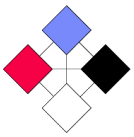
D3.4
Compiler Design and Overview



DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.2

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

Previous deliverables already shared the overall and refined system architecture [D2.1, D2.2], the language design specification [D3.1], as well as the initial and extended compiler prototypes [D3.2, D3.3]. This document presents the DAPHNE compiler design and overview. The DAPHNE compiler is based on MLIR (multi-level intermediate representation) [LA+21] as a framework for domain-specific compilers to facilitate a cost-effective development of our domain-specific language, reuse of compiler infrastructure, and good extensibility. This document first provides the necessary background on MLIR. After that, it presents the DAPHNE compiler design including a high-level overview, the intermediate representation DaphneIR, the compilation chain, and cross-cutting topics of configurability and extensibility. After that, it briefly recaps related work. A prototype of the DAPHNE compiler has been publicly available as open-source code as a part of the DAPHNE GitHub repository since March 2022.

D3.4 Compiler Design and Overview			
WP3 – DSL Abstractions and Compilation			
Type of document	R	Version	1.2
Dissemination level	PU	M36 (Nov 2023)	
Lead partner	TUB		
Author(s)	Patrick Damme (TUB), Philipp Ortner (TUB)		
Reviewer(s)	Dimitrios Tsoumakos (ICCS), Marius Birkenbach (KAI)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial structure and write-up	Patrick Damme (TUB)
V1.1	Incorporated feedback by Dimitrios Tsoumakos (ICCS), Marius Birkenbach (KAI), and Philipp Ortner (TUB)	Patrick Damme (TUB)
V1.2	Additional polishing and finalization	Patrick Damme (TUB)

1 Introduction

Integrated data analysis (IDA) pipelines combine data management and query processing, machine learning training and scoring, and high-performance computing workloads. Deploying IDA pipelines is still a cumbersome task, since traditionally, dedicated systems, frameworks, and libraries have been developed for each of the three areas. To increase user productivity and to avoid performance overheads resulting from the combination of multiple systems, DAPHNE [DB+22] is an open and extensible system infrastructure for developing and executing IDA pipelines. The DAPHNE system architecture includes user-facing language abstractions, a domain-specific optimizing compiler, a runtime for local and distributed execution, multi-level scheduling, hardware accelerators, and computational storage. This deliverable focuses on the DAPHNE compiler as a central component of the overall system, with many interaction points to the other components.

Many systems from the three areas of IDA pipelines employ optimizing compilers [AB+16, BA+20, BK99, N11]. Typically, these systems implement a domain-specific compiler to perform high-level optimizations that would be very hard to recover using a general programming language compiler. In order not to reimplement the entire compiler infrastructure, we decided to base the DAPHNE compiler upon MLIR [LA+21], a framework for domain-specific compilers from the LLVM [LA04] ecosystem. The optimizing compiler has a global view on the IDA pipeline, which offers unique opportunities for optimizing the program in order to improve the runtime performance, memory footprint, or runtime-accuracy trade-off.

Some details on the design of the DAPHNE compiler have already been presented in earlier DAPHNE project deliverables [D2.1, D2.2, D3.1, D5.1]. Furthermore, an initial and an extended prototype of the DAPHNE compiler have already been delivered [D3.2, D3.3]. In this document, we report on the overall design and high-level overview of the internals and key techniques used in the DAPHNE compiler. Compared to the previous deliverables, this document is the first comprehensive and holistic design document of the DAPHNE compiler. Moreover, since the extended compiler prototype [D3.3] we have added support for code generation (besides various smaller improvements), which we describe here for the first time. In detail, the contributions of this document are:

1. We recap the necessary background on optimizing compilers in general and MLIR in particular, to ease the understanding of the DAPHNE compiler design (Section 2).
2. We present the design of the DAPHNE compiler, including a high-level overview, the intermediate representation DaphneIR, the DAPHNE compilation chain, and cross-cutting topics like configurability and extensibility (Section 3).
3. We briefly summarize related work on optimizing compilers in the fields relevant to IDA pipelines (Section 4)

Finally, we conclude the document in Section 5.

2 Background

In this section, we briefly provide some necessary background and concepts to understand the design of the DAPHNE compiler. We first recap the topic of optimizing compilers in general (Section 2.1) and then provide more details on MLIR, the compiler-framework the DAPHNE compiler is based on (Section 2.2).

2.1 Optimizing Compilers

A *compiler* is a software system that translates human-readable source code in some high-level programming language to executable machine code for some target hardware architecture.

Intermediate representations. Compilers typically work on an *intermediate representation* (IR) of the input program, which is more suitable for analyses and optimizations than the text-based input format. An IR normally contains operations to execute plus ways to attach additional information to these operations, e.g., types, properties, and any additional attributes. There is no standard IR that is used by all compilers, but there are typically some similarities. A widely used structure of IRs is the so-called *static single assignment* (SSA) form. In SSA, operations produce values, whereby each value is defined exactly once. Instead of changing an existing value, a new value must be created. Values can be read many times. SSA is widely used in compilers since it simplifies the analysis of complex programs. Examples of widely known IRs include the LLVM IR [LA04] and Java bytecode [LY99].

