Dataflow Representation

The programming model promoted by Ohua with its restrictions on variable usage enable the compiler to easily translate an algorithm into a dataflow graph. This representation exposes pipeline and task-level parallelism while preserving the algorithm semantics. And while the actual internal representation in the compiler consists of several stages exposing different key aspects of the graph more clearly, as explained in Deliverable D4.1, this generalized view suffices to explain the core concepts of the compiler.

Ohua first translates the sequential input algorithm into applicative normal form and afterwards transforms it from an imperative into a functional form. This transformation relies on the notion of state threads which we already explored earlier [7]. Every call to a stateful function there becomes a state thread which accepts a piece of state as input, mutates it, and yields the modified state again. Similarly, a call to a stateful function within a loop turns the whole loop into a state thread.

From this functional description, Ohua translates stateful and stateless function calls into nodes of a dataflow graph. Data dependencies between nodes are transformed into arcs that transfer data values in FIFO order. The different types of nodes in our dataflow graph are denoted as follows:

$$n \ ::= \boxed{f_{\mathsf{SL}}} \mid \boxed{f_{\mathsf{SF}}} \mid \boxed{\texttt{for}} \mid \boxed{\texttt{reuse}} \mid \boxed{trfix}$$

The first two node types execute calls to stateless and stateful functions respectively. In order to perform a call, a node needs to retrieve a data value from each of its incoming arcs and emits the result of the call to its outgoing arc before the next call is constructed. Stateful nodes additionally emit their updated state via a dedicated outgoing arc. Ohua translates loops and tail recursions directly into dedicated dataflow nodes. The `for` node streams the elements of the vector into its outgoing arc. The `trfix` node models the concept of tail recursion in the graph. Both language constructs, loops and tail recursion, open a new contextual scope, i.e., a subgraph. For tail recursion, this subgraph is closed such that the only way for data to enter and leave the graph is the `trfix` node. For loops, data enters the subgraph via `for` and `reuse` nodes and leaves it via a stateful function call node. Data that enters the loop subgraph via the `for` node "drives" the computation. `reuse` nodes gate the arcs that receive the data entering the loop. It attaches a reuse count n where n is the number of loop iterations, i.e., elements in the looped vector. Function call nodes that receive such a data value reuse it over the course of n calls.

This internal representation is generic, which means that it is shared by all language integrations. However, the back-end of the compiler may require certain information from the original input code, such as data type annotations in the case of the Rust integration. To facilitate this, the intermediate representation of the middle-end is designed to encapsulate such information and allow its transfer into back-end code generation.

3.2.3.2 IR for Kernel Compilation

The use cases that we have already explored with Ohua in the EVEREST project are written in Rust. However, the kernel compilation flow invoked via Bambu is unable to work with this language. Therefore, we are using LLVM IR as interfacing layer between both tools, as the Rust compiler builds on top of LLVM and is therefore natively able to emit this IR. With this glue layer in place, `ohuac` does not have to touch the code to be off-loaded, as it is directly fed to `rustc`. Nonetheless, the function in question needs to be annotated with the `#[no_mangle]` macro to allow Bambu to actually find this specific piece of code. Additionally, a few restrictions are placed on the LLVM IR to be generated by `rustc`:

- *No panic unwinding* – When a Rust program encounters an unrecoverable error, a `panic` is triggered, which normally results in the executing thread being halted and the stack being unwound, calling destructors for all data types. This behavior needs to be disabled. Instead, a program must abort on a panic.
- *No overflow checks* – `rustc` adds a number of overflow checks to the code to provide more safety at runtime. This functionality is not required when off-loading the code to an accelerator.
- *No loop vectorization and Superword Level Parallelism (SLP)* – Since Bambu will transform the input code itself, it makes no sense to run performance optimizations like vectorizations beforehand.

As mentioned before, future work will be aimed towards moving to a fully integrated MLIR infrastructure where this flow may also emit kernel specification code in this unified IR.

3.2.4 Machine Learning

The EVEREST project decided to use another machine-learning community framework, TVM [26, 4], for high-level optimizations and to reuse it's IR, RelayIR [21]. RelayIR is a functional intermediate representation for ML tasks, developed by the TVM community and based on HalideIR [19] and we decided to use RelayIR within EVEREST due to several considerations: First, TVM has an active community around Deep Neuronal Networks (DNN), it is actively developed, and has connections to the FPGAs community [16]. Second, a being a functional language, Relay is well suited to be hardware and platform agnostic. Third, TVM contains already a lot optimization passes for Relay IR and finally, Relay IR is able to also represent the training phase of DNNs. For further discussions around the usage of TVM within EVEREST, we refer the reader to Section 3.3 of Deliverable D4.1.

After importing the network description via an ONNX file, the network is converted into a RelayIR module and subsequently optimized using built-in optimization passes, such as constant folding or operator fusion. The optimized RelayIR module of the ONNX representation in Listing 1 is shown in Listing 2.

```
def @main(%input.1: Tensor[(1, 3, 32, 32), float32]) -> Tensor[(1, 10), float32] {
  %3 = fn (%p03: Tensor[(1, 3, 32, 32), float32], %p11: Tensor[(36, 3, 5, 5), float32], %p21: Tensor[(36), float32], Primitive=1) -> Tensor[(1, 36, 28, 28), ↩
        float32] {
    %1 = nn.conv2d(%p03, %p11, Tensor[(1, 3, 32, 32), float32], Tensor[(36, 3, 5, 5), float32], padding=[0, 0, 0, 0], channels=36, kernel_size=[5, 5]) /* ty=↩
          Tensor[(1, 36, 28, 28), float32] */;
    %2 = nn.bias_add(%1, %p21, Tensor[(1, 36, 28, 28), float32], Tensor[(36), float32]) /* ty=Tensor[(1, 36, 28, 28), float32] */;
    nn.relu(%2, Tensor[(1, 36, 28, 28), float32]) /* ty=Tensor[(1, 36, 28, 28), float32] */
  };
  %4 = %3(%input.1, meta[relay.Constant][0] /* ty=Tensor[(36, 3, 5, 5), float32] */, meta[relay.Constant][1] /* ty=Tensor[(36), float32] */) /* ty=Tensor[(1, ↩
        36, 28, 28), float32] */;
  %5 = fn (%p02: Tensor[(1, 36, 28, 28), float32], Primitive=1) -> Tensor[(1, 36, 7, 7), float32] {
    nn.max_pool2d(%p02, Tensor[(1, 36, 28, 28), float32], pool_size=[4, 4], strides=[4, 4], padding=[0, 0, 0, 0]) /* ty=Tensor[(1, 36, 7, 7), float32] */
  };
  %6 = %5(%4) /* ty=Tensor[(1, 36, 7, 7), float32] */;
  %7 = fn (%p01: Tensor[(1, 36, 7, 7), float32], Primitive=1) -> Tensor[(1, 1764), float32] {
    nn.batch_flatten(%p01, Tensor[(1, 36, 7, 7), float32]) /* ty=Tensor[(1, 1764), float32] */
  };
  %8 = %7(%6) /* ty=Tensor[(1, 1764), float32] */;
  %9 = fn (%p0: Tensor[(1, 1764), float32], %p1: Tensor[(10, 1764), float32], %p2: Tensor[(10), float32], Primitive=1) -> Tensor[(1, 10), float32] {
    %0 = nn.dense(%p0, %p1, Tensor[(1, 1764), float32], Tensor[(10, 1764), float32], units=10) /* ty=Tensor[(1, 10), float32] */;
    add(%0, %p2, Tensor[(1, 10), float32], Tensor[(10), float32]) /* ty=Tensor[(1, 10), float32] */
  };
  %9(%8, meta[relay.Constant][2] /* ty=Tensor[(10, 1764), float32] */, meta[relay.Constant][3] /* ty=Tensor[(10), float32] */) /* ty=Tensor[(1, 10), float32] ↩
        */
}
```

Listing 2 – Optimized RelayIR with type annotations of Convolution example shown in Listing 1.

The RelayIR module is extended by additional annotations in order to be useful for FPGA architecture generation within the EVEREST ML tool chain, called `Dosa`. The first important annotation consists of performance characteristics for each (fused) operation. Using the type annotations of RelayIR a custom Relay pass calculates the Operational Intensity (OI) of each operation. The OI is calculated in two versions: One is calculated taking the parameters and the input data into account, the second is calculated with just the input data. These two different OIs are used for architectural decisions later. The OIs for function `%3` of the example in Listing 2 would be:

- OI of function `%3` for data+parameters: 181.02
- OI of function `%3` for just data: 342.24

The second important type of annotations are implementation options. After optimizations of the AST and the performance annotations are done, each operation gets annotated with a list of it's implementation possibilities. This implementation options refer to HLS or Hardware Description Language (HDL) libraries that are re-used by Dosa and could implement this particular operation. Further details for this approach are explained in Section 4.3.