Structure of a compiler. A prototypical structure of a compiler consists of three main components.

- The *front end* takes the textual representation of a program in a high-level language as input. It performs various analyses, such as lexical analysis, syntactic analysis, and semantic analysis. In the end, it generates the IR of the program.
- The *middle end* takes the IR as input and performs various optimization passes on it. The goal of these passes is to improve the program in terms of optimization goals like the runtime performance, memory footprint, compiled binary size, or other application-specific targets. To this end, the passes may add, change, reorder, or remove operations. By default, these rewrites retain the semantics of the program. However, there are also cases when slight deviations are accepted in the interest of improved performance, e.g., for fast mathematical operations or when exploring the runtime-accuracy trade-off. All passes applied at this stage are independent of the target hardware platform, and many compilers allow to enable/disable concrete optimizations to define a trade-off between compilation time and the quality of the generated code. The output is still in the compiler's IR.
- The *back end* consumes the IR optimized by the middle end and performs additional optimizations specific to the target hardware platform, e.g., the concrete processor architecture. Its output is machine code for that target.

In this deliverable, we mainly focus on the middle end of the DAPHNE compiler, since it is the most decisive part for us. The front end is an ANTLR-based parser, and for the back end, we reuse existing facilities from the MLIR/LLVM compiler stack.

2.2 MLIR: A Framework for Domain-specific Compilers

The DAPHNE compiler is based on the *Multi-level Intermediate Representation* (MLIR) [LA+21], a framework for building domain-specific compilers. The MLIR project was started by a team at Google TensorFlow [AB+16] led by Chris Lattner, the creator of LLVM [LA04]. In fact, MLIR is strongly influenced by LLVM, but also seeks to avoid certain shortcomings of LLVM. In the following, we provide some necessary background on MLIR that will be required to understand the DAPHNE compiler's design.

Idea and motivation. The authors of MLIR observe that widely used IRs, such as the LLVM IR, use a single abstraction level. While this does have advantages, it has also led the developers of several programming languages to create their own higher-level IR and compilers on top of LLVM, e.g., Swift, Rust, Julia, and Fortran. This "reinvention of the wheel"-approach often leads to a lot of extra development time and lower-quality compiler systems. The goals of MLIR are thus to (1) make it easy to introduce new abstraction levels, (2) provide the infrastructure to solve common compiler engineering problems, and (3) to offer a reusable framework supporting the combination of different abstraction levels in one IR.

IR elements. The main unit of semantics in MLIR is the *operation*, which can represent anything from an instruction over a loop or a function to a module. Operations consume and produce zero or more *values*, whereby the input values are called *operands* and the output values are called *results*. Values are maintained in SSA form and represent data at run-time. Each value is either the result of an operation or a block argument. Furthermore, each value has a *type*. To express complex programs, operations can be nested in MLIR. An operation can have zero or more *regions*. A region contains one or more *blocks*, which are the units of complex control flow. A block is a linear sequence of operations, concluded by a *terminator* operation that specifies how to select the next block for execution. This nesting paradigm can be applied to express, e.g., loops, functions, or any domain-specific nesting of code. Operations can carry additional compile-time information in the form of typed *attributes*. Finally, for extensibility purposes, MLIR introduces the concept of *dialects*. A dialect provides a namespace for logically related operations, types, attributes etc.

Table 1: Overview of a few MLIR dialects that are used in the DAPHNE compiler.

Dialect	Summary	Example operations
`scf`	Structured control flow	`if, while, for, yield`
`affine`	Affine loops	`for, if, load, store`
`arith`	Arithmetic operations	`addi, addf, muli, mulf`
`math`	Complex mathematical operations	`sin, atan, log, floor`
`memref`	Memory access operations	`alloc, realloc, load, store`
`llvm`	Almost 1:1 mapping to LLVM IR	`add, fadd, call, br`

Existing MLIR dialects. Off-the-shelf, MLIR ships with a multitude of existing dialects[1] for various purposes and at various abstraction levels from domain-specific operations for linear algebra down to operations for dedicated kinds of hardware accelerators. Table 1 gives an overview of some of the dialects that play a noteworthy role in the DAPHNE compiler.

Compilation chain. Being a framework for building compilers, MLIR provides the infrastructure to create and apply reusable *compiler passes* to rewrite the IR. Compilers based on MLIR can define their own modular pass pipelines that can contain custom compiler passes, but also various commonly used passes given by MLIR, such as common subexpression elimination and loop-invariant code motion. To allow existing MLIR passes to interact with custom operations with unknown semantics, *traits* and *interfaces* can be attached to new operations. Traits express that an operation has a certain property (e.g., that it has no side effects), and can be checked and reacted to by compiler passes. Interfaces must be implemented by the operation and provide custom logic that is invoked by compiler passes (e.g., operations can provide logic to fold them if the operands are compile-time constants).

MLIR ecosystem. When the DAPHNE project started, MLIR was still a relatively young project. Meanwhile, MLIR has grown significantly and received adoption in academia and industry by many domain-specific compilers and systems [AB+16, PG+19, JKG22]. There is a vivid developer community around the framework[2]. This community, in combination with the high-quality compiler infrastructure MLIR provides, makes it a solid foundation for the DAPHNE compiler.

3 DAPHNE Compiler Design

In this section, we describe the design of the DAPHNE compiler in detail. We start with a high-level overview of the compiler (Section 3.1). After that, we present individual components in more detail: the intermediate representation DaphneIR (Section 3.2), the compilation chain (Section 3.3), and the aspects of configurability and extensibility (Section 3.4).

3.1 Overview

Integration into the DAPHNE system architecture. The DAPHNE compiler is situated between the user-facing language abstractions and the DAPHNE runtime. The design of DAPHNE's language abstractions DaphneDSL and DaphneLib has been described in detail in a previous deliverable [D3.1], just like the DAPHNE runtime [D4.1, D4.3]. The DAPHNE compiler plays a central role in the overall system architecture since it connects the other components and since most features of the system require at least some degree of compiler support.

High-level overview. Figure 1 shows a high-level overview of the DAPHNE compiler in the context of the remaining system components, the important internal components of the compiler itself, as well as their interactions with each other. The DaphneDSL program specifying the user's IDA pipeline is first processed by an ANTLR-based parser, which outputs an initial,

[1] https://mlir.llvm.org/docs/Dialects/
[2] https://discourse.llvm.org/c/mlir/31

unoptimized DaphneIR representation of the program. DaphneIR is DAPHNE's intermediate representation and is a domain-specific MLIR dialect. The task of the DAPHNE compiler is twofold: On the one hand, it simplifies and optimizes the IR in numerous ways to improve the runtime behavior with respect to a certain optimization goal, such as runtime performance, (peak) memory footprint, compiled binary size, or a combination thereof (multi-objective optimization). On the other hand, it lowers the IR from high-level domain-specific operations to low-level LLVM IR. The LLVM IR representation of the program is finally compiled just-in-time (JIT) to executable code, whereby pre-compiled DAPHNE runtime libraries are linked. The output of the DAPHNE compiler is an executable code artifact including control flow, scalar operations, and calls into DAPHNE runtime components, such as device kernels [D7.1], the vectorized engine [D2.1, D2.2], and routines for I/O and memory management.