4 High-Level Transformations and DSE

This section describes how the compiler framework leverages the IRs described in Section 3.2. The section starts by describing the optimizations that the framework includes at the kernel level (Section 4.1), including a prototype of the multi-variant generation flow. The dataflow transformations for performance and to enable seamless offloading to accelerators are described in Section 4.2. Section 4.3 describes the transformations applied to ML workloads. This section closes with general remarks on the code generation flow after the middle end in Section 6.

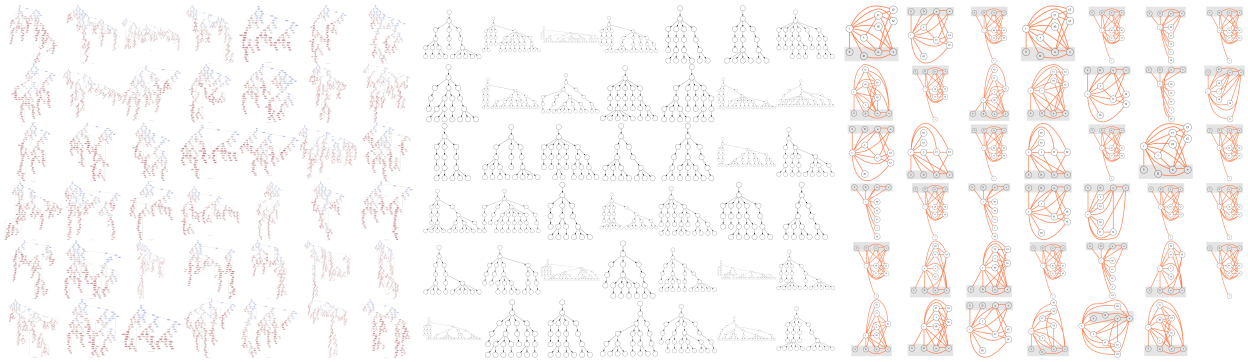
4.1 Kernel Transformations and Optimizations

Using the MLIR infrastructure, we have implemented / have access to a set of kernel transformations applicable to our higher abstraction levels.

- Expression canonicalization: MLIR supports a canonicalization mechanism that repeatedly applies a set of rewrite rules until a fixpoint is reached. In our abstract `cfdlang` and `teil` dialects, we use this extensively to simplify the kernel programs. As a result of using abstract number types, this allows us to reason about expression equality for all our DSL's statements.
- Tensor expression rewriting: The `teil` dialect was designed with implicit tensor elements in mind such that algebraic identities could be exploited easily. Our tensor expression rewrite patterns make use of this, reducing the time complexity by exploiting, e.g., associativity and distributivity. Aside from algebraic transformations, such as factorization of tensor contractions, we support platform-specific implementation choices. In an approach similar to [22], we can use tensor rewrites to establish different forms that map to specific hardware, such as systolic arrays or Single Instruction Multiple Data (SIMD) execution units. Currently, we can rewrite contractions into TTGT [15] for use with accelerators that offer efficient general matrix-matrix multiply (GEMM) implementations.
- Operator scheduling: For efficient implementation on FPGA devices, careful consideration of the memory architecture is needed. A first step is to optimize streams for the available throughput of the system memory banks, which is especially important for our High Bandwidth Memory (HBM) targets. In MLIR, we can establish the required pipeline structure on a high abstraction level before proceeding to sub-kernel implementation.
- Scalar type implementation: Since `cfdlang` and `teil` use abstract scalar types, we defer the concrete implementation to target-aware lowerings. We plan on adding quantization hints to `cfdlang`, which will be used in `teil` to guide the placement of precision boundaries. Currently, these are placed explicitly, and are then lowered onto synthesizable target types using the `base2` dialect.
- Bufferization: Another part of memory optimization is the size, layout and lifetime of buffers for intermediary results. We currently rely on MLIR's built-in reasoning, extended by our previous efforts to reduce buffer usage through after-scheduling sharing. We plan to merge these steps into one, potentially inside MLIR, allowing more back-end flows to make use of these optimizations.

Outside of MLIR, we support a Mnemosyne-enabled flow (cf. Section 5.2) powered by in-MLIR kernel analyses exported from our compiler. In addition, we use our new MLIR infrastructure to facilitate MLIR-to-DSL conversions, which provide interactive feedback to the domain expert. An example of this was also featured in our 4th webinar, and is also included in Deliverable D4.3.

Our current method for guided HLS still relies on low-level abstraction IRs, 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



(a) As control flow graphs. (b) As loop structure graph. (c) As memory sharing graph.

Figure 8 – Different implementation variants of the kernel in Figure 2. Each cell in a matrix is a variant, with cells at the same location corresponding to the same variant.

format in favor of passing this information directly in MLIR. We are working towards further improving these capabilities.

We perform DSE at two different abstraction levels. The first is the selection of alternative formulations in expression rewriting, such as systolic arrays, and the second is during the HLS lowering step. In this last step, polyhedral scheduling alone introduces a huge flexibility in terms of possible implementation, as illustrated by Figure 8. In addition to software enabled variance through tunable hyperparameters, this adds a dimension of hardware kernel variants that need to be ahead-of-time compiled.

In Deliverable D5.1, we present mARGOt as a runtime autotuner to implement both scenarios. Currently, we have the ability to estimate performance indicators for the variants we are able to generate, and classify them, all within our IR. We plan on extending this classification to certain plausible runtime scenarios, i.e. conditions under which the optimal trade-off between resources, accuracy, performance and other varying properties changes. We will include a set of variants for each of these classes into the compiled artifact, which mARGOt is then capable of selecting from at runtime. For instance, in our simplest demonstration scenario, this includes providing a variant for each target (e.g. CPU + bus attached FPGA + cFPGA), selecting based on availability.

4.2 Dataflow Transformations and Optimizations

Ohua conducts a number of dataflow transformations that have been outlined in Deliverable D4.1 already. These are mainly concerned with improving performance of the algorithm independently of the target architecture. Additionally, some transformations are necessary to nodes that are to be deployed to an FPGA.

4.2.1 Performance-Related Transformations

Naturally, a dataflow graph exposes task-level parallelism (nodes with no data dependencies between one another may execute in parallel) and pipeline parallelism. The first and perhaps simplest transformation therefore is to also enable data parallelism. Data parallelism arises from an implicit (in-)dependence between the same stateless function call across loop iterations. As such, every stateless function call inside a loop is an opportunity for data parallelism, but introducing data parallelism into a static dataflow graph as shown in Figure 9a leads to suboptimal performance. This is due to the assumption that all n nodes $f^1_{\text{SL}} \dots f^n_{\text{SL}}$ perform exactly the same computation which is often not the case. Instead, inputs often dictate how long a function will run. As such the deterministic merge in the `collect` node stalls waiting for straggling work [12].

To mitigate this effect without sacrificing determinism, we integrate dynamic dataflow into our static dataflow graph. In a dynamic dataflow graph, nodes are created at runtime. A node is a task such as executing a stateless function call that gets *spawned(/forked)* on demand and executes once. Spawning a task creates a handle to its *future value*, i.e., the result of the stateless functionc call. This handle provides a `get` method to *join* the forked and the current task by blocking until the call completed and the result is available. Tasks are

(a) Static data parallelism

(b) Dynamic data parallelism

Figure 9 – Static vs. dynamic data parallelism in the dataflow graph.

processed by a pool of threads. Whenever a thread is idling, it may steal tasks from other threads to reduce idle time. In case of the PTDR algorithm, a thread that already finished its delay computation may steal queued computations from a thread with a long-running delay computation.

The transformation in Figure 9b integrates dynamic dataflow to data-parallelize nodes with stateless function calls and uses the static dataflow to preserve the data value order, i.e, determinism and the semantics of the algorithm. Instead of replicating the stateless function call f_{SL} node, we lift it into a `spawn`<f_{SL}> node. For every received input, when normally a stateless function call would be executed, the `spawn`<f_{SL}> node submits this computation as a task to a work-stealing runtime system and emits corresponding *future*. The downstream `get` node retrieves the value from the future. No reordering takes place because both `spawn`<f_{SL}> and `get` are stateless function call nodes in the static datalfow graph connected via a FIFO channel.

When applications evolve around manipulating a large piece of shared state, they are called *irregular*. Parallelizing such computations often yields *amorphous data parallelism*. This means that the order in which elements from the input worklist are processed will dictate, how the remaining elements will be processed and whether new work elements may be created by processing an element. Hence, updates of the data structure not only depend on the input data, but also on the current state of the data structure itself. This direct loop-carried dependency means that parallelizing a loop operating on such a piece of state may not be parallelized in a straightforward manner. Existing approaches such as Software Transactional Memory allow the parallelization of these loops by wrapping them in transactions: Small code blocks which detect conflicting accesses to shared state and issue recomputations where necessary. However, these conflicts are an implicit side-effect that can significantly impact performance.