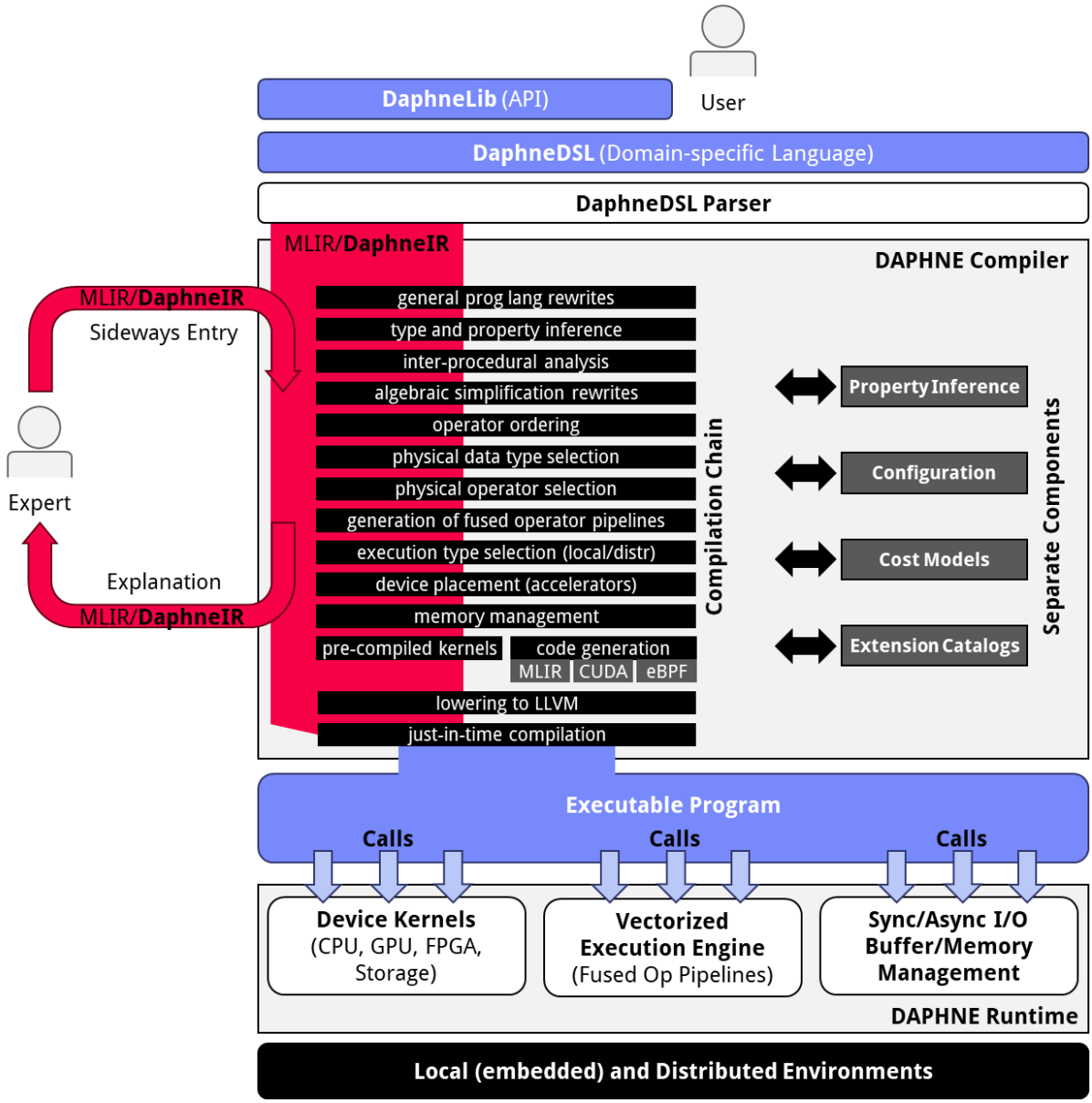


Figure 1: Overview of the DAPHNE compiler in the context of the system architecture.

Compilation chain overview. Internally, the DAPHNE compiler consists of a compilation chain comprising various compiler passes as well as separate compiler components that are used by these passes and other components of the system. Each compiler pass is either a custom DAPHNE pass or a standard MLIR pass. DAPHNE passes may utilize components from MLIR, and MLIR passes may call into custom DAPHNE compiler code made available through standard MLIR interfaces. Furthermore, expert users may inject IR into the compilation chain and extract the IR at any point. In the following, we describe the design of major components of the DAPHNE compiler in more detail.

3.2 Intermediate Representation DaphneIR

DaphneIR dialect. Being based on MLIR as a framework for domain-specific compilers, the DAPHNE compiler uses MLIR as its intermediate representation. More specifically, DAPHNE defines its own new MLIR dialect, DaphneIR. There are several reasons why we decided to create a new MLIR dialect rather than solely building upon existing ones: First, targeting *integrated* data analysis pipelines, we need to ensure seamless interoperability between matrix operations from linear algebra and frame operations from relational algebra. To this end, we need to be able to represent both at compatible abstraction levels. Second, creating our own dialect guarantees a required level of independence from the evolution of high-level standard MLIR dialects, such as `linalg`, which is especially important since MLIR was still a relatively young project when we started using it in late 2020.

Like every MLIR dialect, DaphneIR consists of custom types, operations, traits, and interfaces. All of these are defined declaratively in TableGen[3], a compact text-based representation from the LLVM ecosystem allowing the generation of C++ source code in the MLIR/LLVM compiler frameworks.

DaphneIR types. Table 2 provides an overview of all types used in DaphneIR. DaphneIR defines data types and value types, which are essentially a 1:1 mapping to the types supported by DaphneDSL [D3.1]. In terms of data types, DaphneIR supports `Matrix` (parameterized with a value type) and `Frame` (parameterized with a value type per column). Regarding value types, DaphneIR reuses signed and unsigned integer types and floating-point types of different bit widths from MLIR. Besides that, DaphneIR defines its own String type, since MLIR does not offer that off-the-shelf. Furthermore, DaphneIR explicitly supports an `Unknown` type, which is used to express that the type of some SSA value is not known (yet) at the current stage of the compilation chain. This `Unknown` type can be used as both a data type and a value type. All DaphneIR operations allow inputs of the `Unknown` type, and all DaphneIR operations supporting more than one result type allow the `Unknown` type for their result. Additionally, at the lower abstraction levels, DaphneIR also offers some more technical types for special purposes, such as `DaphneContext` and `VariadicPack`.

[3] https://llvm.org/docs/TableGen/index.html

Table 2: Overview of types used in DaphneIR.

Data types	
`Matrix`	A matrix with a common value type for all elements
`Frame`	A frame with an individual value type per column
Value types	
`UI8, UI32, UI64, SI8, SI32, SI64`	Unsigned and signed integer types of various bit widths
`F32, F64`	Floating-point types of various bit widths
`I1`	Boolean type
`String`	Character string type
Unknown type	
`Unknown`	A hitherto unknown data/value type
Technical types	
`DaphneContext`	Runtime context passed to all kernels
`VariadicPack`	Auxiliary type for variadic kernel arguments

Operations. Table 3 gives an overview of the classes of operations in DaphneIR, along with some example operations. As an entry point from the DaphneDSL parser, DaphneIR defines several (classes of) operations which map 1:1 to DaphneDSL built-in functions and DaphneDSL operators. For instance, the DaphneDSL built-in function `sqrt()`, which calculates the elementwise square root of a matrix, corresponds to the DaphneIR operation `daphne.ewSqrt`; and the DaphneDSL operator `==`, which calculates the elementwise equality of two matrices, corresponds to the DaphneIR operation `daphne.ewEq`. Furthermore, DaphneIR contains (classes of) operations which do not correspond directly to any DaphneDSL construct, but are only created by the DAPHNE compiler during the optimization and lowering of the IR. As an example, for memory management purposes, DAPHNE uses reference counting to ensure that each allocated data object is freed exactly once. This is achieved in collaboration between the compiler and runtime, whereby the compiler inserts `daphne.incRef` and `daphne.decRef` operations to increase and decrease the reference counter at the right points in the IR.

Traits and interfaces. DaphneIR defines its own traits and interfaces on operations to interact with DAPHNE compiler passes. DAPHNE extensively uses traits and interfaces for various purposes, most importantly for: (1) type and property inference, (2) pipeline fusion, (3) heterogeneous hardware support, and (4) compiler hints by the user. More details can be found with the respective passes of the compilation chain in Section 3.3.

Table 3: Overview of the classes of operations in DaphneIR, along with example operations.

Class of operations	Example operations (mnemonics)
1:1 mapping to DaphneDSL	
Constants	`constant, matrixConstant`
Data generation	`fill, seq, rand, ...`
Properties	`numRows, isSymmetric, ...`
Matrix multiplication	`matmul`
Elementwise unary/binary	`ewSqrt, ewAdd, ewEq, ...`
Generalized outer product	`outerAdd, outerEq, ...`
Aggregation (full, row-wise, column-wise, cumulative, grouped)	`allAggSum, rowAggMean, colAggMin, cumAggProd, grpAggMax, ...`
Left/right indexing	`insertRow, extractRow, sliceCol, ...`
Reorganization	`transpose, reshape, order, ...`
Matrix decompositions	`eigen, svd, ...`
Deep neural network	`conv2dForward, biasAddForward, avgPoolForward, ...`
Miscellaneous matrix operations	`replace, upperTri, solve, ...`
SQL support	`sql, registerFrame`
Extended relational algebra	`filterRow, thetaJoin, group, intersect, ...`
Conversions	`cast, copy, quantize, ...`
Preprocessing	`oneHot, ...`
Input/output	`print, readMatrix, readFrame, write, ...`
2nd order	`map, paramserv, ...`
Control flow	`conditional, genericCall`
Not available in DaphneDSL	
Parallelism	`vectorizedPipeline, distributedPipeline`
Context	`createDaphneContext, destroyDaphneContext`
Memory management	`incRef, decRef`
Profiling	`startProfiling, stopProfiling`
Kernel call	`callKernel`

Integration with existing MLIR dialects. DaphneIR seamlessly integrates with and relies on several existing MLIR dialects. Regarding *types*, as stated above, DaphneIR reuses several scalar types from MLIR. In terms of *operations*, DaphneIR is complemented by MLIR's dialect for structural control flow (SCF) to add support for if-then-else constructs and loops. Furthermore, as one alternative compilation route through the DAPHNE compiler, the lowering of DaphneIR's domain-specific operations to lower-level MLIR dialects like `affine`, `arith`, and

`memref` is supported. Concerning *traits and interfaces*, DaphneIR operations possess certain standard MLIR traits and implement certain MLIR interfaces to allow processing by standard MLIR compiler passes. For instance, most DaphneIR operations have the `Pure` trait, which tells MLIR's pass for common sub-expression elimination (CSE) that two instances of this operation with the same operands can safely be deduplicated by removing one instance. Furthermore, many DaphneIR operations implement MLIR's canonicalization interface, whose code is then called by MLIR's canonicalizer pass. As a final remark, we could have split DaphneIR into multiple dialects at different abstraction levels. However, we consciously decided against that since it would have complicated the IR design with no clear benefit if these dialects are used only within DAPHNE.

3.3 Compilation Chain

In this section, we give an overview of the DAPHNE compiler's compilation chain, focusing on the most important compiler features. We present the steps in the order they are applied (whereby some general programming language rewrites are applied at multiple points in the compilation chain).

3.3.1 General Programming Language Rewrites

Motivation. Like most compilers, the DAPHNE compiler applies various well-known general programming language rewrites that are useful at all levels of abstraction. Examples include common sub-expression elimination (CSE), dead code elimination (DCE), constant folding, constant propagation, branch removal, code motion, loop unrolling, and function inlining. These rewrites can simplify the structure of the IR and thereby reduce the amount of work for subsequent compiler passes and enable additional optimization opportunities.

Reusing MLIR. One of the reasons why we decided to base the DAPHNE compiler on MLIR is that MLIR already provides passes for these general programming language rewrites. The DAPHNE compiler employs these passes at several points in the compilation chain to tidy up the IR. To allow the existing MLIR passes to handle DaphneIR operations, we attach existing MLIR traits and interfaces to DaphneIR operations, which is the intended way of informing MLIR passes of the characteristics of custom operations. For instance, most DaphneIR operations have the `Pure` trait (formerly `NoSideEffects`), which allows MLIR's CSE/DCE pass to remove repeated operations on the same operands. Furthermore, many DaphneIR operations implement the `fold()` interface, which evaluates operations at compile-time, if their operands are compile-time constants. MLIR applies several general programming language rewrites during its canonicalization pass, such as constant folding and branch removal. Other rewrites are performed by separate passes, such as loop-invariant code motion and function inlining.

3.3.2 Type and Property Inference

Motivation. All DAPHNE data objects (matrices and frames) have a data type and a value type as well as general and data type-specific interesting (data) properties. Knowing the data/value type of an object at compile-time is crucial for dispatching to the right runtime kernel. Furthermore, interesting properties are the basis for various compiler and runtime decisions. The DAPHNE compiler exploits interesting properties for, e.g., algebraic rewrites, operator ordering, and physical operator selection. The DAPHNE runtime can exploit interesting

properties for selecting dedicated code paths in pre-compiled kernels. For this purpose, compile-time information on interesting properties can be passed on to runtime kernels.