We provide a data parallelism transformation that makes these implicit effects explicit in the dataflow graph and exposes a knob to fine-tune runtime performance. The transformation targets irregular algorithms that (tail-)recurse over a worklist wl to evolve a complex data structure, i.e., a state s. We distinguish between two patterns: One, where the update to the state s happens inside of a loop and one where the state update happens after the loop has completed. The idea of this transformation is to limit the number of possible conflicts by running only a small number of computations from the worklist in parallel before updating the state. This ensures that the state is updated more frequently and following computations run on the updated state. Doing so exposes an inherent trade-off because smaller batch sizes lead to more iterations, which generate more overhead. At the same time, too big batch sizes lead to more conflicts, because more computations have been executed on the same state snapshot, leading to stale data that's no longer applicable. We depict our transformation that extracts amorphous data parallelism in Figure 10. In both cases, the `take_n`-node extracts the first N data items from the worklist wl and concatenates the $rest$ with the recomputations *after* the s was updated.

4.2.2 Transformations for Offloaded Kernels

Since Ohua regards the functions that algorithms are composed of as black boxes anyway, offloading single functions onto a hardware accelerator only affects the generation of the wrapper around the node. Instead of actually calling the function associated with the node, the wrapper becomes shallow. It merely forwards any incoming data to the FPGA and pipes computation results back to the output arcs of the node.

It is also conceivable to split the node into two parts, where the first will transmit the input data to the accelerator while the second one receives the computation results. This would enable pipelining the data transmissions for a possibly small performance gain.

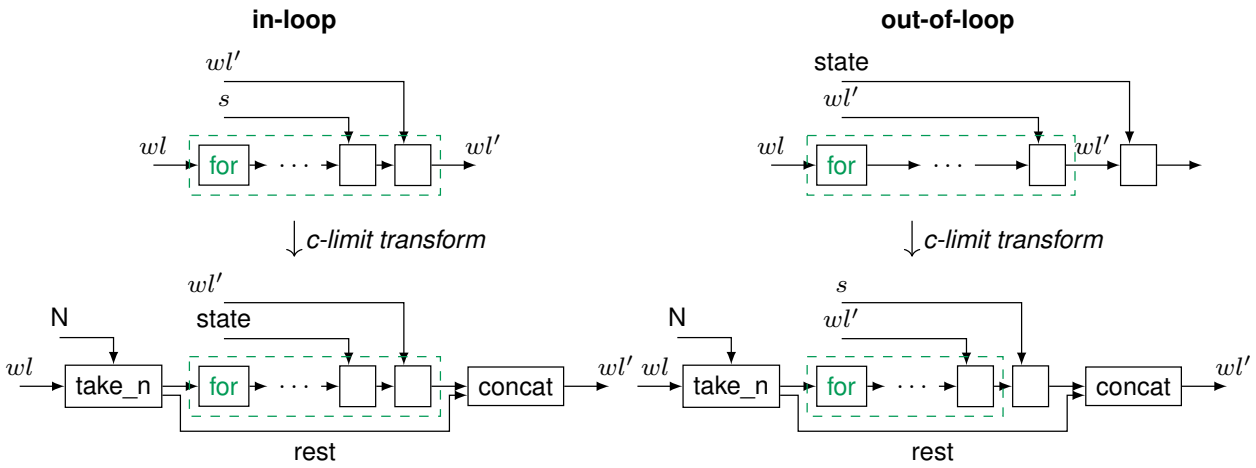


Figure 10 – Transformation for amorphous data parallelism

4.3 Machine Learning

Before describing the DSE that are applied to the imported RelayIR module of Section 3.2.4, we briefly elaborate on the motivation to do those transformations:

4.3.1 Engine and Streaming type of ML architectures

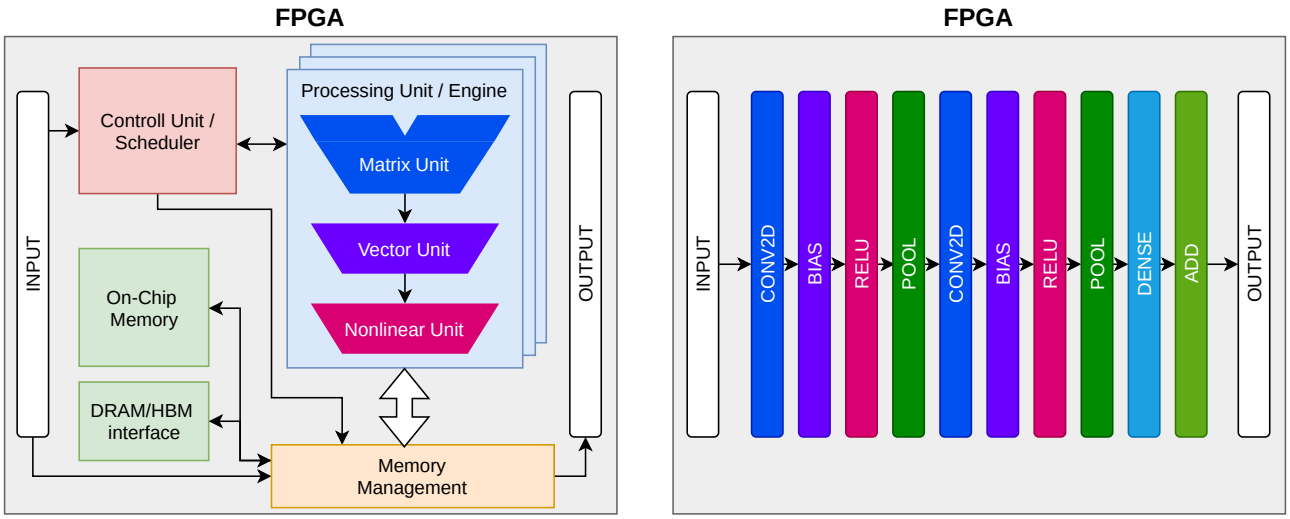
For over a decade, the FPGA community researches the acceleration potential of AI applications. The recent publicity and promotion of AI in the industry lead to a "Cambrian explosion"[9] of new architectures and products for accelerating DNNs on FPGAs, scaling from Edge to Cloud and for a wide variety of applications. Despite this variety, all existing frameworks can be sorted in two categories: **Engine-type** or **streaming-type** of acceleration (micro-)architectures, as depicted in Figure 11.

The first, engine-type, often also referred to as *NPU* or *xPU*, and shown in Figure 11a, consists of one or multiple custom designed processing units (i.e. engines) that can execute domain specific instructions. These processing engines often contain dedicated units for matrix multiplication, vector processing, and non-linear functions, since this are the mathematical foundations for today's DNNs. Consequently, a DNN is broken-down by a compiler into instructions that can be handled by those processing engines. 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: The processing elements can contain a variety of different specialized units, with different data sizes or types. In addition, the control unit and memory management can be either quite *stupid*, which means scheduling must mostly be decided by a compiler before run-time, or more intelligent with out-of-order execution or dynamic memory management. Examples for this type of acceleration architecture is TVM's VTA [16], Xilinx's Vitis AI [34], and Microsoft's Brainwave [9].

The second architecture template, the streaming-type, depicted in Figure 11b, *bakes* the application specific operations into the FPGA logic, so that at run-time the data *just* streams through the logic. This type of accelerator can achieve higher throughput with lower latencies, at the cost of a higher resource usage, compared to the engine-type. Despite this fixed principle, the design-space of this template is also huge: Starting with different data precision per operation to a large variety of loop-unrolling approaches. Example frameworks for this type of accelerators are hls4ml [5], Haddoc2 [1], or FINN [2].

4.3.2 Leveraging Existing ML-tools for FPGAs

There are already a lot of DNN-to-FPGA tools available in public literature. Each of these tools solves a particular challenge in a good, thought-through and efficient way, so why not reuse these efforts of the community? For example, if a user needs a solution for low-latency inference with small kernels on Xilinx FPGAs, there is an



(a) Engine-type accelerator (b) Streaming-type accelerator

Figure 11 – Two major types of NN accelerator architectures (Based on [32]).

actively developed framework available (hls4ml [5]). For Intel FPGAs, the user may choose another framework with engine-type [3]. In another situation, if the user has to solve a problem that requires extreme throughput, but could maintain it's accuracy with binary weights, LogicNets would be a solution [30].

However, for choosing this options, the user must be aware of their existence and also the trade-offs between them. With Dosa, we try to provide a holistic DNN-to-FPGA solution by automatically taking these decisions, but without "re-inventing the wheel" by re-implementing already proven solutions from the research community. Hence, we try to re-use existing (third-party) open source frameworks as much as possible and Dosa will offload a particular problem to another tool, if it detects this tool can solve this particular problem efficiently for the desired target device.

Following this path, Dosa needs a hardware-agnostic application-independent unified description of the DNN to be able to decide to offload which part to which tool. We decided to use RelayIR, as justified in Section 3.2.4.