Interesting properties. The DAPHNE compiler supports a rich and extensible set of interesting properties. These properties can range from simple boolean properties (such as `isSymmetric` or `hasNaN`) over scalar numeric properties (such as `numRows`/`numCols` (shape), `sparsity`, `min`/`max`) to complex structured properties (such as value histograms for cardinality estimation or sketches for sparsity estimation). Value types and interesting properties are represented as *parameters* of DaphneIR's MLIR types `Matrix` and `Frame`, i.e., properties are technically attached to the types of SSA values. To enable reuse in the DAPHNE runtime, all interesting properties are bundled in a custom structure, whereby `Matrix` and `Frame` have a single parameter of that structure type.

Inference pass. DAPHNE's compilation chain includes an inference pass, which is responsible for all type and property inference. Individual instances of the pass can be configured with the subset of properties to infer, since the inference of certain properties (e.g., number of rows and sparsity) can be computationally expensive. The inference pass performs a single pass over the entire IR in each MLIR function. For each operation, it infers all requested but hitherto unknown properties of the results based on the properties of the operands (and, for certain operations, the operand values themselves). Combining the inference of multiple or all properties in a single pass is important since a particular property of an operation's result may depend on different properties of the operands. Furthermore, the inference pass is interleaved with the MLIR canonicalizer pass, which is also responsible for constant folding, since certain properties of certain operations' results depend on the values of the operand (e.g., the minimum and maximum values in the result matrix of the `daphne.rand` operation depend on the values of the `min` and `max` operands of that operation), while the result values of certain operations depend on their operands' properties (e.g., the value of `daphne.numRows` on a matrix *X* is the inferred number of rows of *X*).

The inference pass interacts with DaphneIR operations via custom traits and interfaces, which are defined in the inference component of the DAPHNE compiler. Traits are used for frequently occurring cases, such as `ValueTypeFromFirstArg` or `ValueTypeFromArg`. The program logic for interpreting these traits is situated in the compiler's inference component as well. Besides that, custom interfaces are used to cover more complex and non-reoccurring cases as well as all DaphneIR operations with multiple results. The inferred value of a property can be *unknown* after the inference in case of insufficient information on the operation's operands.

Handling unknowns. The data/value type as well as each interesting property can be *unknown* (at a particular point) at compile-time. For data/value types, DaphneIR defines the dedicated `Unknown` type (see Section 3.2). For interesting properties, either a particular value (e.g., `-1`) or a `nullptr` is used to indicate that the value of the property is unknown. At the beginning of the compilation chain, virtually all data/value types and properties of all SSA values are unknown. During type and property inference, most of these unknowns can be replaced by inferred or estimated values. Unknowns can also arise from conditional control flow. For instance, if a certain property of the *i*-th result of the then-branch and the else-branch of an `scf.if` operation differs, this property is set to unknown after the `scf.if`. Likewise, a property of a loop-carried variable changes inside a for/while loop's body, we reset the value to

unknown. In fact, data/value types and interesting properties may remain unknown at compile-time. However, knowing types and properties is important, e.g., when they are the basis for memory estimates that decide whether an operation can be executed locally or requires distributed computations. In this case, the DAPHNE compiler takes measures to exploit the information once it becomes available at run-time. In the simplest case, this can be achieved by generating alternative code paths at compile-time and choosing the one to take based on the actual type or property information at runtime. For the general cases, the DAPHNE compiler identifies and flags IR regions for *dynamic recompilation* [BB+14], i.e., it generates calls back into the compilation chain, which are triggered at run-time and allow the recompilation and reoptimization of these code regions based on actual type and property information.

3.3.3 Inter-procedural Analysis

Up to here, type and property inference took place only intra-procedurally, i.e., within a single function. However, complex IDA pipelines are typically assembled out of numerous user-defined functions to handle the code complexity. Calls to user-defined functions constitute natural boundaries to type and property inference. To pass information from the call sites to the function definitions, the DAPHNE compiler performs inter-procedural analyses of the IR by analyzing the function call graph. After determining the types and properties of the arguments to a function call, it creates a new instantiation of the entire user-defined function, whereby information on types, interesting properties, and constant values is made available inside the newly created variant of the function. This extra information allows for additional optimizations inside the function body, which are done by a new round of intra-procedural type and property inference. At the current state of the implementation, we specialize all encountered variants of a function. Ultimately, we are going to apply this instantiation of functions in a conscious way, to balance the potential benefits and costs, e.g., by restricting the code size or checking if decisive optimizations are enabled by a specialization.

3.3.4 Algebraic Simplification Rewrites and Operator Ordering

Motivation. While the initial DaphneIR program already specifies what to calculate, it may not be an efficient way to do that. Both linear and relational algebra offer numerous algebraic equivalences, i.e., pairs of different algebraic expressions yielding the same result. Two equivalent expressions may have vastly different computational costs or memory requirements. Algebraic equivalences are normally exploited by domain-specific compilers to optimize the program [BB+14]. Typical optimization goals include reducing the number of operations and intermediate results, reducing the size of intermediates and the computational effort for the operations on them, and improving the asymptotic complexity of the expression. The DAPHNE compiler employs various kinds of such optimization rewrites, which we elaborate on next.

Table 4: Examples of static simplification rewrites.

X / 1 → X	trace(XY) → sum(X * t(Y))
X * 1 → X	sum(X + Y) → sum(X) + sum(Y)
X – 0 → X	sum(t(X)) → sum(X)
X + X → 2 * X	sum(alpha * X) → alpha * sum(X)

Table 5: Examples of dynamic simplification rewrites.

t(X) → X, iff X is symmetric	rowSums(X) → X, iff ncol(X) = 1

Algebraic simplification rewrites. The DAPHNE compiler applies an extensive and easily extensible set of static and dynamic simplification rewrites [BB+14]. *Static rewrites* exploit algebraic equivalences that always hold without any conditions on the operands' properties and are assumed to be always beneficial. Some examples can be found in Table 4. Dynamic simplification rewrites can only be applied (or are only beneficial) when the operands of the expression to rewrite meet certain conditions in terms of their interesting properties, especially their shape. A few examples are given in Table 5. MLIR, as a framework for domain-specific compilers, supports the definition of custom rewrite patterns that match certain sub-DAGs of operations, optionally check a condition, and replace the original sub-DAG with a new, rewritten one. These rewrite patterns can be applied with a custom compiler pass or as a part of MLIR's canonicalizer pass. Since the latter is generally used to unify and simplify the IR, we integrate simple, inexpensive simplification rewrites as canonicalizations in the DAPHNE compiler and apply them at multiple points in the compilation chain. This is important, since new opportunities to apply these rewrites can arise from changes to the IR in other passes.

Operator ordering. As a special kind of algebraic rewrites, the DAPHNE compiler reorders sequences of operations. Two of the most decisive examples are *matrix multiplication chain optimization* from linear algebra and *join ordering* from relational algebra. Both rewrites exploit that the respective operation (matrix multiplication and join) is associative, i.e., the parenthesization, and thereby the order of evaluation, can be chosen arbitrarily. However, different orders can result in vastly different sizes of intermediate results and computational effort for the operations. The goal of typical operator ordering is, thus, to minimize the size of intermediates, and there are several approaches for doing so for both matrix multiplication chain optimization and joins. Besides operator ordering for these specific operations, the DAPHNE compiler aims to find a good data-flow graph linearization, in general. To this end, independent operations in the IR can be reordered to minimize, e.g., the time for which intermediates need to be kept, to reduce the peak memory footprint. All these operator orderings are integrated into the DAPHNE compiler as separate compiler passes as they typically incur a higher cost.

3.3.5 Physical data type selection

Motivation. One of the design principles of DaphneDSL is data independence [D3.1], i.e., users work with abstract matrix and frame data types, while the physical data representation (such as dense or a particular sparse format) is chosen automatically by the system. This independence is reflected in DaphneIR by defining a logical `Matrix` and `Frame` type, as mentioned above. Automatically selecting the physical data type is important, since the optimal data representation depends on the data properties, which are hard to oversee manually.

Physical data type selection. Off-the-shelf, DAPHNE supports multiple physical data types, such as a dense matrix and a compressed sparse row (CSR) matrix, and offers extension hooks for adding new physical data types. The compilation chain contains a pass that selects a suitable

physical data type for each matrix or frame-typed intermediate result. This decision is based primarily on *memory estimates*, which in turn depend on the interesting data properties. For instance, sparsity is a key property estimated by the DAPHNE compiler and can be used to decide if a matrix should be represented in a dense or a sparse format. Since physical data types added through extensibility mechanisms could be domain-specific and exploit any data property (e.g., `isSymmetric`), dedicated size estimation functions can be registered in the extension catalog along with a physical data representation. The DAPHNE compiler evaluates those and selects a representation with low or low enough memory requirement. Besides the data properties, the access patterns of the operations processing the data objects are also considered. For instance, a CSR representation of a matrix supports the inexpensive extraction of row segments, while compressed sparse column (CSC) better supports column segments. These characteristics can be exploited in the context of DAPHNE's vectorized pipelines described below. The decision is stored in form of an MLIR type parameter attached to the `Matrix` or `Frame` type instance and is respected by subsequent passes.

3.3.6 Physical operator selection

Motivation. The domain-specific operations in DaphneIR primarily express what result shall be calculated at the logical level, but not how to calculate the result in detail at the physical level. In fact, many operations from linear and relational algebra can be computed by different algorithms, i.e., physical operators. A physical operator may not be applicable in all cases but yield an improved runtime behavior in the cases it supports. For instance, the relational join operator can be computed by a nested-loops join (applicable on all data and join predicates), a hash-join (applicable on all data, but only equivalence predicates), and a sort-merge join (applicable only on sorted data, and equivalence predicates). Likewise, there are different physical operators for matrix multiplications, such as a general matrix-matrix multiplication, a symmetric rank-*k* operation, and various forms of matrix-vector multiplications, most of them with special cases for transposed inputs matrices.