4.3.3 DSE for ML Workload

Both architecture *templates*, streaming and engine, as discussed in the previous subsections are well justified for different reasons. The streaming template is best if used for smaller DNNs that require high-throughput and/or low latencies. Engine-type accelerators are better for larger DNNs, resource or cost constrained use-cases, or latency-relaxed environments. To decide which architecture is best for a particular network, performance characteristics and the targeted performance are required.

Alongside the input `.onnx`, the user of Dosa must provide *target constraints*, as also described in Deliverable D4.1. Those constraints state the resource budget, in terms of how many devices of which type are available, or the desired throughput in samples-per-second. After annotating the OI of each operation in the AST as described in Section 3.2.4, the required performance as well as bandwidth requirements are calculated for each operation and compared to the available bandwidth for a particular hardware. This analysis can best be visualized using a *Roofline* diagram, as shown in Figure 12. This analysis is performed for each operation twice: Once as streaming-type and once as engine-type, using the two types of OI annotations described in Section 3.2.4.

As can be seen in Figure 12, the OI for dense and convolution differs strongly for the different architecture templates, while max pool is indifferent. In this case, the dense operation in an engine-type accelerator would be heavily limited by the network or DRAM bandwidth, while the streaming-type accelerator would perform nearly optimal. In contrast to this, the difference of the OI of the two 2D-convolutions are with half-a-magnitude less significant and unimportant, since all versions are "only" compute-bound. But "baking-in" the two convolu-

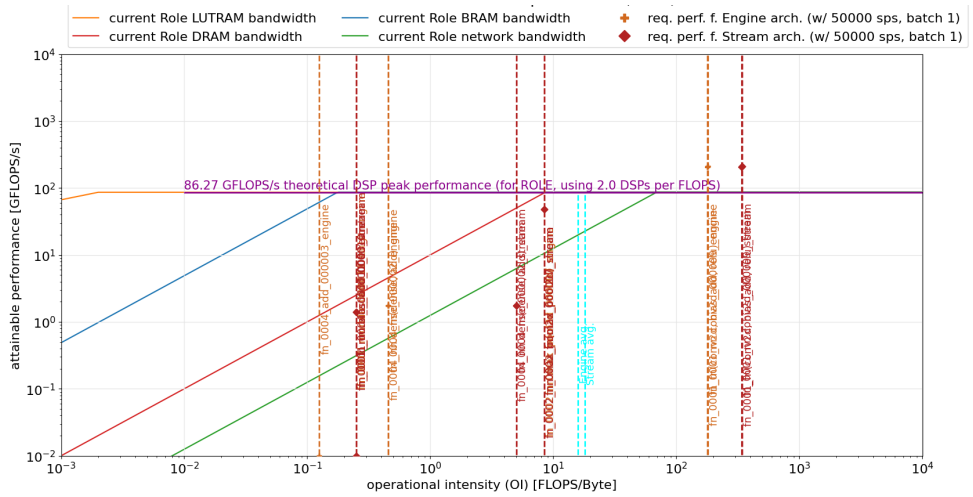


Figure 12 – Per-operation roofline analysis of the example convolution of Listings 1 and 2.

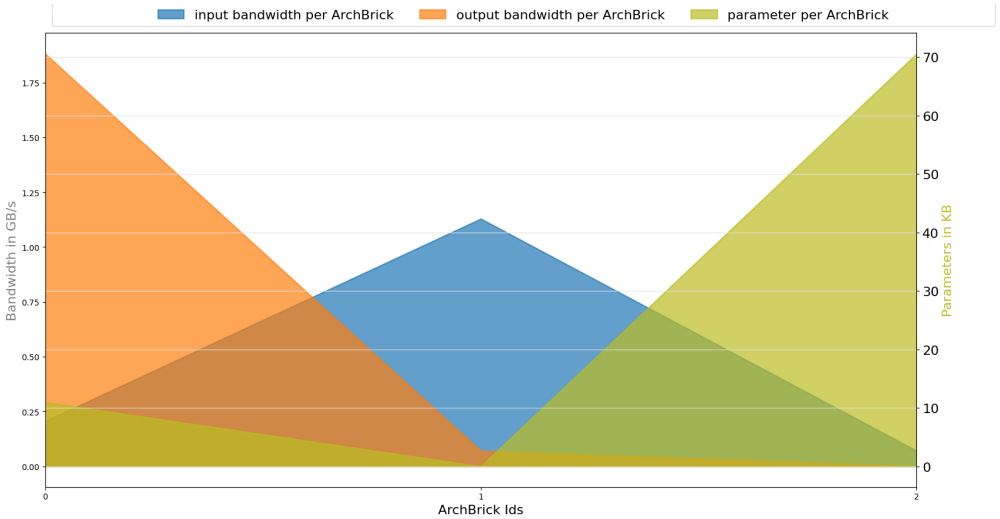


Figure 13 – Bandwidth and parameter requirements per operation for the example convolution of Listings 1 and 2. (`ArchBrickId` is roughly the layer number)

tions in the FPGA logic, would consume twice the resources than if both would use the same engine. Following this path, it would make sense to create an accelerator where the first layers are executed on an engine type and the dense layers at the end on a streaming architecture. This would achieve the same performance like an all-streaming approach, but would save resources. On the contrary, the required bandwidth for data between layers within a DNN tends to decrease throughout the network, as shown in Figure 13. Looking at this figure, one can get the opposite impression and could argue for a streaming-architecture in the beginning and engine-types in the end, to accommodate the high-bandwidth requirements of the first layers.

Both previous described proposals profit from the combination of streaming and engine templates. But which one is the “correct” or “better” one is only possible to tell after analyzing every operation of a given DNN, based on performance goals provided by the user. Based on this performance requirements, a compiler could decide if the engine-type accelerators in the beginning would be sufficient. Consequently, asking the user for performance targets, has two positive effects: First, it allows for a holistic analysis and DSE. And second, it enables early feedback to the user if the available hardware and architectures would meet this goals.

Based on this analysis, the best possible architecture for the available hardware is selected and forwarded to hardware generation. This step is described in Section 5.

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 EVEREST hardware generation flow is shown in Figure 14.

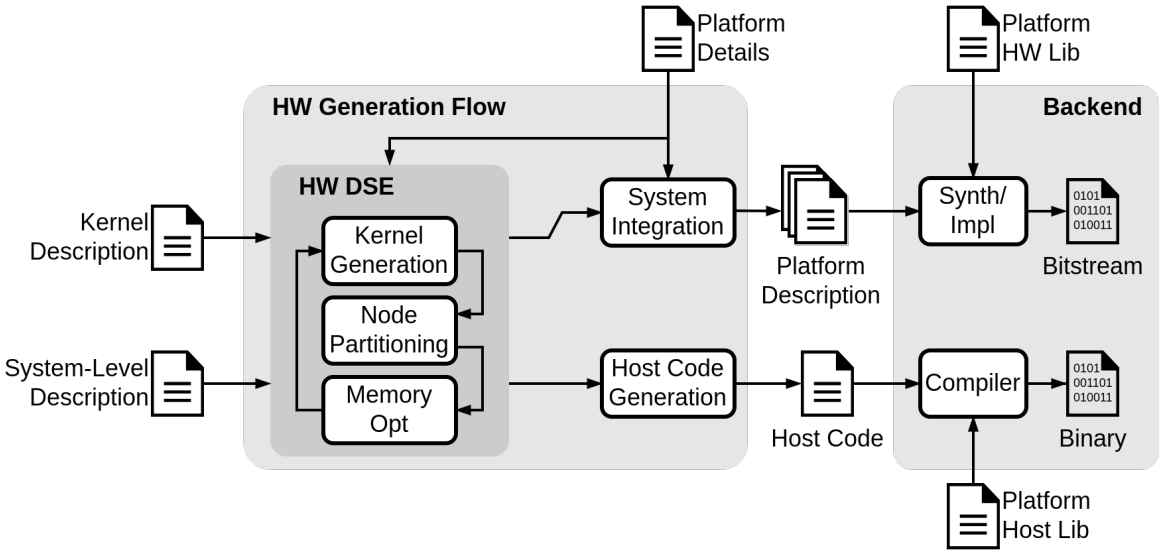


Figure 14 – EVEREST hardware generation flow.

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 a preliminary system level description that includes the minimal connectivity of the kernel. 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);
- it optimizes the on-chip memories by searching for sharing opportunities (Section 5.2);
- it creates the system-level description of the hardware architectures by replicating the kernels to operate in parallel and coordinate the associated data transfers based on the characteristics of the target platform (Section 5.3). 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 high-level description of the kernel is produced by the compiler 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 is instead dependent on the synthesis flow used to target the specific target node. For examples, we use Vivado HLS 2019.2 for IBM cloudFPGA nodes and Vitis HLS 2021.1 for the Xilinx Alveo nodes (see Section 5.3 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 include specific hardware-oriented optimizations to improve the hardware generation.

When the compiler flow emits C/C++, we currently use Xilinx HLS tools to synthesize the corresponding hardware descriptions. The `ap_fixed` library is used to specify custom data types so that they can be automatically synthesized.

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 an 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 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 exploit 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. The EVEREST SDK supports two main flows: one starting from an MLIR description and the other from a Rust application description. The former case takes as input the MLIR description generated after the optimizations covered in previous sections. MLIR is then lowered into the LLVM IR dialect and mapped to an equivalent LLVM IR description which is supported as an input description for the HLS flow. As well, the latter case takes as input a Rust application description which is compiled through the Ohua compiler. It generates a parallel dataflow runtime with the necessary glue code to interface with an off-loaded kernel. The Rust compiler is then used to generate LLVM IR in ways compatible with the downstream HLS flow. 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 HDL code, especially for ML applications, if the ML compilation flow detects that the usage of such libraries would produce the better result (cf. Section 4.3.2) . 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 proposed approach aims to leverage high-level code optimizations to provide a hardware-oriented input description to the HLS. Figure 15 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.

As previously mentioned, 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

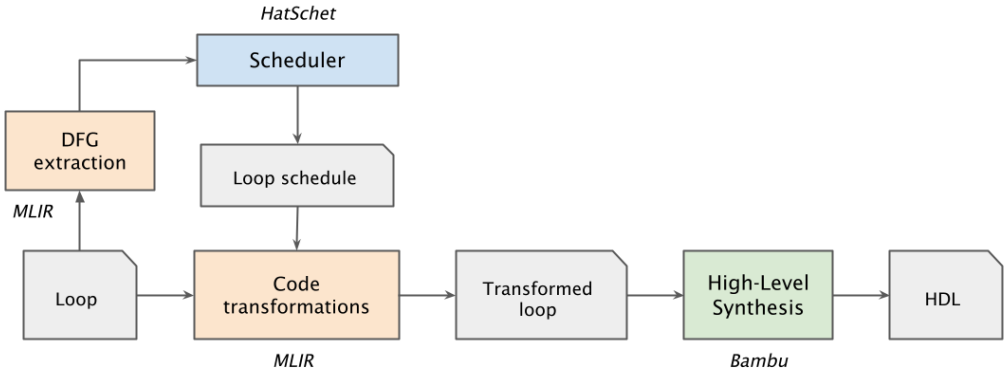


Figure 15 – Overview of the optimization flow for synthesis-oriented loop pipelining starting from MLIR description.

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.

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.

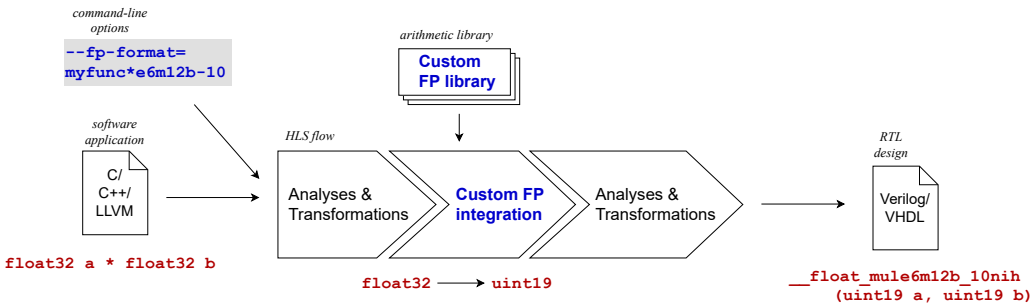


Figure 16 – Sample flow of custom floating-point application implementation through PandA Bambu HLS starting from generic input representation

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 16. 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 Section 3.2.2. 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 these custom types are converted into function calls to corresponding templatized functions from the Bambu HLS library. 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 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 [18]. 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 [10, 18]. 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 5.3) 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 that are reused across multiple blocks must be streamed through these blocks and buffered inside them to keep 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.

5.3 System Integration

The overall system architecture produced by the EVEREST system integration part is described in C++. The description wraps the kernels directly described in HDL, which are inserted as black-boxes. This C++ wrapper is later synthesized with platform specific tools. The flow also produces the platform configuration file based on the number of CUs that can be instantiated (if needed) and all script files for running the backend tools. Note that the configuration file specifies also the proper connections to the memory channels. These files are generated for each of the target nodes. Olympus reads the kernel interface to specify how to connect the input/output ports to the rest of the system. Data ports are connected to memory channels via AXI Master interfaces, while configuration ports are connected to the host via AXI-Lite, memory-mapped interfaces. Data exchanged with the memory channels are buffered on-chip to allow fast, fixed-latency access during the kernel execution.

The generation of the system architecture proceeds through an exploratory and optimization phase that requires information about the specific target platform. Figure 17 shows an overview of the flow. After the kernel level HLS, we can obtain an estimate of the resources needed to implement the kernel on the target FPGA device. Olympus generates the C++ description of the memory architecture around the kernel description, where each kernel description is integrated as custom RTL (*blackbox*). Since the hardware cost of the kernel may limit the number of parallel units, the Olympus exploratory phase is essential to understand which of the optimizations described above can be applied given the FPGA available resources. Indeed, we characterize each optimization with an estimation of the extra resources. With this information, Olympus assists the designer in selecting the most suitable optimizations and automatically generating the corresponding CU description around the HLS-generated code of the kernel and the system configuration file for interfacing with the synthesis

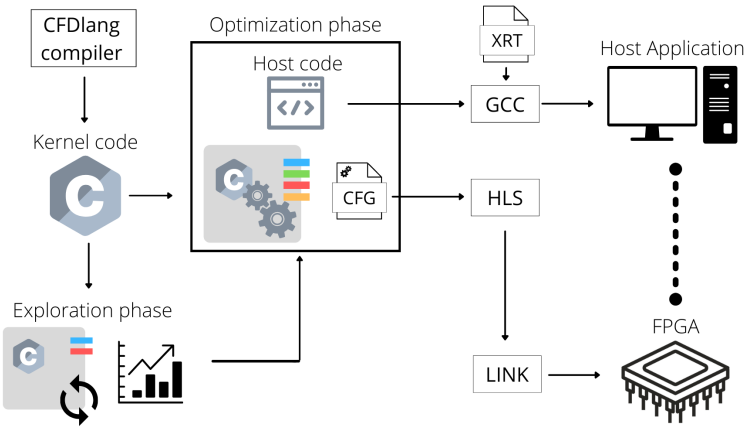


Figure 17 – Olympus hardware generation flow

tools. Each CU can feature multiple kernels, each of them connected to a lane to fully utilize the AXI bandwidth. The wrapper implements data-movement optimizations and are designed accordingly with changes to the host application and the configuration file. For example, the kernel may benefit from a change in the way data is written to and read from global memory and therefore the host application must adapt to this type of behavior (see Deliverable D3.1 for more details). The configuration file, instead, defines how each CU interfaces with the memory. By modifying the configuration file, Olympus optimizes how each CU is connected to the individual channels. The resulting components are then passed to the synthesis tool. Olympus implements abstract classes to specify the common structure of the backend, which is then specialized based on the specific target node. For example, we support Vivado for cloudFPGA and Vitis for the Alveo-based servers. The synthesis tools automatically generate the bitstream required for board configuration. Supporting a new target platform only requires to create the proper specialization of the backend.

5.4 Integration Test: The Case of Computational Fluid Dynamics

In this section, we show a prototype flow that we used to evaluate the integration of our methods in the generation of several implementations of the CFD application (Inverse Helmholtz operator). Our DSL-to-FPGA combines the frontend compiler, one of the HLS tools, some memory optimizations, and the system integration step (cf. [23]). As discussed in Deliverable D4.3 and Deliverable D4.4, CFDlang is implemented on top of the MLIR infrastructure, Mnemosyne is an open-source tool[1], and Olympus is a new in-house prototype. Olympus is built in Python on top of the Pyverilog library [25] for hardware generation (i.e., the generation of the kernel wrappers around Mnemosyne artifacts) and the Pycparser library[2] for code generation. With our flow, we aimed at targeting a Xilinx Alveo U280 card (one of the possible FPGA targets in EVEREST – see Deliverable D6.2. We used Xilinx Vitis 2021.1 for synthesis generation and bitstream creation. Unless otherwise specified, we target a synthesis frequency of 450 MHz for both the platform and the CU description.

In the following, we evaluate the cumulative benefits introduced by each optimization, we discuss the major challenges in the implementation of CFD applications, and we compare our results with Intel ones [20] in terms of performance and energy efficiency by using the GFLOPS and GFLOPS/W metrics, respectively. In particular, given the polynomial degree p, we assume that each contraction is composed of three loops that execute two floating-point operations (one addition and one multiplication) for $p \times p \times p \times p$ times each. Similarly, the Hadamard product requires $p \times p \times p$ multiplications. So, the entire Inverse Helmholtz operator the following number of floating-point operations:

$$N_{op}^{el} = 2 \cdot [2 \cdot (p \cdot p \cdot p \cdot p) + 2 \cdot (p \cdot p \cdot p \cdot p) + 2 \cdot (p \cdot p \cdot p \cdot p)] + (p \cdot p \cdot p) = (12 \cdot p + 1) \cdot (p \cdot p \cdot p) \quad (1)$$

We tested the flow by creating the systems for two polynomial degrees, i.e., $p = 11$ and $p = 7$. So, a single

[1] https://github.com/chrpilat/mnemosyne
[2] https://github.com/eliben/pycparser

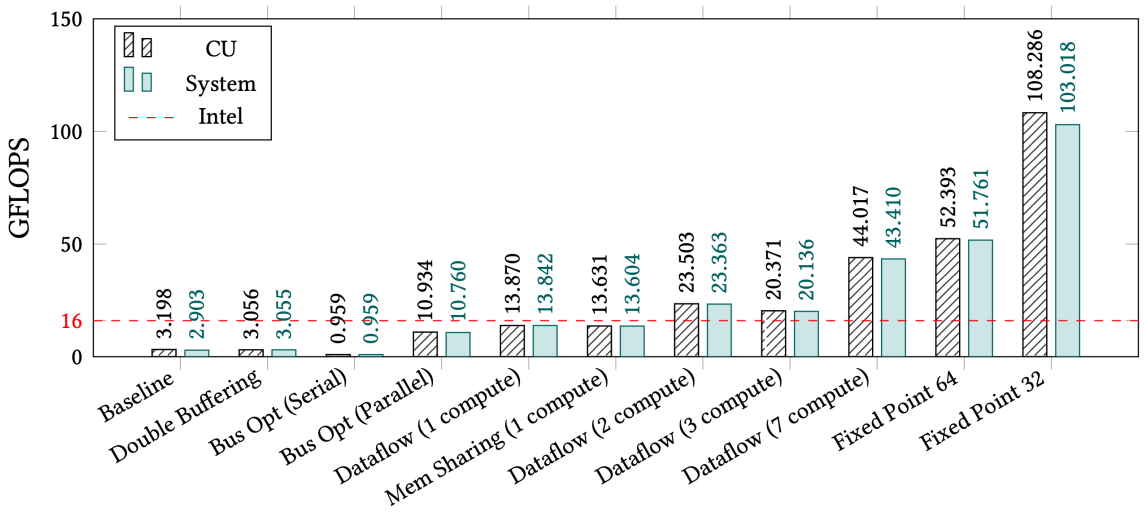


Figure 18 – Performance of each optimization implemented with 1 CU and $p = 11$.

element requires to execute N_{op}^{el}=177,303 floating-point operations when $p = 11$ and N_{op}^{el}=29,155 floating-point operations when $p = 7$. The total number of floating-point operations for a CFD simulations can be obtained as:

$$N_o p = N_{eq} \times N_{op}^{el} \tag{2}$$

We executed all experiments with $N_{eq} = 2,000,000$, i.e., we simulated 2,000,000 elements. The GFLOPS metric is then obtained by dividing this number for the application execution time, while the GFLOPS/W metric is obtained by dividing the GFLOPS metric by the average power consumption of the system. To get accurate information about power consumption, we profiled the power consumption during the system execution with Xilinx XRT.

We executed our CFDlang on the DSL description in Figure 2 to generate the C kernel for hardware optimization and HLS. We first performed experiments to evaluate the effects of optimizations with $p = 11$. In particular, we progressively added the following optimizations:

- Baseline: No optimizations are used. The code executes the kernels and data transfers in series, while each compute unit contains only one kernel and it is connected to the HBM with 64-bit AXI channels.
- Host-HBM Double Buffering: We introduce double buffering to hide CPU-FPGA communication latency.
- Bus Optimization: We evaluate the effect of widening the bus to 256 bits, with only one kernel unit (and serializing the data) and with multiple lanes feeding parallel kernel units.
- Dataflow optimization: We create several variants of the compute functions with one, two, three, and seven subkernels. We indeed evaluate the performance vs. resources trade-off.
- Resource Optimization: We apply on-chip memory sharing (only in the case of dataflow implementations with one block inside the compute part) and fixed-point data types (with 64- and 32-bit implementations).

For each of these implementations, we measured total and kernel execution times, maximum and average power consumption, and cost in terms of hardware resources. Figure 18 shows the performance (in terms of GFLOPS) achieved in each experiment when adding the specific optimization on top of the previous ones. In each experiment, the left red bar shows the GFLOPS of the CUs on their own, without considering host-FPGA data transfers, while the right blue bar includes the entire application. Comparing the two bars allows us to evaluate the peak performance of the kernels and the effects of data transfers.

The Baseline case achieves only 3 GFLOPS while Intel implementations are around 16 GFLOPS. Also, the difference between the CU performance and the overall system performance is significant. This is due to the serial nature of the implementation where data is transferred from the host to the HBM, then processed by the CU and sent back to the host before starting a new batch.

	f_{max} (MHz)	LUT		FF		BRAM		URAM		DSP	
Baseline	274.6	141137	(10.8%)	214402	(8.2%)	244	(12.1%)	57	(5.9%)	150	(1.7%)
Double Buffering	259.8	148873	(11.4%)	228561	(8.8%)	246	(12.2%)	57	(5.9%)	150	(1.7%)
Bus Opt (Serial)	286.5	146088	(11.2%)	225542	(8.7%)	268	(13.3%)	3	(0.3%)	55	(0.6%)
Bus Opt (Parallel)	296.6	182632	(14.0%)	295340	(11.3%)	330	(16.4%)	12	(1.3%)	192	(2.1%)
Dataflow (1 compute)	286.2	215199	(16.5%)	335009	(12.8%)	330	(16.4%)	240	(25.0%)	592	(6.6%)
Dataflow (2 compute)	291.9	291964	(22.4%)	446258	(17.1%)	330	(16.4%)	240	(25.0%)	1068	(11.8%)
Dataflow (3 compute)	266.3	293757	(22.5%)	448385	(17.2%)	298	(14.8%)	164	(17.1%)	1096	(12.1%)
Dataflow (7 compute)	199.5	473743	(36.4%)	735030	(28.2%)	330	(16.4%)	252	(26.3%)	3016	(33.4%)
Mem Sharing (1 compute)	282.4	229115	(17.6%)	336133	(12.9%)	282	(14.0%)	124	(12.9%)	592	(6.6%)
Fixed Point 64	233.8	254242	(19.5%)	342390	(13.1%)	330	(16.4%)	252	(26.3%)	4368	(48.4%)
Fixed Point 32	244.5	231062	(17.7%)	346507	(13.3%)	1338	(66.4%)	0	(0.0%)	2294	(25.4%)

Figure 19 – Resource utilization for each optimization implemented with 1 CU and $p = 11$. Highlighted in red is any value over 25% utilization, indicating possible issues when instantiating more than one CU.

After the Double Buffering optimization, the CU performance remains similar, with a small degradation due to overhead, while the system performance is now exactly the same as the CU performance. This is an improvement over the Baseline implementation, because now the host to HBM data transfers are happening in parallel to and are entirely hidden behind the CU execution.

We then executed two experiments for evaluating Bus Optimization. In the *Serial* version, we attempt to utilize the full bandwidth of the 256-bit bus by packing four doubles. The CU reads them in parallel but then it serializes them when it needs to access its own local buffers. While this optimization is supposed to speed up data reads from the HBM, its implementation in the CU leads to a performance *degradation* of about $3\times$. This is mostly due to the complexity of the logic for aligning the data that is not well handled by HLS tools. To mitigate this, but still use the full bus bandwidth, this optimization was replaced with the *Parallel* implementation where four kernels are instantiated in the CU and the data for each "lane" is stored in separate buffers, one for each kernel. This led to a $3.52\times$ speedup over the Double Buffering optimization. This is close to the ideal speedup of four when using four kernels in parallel, and the discrepancy can be attributed to the additional hardware complexity that slightly decreases the execution frequency. This *Parallel* implementation is used in the following experiments.

Next, we tested various forms of the Dataflow Optimization. Each implementation of this optimization separated the kernels into read, compute, and write modules and streams were used to pass data between them, allowing a pipelined structure. When using one compute subkernel (*Dataflow (1 Compute)*) test, the speedup was $1.29\times$ due to the overlapping execution of the read, compute, and write modules. However, the compute module was dominating the execution time so it was further split into 2, 3, and 7 modules. All three of these tests gained speedup over the *1-Compute* version by breaking the total execution time of a single module down further. However, *3-Compute* modules was slower than *2-Compute* modules. Indeed, in each case, the module with the longest latency was the same, but the extra modules and control routing caused the tools to frequency scale the *3-Compute* case to execute at 266 MHz whereas the *2-Compute* case executed at 292 MHz. When this is taken into account, the performance of both tests is approximately the same. The *7-Compute* test, however, performed the best because each of the compute modules was much smaller than the previous tests. In this case, the latencies of these modules were now slightly shorter than the latency of the read module, meaning that this is the limit of the performance increase by dividing the compute portion. The *7-Compute* test gained a total speedup of $4.03\times$ over the Bus Opt *Parallel* implementation.

At this point, we want to replicate the CUs using the remaining area available in the FPGA fabric to maximize parallelism. We first evaluate the hardware cost of each implementation. The numbers of LUT, FF, BRAM, URAM, and DSP used by each case for $p = 11$ are shown in Figure 19. In general, each test from Baseline to Dataflow (7 Compute) showed an increase in resource utilization. Any utilization value over 25% is shown in red. These are the resources most likely to cause placement and routing issues when instantiating multiple CUs. We tested a few methods to reduce resource utilization and increase the number of instantiated CUs.

In this circumstance, the *Mem Sharing* optimization does not apply to the Dataflow *7-Compute* implementation, because each compute module only uses arrays which cannot be shared, as they are always in use during the execution of the module. Instead, it can be applied only to the Dataflow *1-Compute* implementation where several arrays are used in the compute module. Mnemosyne generated an architecture to internally

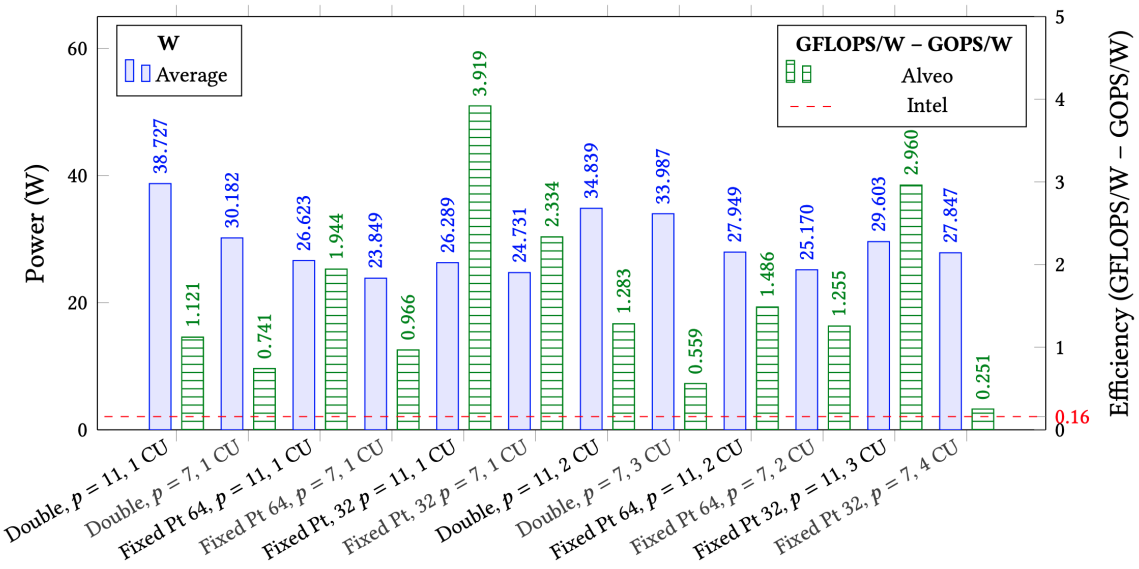


Figure 20 – Power usage and energy efficiency of the Dataflow (7 Compute) optimization with each datatype, $p = 11$ or $p = 7$, and 1-CU or multiple-CU.

share arrays based on their liveness intervals. This decreased the BRAM utilization by 14.5% and the URAM utilization by 48.3% while the LUT and FF utilization only increased minimally and the DSP utilization remained the same. Also, the execution time was only slightly reduced (a slowdown of $0.98\times$). This optimization is beneficial when on-chip memory inside the compute unit is the limiting factor and replicating the CUs can bring more improvements than dataflow execution.

Another method to reduce resources is to change the numerical representation. All of the previous tests used the floating-point format with double precision. In general, fixed-point representations utilize fewer resources than floating-point ones. We tested 64- and 32-bit fixed-point representations by modifying the Dataflow *7-Compute* implementation. The 64-bit implementation uses 24 bits for the integer portion and 40 bits for the fractional portion. The 32-bit implementation uses 8 bits for the integer portion and 24 bits for the fractional portion. These values are provided by the user after an analysis of the algorithm. Because the 32-bit data is half the size, we instantiate 8 kernels per CU and divide the 256-bit bus into 8 lanes. In the *Fixed Point 64* test, the LUT utilization reduced by 46.3%, the FF utilization reduced by 53.4%, the RAM utilization remained the same, and the DSP utilization increased to 44.8%. In the *Fixed Point 32* test, with respect to the *Fixed Point 64* test, the LUT and FF utilization remained roughly the same. The DSP utilization was nearly halved. The BRAM increased by about four times while the URAM decreased to zero. This is because the data representation is half as long, so the overall size of the data structures are half as big. The arrays representing the tensors are no longer big enough for the tool to decide it is efficient to use URAM to store them. When taking into account the size of the physical memories, the total memory space is approximately halved. The performance of the *Fixed Point 64* test had a slight speedup of $1.19\times$ due to the simplification of the logic allowing the frequency to be higher. The Dataflow *7-Compute* test with double format was scaled to 199 MHz while the *Fixed Point 64* test was scaled to 234 MHz. The performance of the *Fixed Point 32* test had a speedup over the double format of $2.37\times$ and it reaches up to 103 GFLOPS. This represents a speed up of more than $35\times$ over the Baseline version. The *Fixed Point 64* test exhibited a mean square error of 9.39×10^{-22} while the *Fixed Point 32* test had a mean square error of 3.58×10^{-12}. It is up to the application designer to determine what an acceptable error is and decide on an appropriate number format, and our flow can help facilitate a design space exploration of these parameters.

Figure 20 shows the power consumption of the different implementations and a comparison of the energy efficiency (GFLOPS/W or GOPS/W depending on the data format) with the Intel implementation. The bars reported the average power consumption measured with the XRT infrastructure. The dashed red line represents the power efficiency estimation (0.16 GFLOPS/W) of the Intel platform which is the performance obtained in [20] (16 GFLOPS) divided by the thermal design power of the CPU (100W – conservative estimate). Since the results available in [20] refer to a vectorized implementation on a single thread, the fair comparison is with the 1-CU implementation for fair comparison. We also include the results of the multiple-CU implementations, to

show the effects of replication on both power consumption (W bars) and energy efficiency (G(FL)OPS/W bars). As expected, the fixed-point implementations are more efficient than the floating-point ones. Also, reducing the bitwidth from 64 to 32 bits allows us to achieve the maximum efficiency. This is because these implementations are much faster and use less hardware resources. The $p = 7$ implementations have lower average power consumption than their $p = 11$ counterparts, due to their smaller resource utilization. However, in most cases the efficiency of the $p = 7$ cases is lower due to their longer overall execution time. The multiple-CU implementations are generally less efficient than their single-CU counterparts, both because of the increased work occurring in parallel, yielding a higher average power, and because of longer execution times from frequency scaling. While all implementations are far more efficient than Intel ones, the most efficient cases are *Fixed Point 32* with $p = 11$ and 1 CU and the same case with $p = 7$. These cases are about $25\times$ and $15\times$ more efficient than the Intel estimate, respectively.

6 Code Generation and Runtime Integration

Sections 4.1-4.3 partly described how code can be generated for the different classes of applications or kernels. In general, and as depicted in Figure 1, there are multiple possible paths after the middle-end into the downstream compilation process. For every type of kernel, the compiler framework can generate standalone implementations that run on CPUs. This is achieved via source-to-source compilation (e.g., DSL-to-C, experimental MLIR-to-C, Ohua(Rust)-to-Rust), by using the default TVM code generation for CPUs, or by using the LLVM compiler to process LLVM-IR.

A large focus of the EVEREST project is on interfacing with HW generation flows (cf. Section 5). As shown in Figure 1, we currently support multiple paths to this end, including C code with pragma annotations and direct interfacing via compiler IR. As described before, and illustrated in Figure 4, we see great potential in rich interfaces via MLIR as we do with the Bambu HLS tool.

For the runtime integration, different versions of a kernel or application can be generated, as illustrated in Figure 8 for instance. With this deliverable we thus describe provide an initial way forward to interfacing with the runtime and auto-tuning support described in Deliverable D5.1.

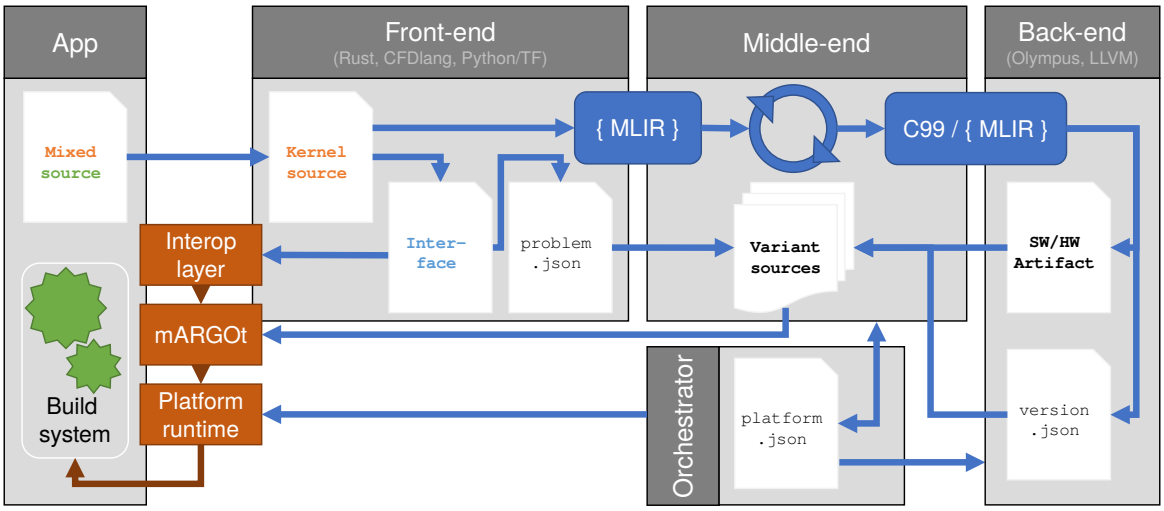


Figure 21 – Application integration workflow.

Figure 21 shows an overview of how the compiler toolchain integrates an application for the EVEREST platform. Following the observations from Figure 4, we place an MLIR-unified front-end component at the start of the flow, which extracts the relevant sources. It also fulfills the role of language and build system integration with the original application – it must analyze and describe the kernel interfaces to later stages, while offering a mechanism to rebuild the app with their changes. We then move on to one of the middle-ends described in Section 4. This component accepts the problem definition from the front-end, and is expected to produce a set of code variants that integrate with mARGOt. This assumes that the middle-end can consume and produce platform descriptors , and discover variants automatically. In a simplified flow, mARGOt is not used and only one artifact is generated using a static platform description, such as described in Section 4.3. The middle-end depends on a back-end to generate the platform-specific artifact, which could be an LLVM toolchain, or our hardware flow described in Section 5.

An implicit dependency on a platform description and the deployment infrastructure is represented by the orchestrator component in Figure 21. The components of this flow are reflected in a chain of dependencies that are added to the application. These libraries are provided by the EVEREST SDK or built by the compiler flow, and together they constitute the application runtime. Following the terminology from Figure 21, the following layers of dependencies are found:

Interop layer. At the most abstract level, the application needs to be rebuilt using the modified kernel implementations. This requires glue code to connect the extracted parts back to the original application with as little manual labor as possible. In this interop layer, there are static, language-specific libraries distributed with the EVEREST SDK, and application-specific sources. The latter contains code that is generated by the

front-end to achieve interoperability with its extracted interface, and in some cases user-provided changes to the original application.

mARGOt variants. The main output of our flow are the actual kernel implementations, which are called from the interop layer. In the simplest case, mARGOt is not used, and a single variant is compiled and referenced directly. In all other scenarios, which include those that can adapt to the hardware configuration present at runtime, mARGOt is used to implement a single entry-point per kernel. The mechanism for bundling this variant set is described in Deliverable D5.1.

mARGOt runtime. Whenever the application runs with mARGOt, mARGOt's runtime and autotuning facilities need to be made available to the variant bundle. Thus, a multi-variant flow adds an implicit dependency to the statically distributed mARGOt library itself.

Platform runtime. At the bottom level of these transitive dependencies, a platform-specific runtime library must be statically distributed for every EVEREST platform. This library allows the kernel implementation, which contains both the host and device code for the generated replacement, access to the hardware. The platform runtime is therefore specific to the hardware configuration that is used, and in the case of HPC environments, depends on the resource management facilities. In our case, the latter are implemented by the orchestrator, which controls access to the devices that kernels will be deployed to at runtime.

7 Conclusions

In this deliverable we described the current status of the compilation framework, explaining how we have thus far implemented the definition from Deliverable D4.1. We explained how multiple different components (cf. Figure 1) interoperate to provide support for the challenging landscape of languages and requirements of the EVEREST use cases. The extension to tools, IRs and the novel compilation and hardware generation flows demonstrated a promising initial compiler framework with which efficient HW-SW implementations of key components from the use cases can be generated. This deliverable also show how data-related design decisions from Deliverable D3.1 are implemented (e.g., data allocation and number representations) and an initial proof-of-concept of the connection to the runtime system described in Deliverable D5.1. We have shown how we can generate code to the different types of nodes of the EVEREST platform, with focus on FPGA acceleration. The details on the usage of the tools are however relegated to Deliverable D4.3.

Moving forward, the compilation framework will evolve into full support of end-to-end use cases, as opposed to isolated software components. With the runtime system in place, the interaction with WP5 will be intensify in the second half of the project. As already alluded at the start of this deliverable and in the project proposal itself, we will continue the efforts towards integration of the analysis and synthesis flows.

Acronyms

API Application Programming Interface. 6, 37

AST Abstract Syntax Tree. 9, 18, 37

CFD Computational Fluid Dynamics. 6, 8, 30, 31, 37

CU Computational Unit. 28–34, 37

DFG Dataflow Graph. 10, 11, 37

DNN Deep Neuronal Networks. 17, 22–24, 37

DSE Domain-Space Exploration. 4, 19, 20, 22–24, 37

DSL Domain-Specific Language. 6, 8, 9, 11, 13, 14, 19, 30, 31, 35, 37

FFI Foreign Function Interface. 9, 37

FPGA Field Programmable Gate Array. 6, 7, 11, 17, 19–26, 28–32, 37

GEMM general matrix-matrix multiply. 19, 37

HBM High Bandwidth Memory. 19, 31, 32, 37

HDL Hardware Description Language. 18, 26, 27, 37

HLS High-Level Synthesis. 5–7, 11, 14, 18–20, 25–32, 35, 37

HPC High-Performance Computing. 5–8, 36, 37

IR Intermediate Representation. 11, 13, 14, 16, 19, 20, 26–28, 35, 37

JSON JavaScript Object Notation. 26, 37

ML Machine Learning. 4, 11, 17, 19, 22, 23, 26, 37

MLIR Multi-Level Intermediate Representation. 4, 6–9, 11, 13–15, 17, 19, 20, 26, 27, 35, 37

OI Operational Intensity. 17, 37

ONNX Open Neural Network eXchange. 11, 17, 37

PLM Private Local Memory. 37

RRTMG Rapid Radiative Transfer Model for GCM Solvers. 9, 37

RTL Register Transfer Level. 7, 26, 28, 29, 37

SIMD Single Instruction Multiple Data. 19, 37

SLP Superword Level Parallelism. 17, 37

TCL Tool Command Langauge. 26, 37

UB Undefined Behavior. 8, 37

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] 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.

[3] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus. In 2020 International Conference on Field-Programmable Technology (ICFPT), pages 10–19, December 2020.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[5] 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.

[6] 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.

[7] Sebastian Ertel, Justus Adam, Norman A Rink, Andrés Goens, and Jeronimo Castrillon. Stclang: State thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, pages 146–161, 2019.

[8] 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.

[9] 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.

[10] 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.

[11] Google, Inc. Protocol Buffers, 2022.

[12] Kim P Gostelow and Wil Plouffe. Indeterminacy, monitors, and dataflow. In Proceedings of the sixth ACM symposium on Operating systems principles, pages 159–169, 1977.

[13] 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.

[14] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. Proceedings of the ACM on Programming Languages, 4(POPL):1–32, 2019.

[15] Devin A. Matthews. High-performance tensor contraction without transposition. SIAM Journal on Scientific Computing, 40(1):C1–C24, 2018.

[16] 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.

[17] 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.

[18] 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.

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[20] 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.

[21] 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.

[22] Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, and Martin Kong. Occ: An automated end-to-end machine learning optimizing compiler for computing-in-memory. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 1–1, 2021.

[23] Stephanie Soldavini, Karl FA 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. arXiv preprint arXiv:2203.10850, 2022.

[24] 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.

[25] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, Applied Reconfigurable Computing, pages 451–460, Cham, 2015. Springer International Publishing.

[26] The Apache Software Foundation / TVM community. Apache TVM, 2022.

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

[28] The ONNX community. Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification, 2022.

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

[30] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In Proceedings of the 30th IEEE International Conference on Field-Programmable Logic and Applications (FPL), pages 291–297. IEEE, 2020.

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:1802.04730, 2018.

[32] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. ACM Comput. Surv., 51(3):56:1–56:39, June 2018.

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

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