Physical operator selection. Physical operators are represented as dedicated DaphneIR operations, and the DAPHNE compiler has a pass that rewrites a DaphneIR operation to another one representing a physical operator. The decision on the rewrite is based on interesting properties and parameters of the operation. Besides that, runtime kernels may also select different code paths based on the same information. Making the decision at compile-time has the advantage that the compiler can, e.g., calculate costs more accurately when it knows the physical algorithm, while making the decision at run-time enables the access to the actual property information, which may not be available at compile-time. Thus, DAPHNE combines both approaches.

3.3.7 Generation of Fused Operator Pipelines

Motivation. Basic runtime plans of kernels with materialized intermediates offer good performance and simplify debugging. Although this model is commonly used in existing ML systems [AB+16, PG+19, BA+20] and column stores [BK99], it suffers from several limitations, such as large temporary memory and memory-bandwidth requirements through the materialization of intermediate results, too fine-grained synchronization barriers per operator through multi-threaded kernels, and a too coarse-grained device placement of entire

operators. To address these limitations, DAPHNE is equipped with a vectorized execution engine for compiled operator pipelines, which is at the same time the central means for parallelism in both the local (e.g., multi-threading and multi-device) and distributed (e.g., multiple worker nodes) runtime. We have already described the overall design of the vectorized engine in a previous deliverable [D2.2] and, thus, focus primarily on its compiler aspects here.

Vectorizable operations. DaphneIR defines the custom `Vectorizable` MLIR interface, which indicates that an operation can be processed by splitting the inputs into segments, processing segments independently, and combining the results in the end. DAPHNE supports either splitting an operand into row or column segments or broadcasting an operand. Individual results can be combined by concatenating rows or column segments or by aggregation. To mention some examples, (1) for elementwise unary operations, the input matrix can be split into row segments, the operation can be performed on each row segment individually, and the resulting row segments can be concatenated to obtain the result; (2) for column-wise summation, the input matrix can be split into row segments, the operation can be performed on each row segment individually resulting in a single row per segment, the individual results can be combined by elementwise summation.

Pipeline fusion. The compilation chain contains a dedicated pass for fusing vectorizable operations into pipelines. Initially, each vectorizable operation constitutes a separate pipeline. By that, the operation can already be executed in parallel. However, to unfold the full potential, we create longer pipelines. The pass identifies producer-consumer relations between existing pipelines, i.e., cases where the result of the last operation *A* in one pipeline serves as an operand to the first operation *B* in another pipeline. If the result of *A* is combined along the same axis as the operand of *B* is split, the two pipelines can be stitched together. That way, we avoid the full materialization of the intermediate result between operations *A* and *B* and enable a cache-conscious processing. Note that we use the definition-use chain for this analysis. However, since the operations in a block always have a particular order in MLIR, we need to move independent operations between *A* and *B* before or after the pipeline. Finally, a `daphne.vectorizedPipeline` operation is created and all operations in the pipeline are moved into a nested region inside this operation. To the outside, the vectorized pipeline operation reveals all inputs and outputs of the entire pipeline, which can be arbitrarily many. The split mode for each input as well as the combine mode for each result are captured as MLIR attributes of the `daphne.vectorizedPipeline` operation. At this stage of the compilation chain, pipelines are merely created, while the lowering takes place in later compiler passes. Pipelines executed locally are ultimately lowered to a runtime call into the vectorized engine, whereby a pointer to a function containing the compiled pipeline body is passed.

3.3.8 Execution Type Selection and Device Placement

Motivation. Besides the local multi-threaded execution that is applied by default in DAPHNE's vectorized pipelines, we also want to employ parallelism by distributing computations over multiple nodes in the distributed runtime and by offloading work to heterogeneous hardware accelerators. While most of the complexity of that relates to WP4 and WP7, the DAPHNE compiler provides the infrastructure for making fundamental decisions in that context.

Distributed execution. The compilation chain contains a custom pass that decides, for each fused pipeline, if it shall be executed in the distributed runtime. This decision is based on (a) the interesting properties of the pipeline inputs, intermediates, and outputs, especially their data sizes, (b) the way the inputs need to the split and the outputs need to be combined, (c) the operations in the pipeline, (d) the availability of distributed kernel variants, and (e) the available distributed nodes. In general, we distribute a pipeline if the data involved is too large for purely local execution. While the details of the decision are developed in cooperation with WP4, the DAPHNE compiler provides all necessary infrastructure for easily integrating complex decision-making strategies. When a distributed execution is chosen, the `daphne.vectorizedPipeline` operation is replaced by a `daphne.distributedPipeline` operation with the same operands/results and pipeline body. This rewrite is done since pipelines must be lowered in a slightly different way for distributed execution. Passing a pointer to the MLIR function created from the pipeline body to the runtime is not helpful, since the distributed nodes cannot access this function on the coordinator. Instead, the textual representation of the IR is given to the runtime when calling the `distributedPipeline` kernel to allow the transmission of the code to each worker where it is parsed and locally optimized.

Hardware accelerators and computational storage. Likewise, the compilation chain entails a pass that decides, for each fused pipeline, if it shall be executed on a particular hardware accelerator, e.g., GPU or FPGA. This decision is based on (a) the interesting properties of the pipeline inputs, intermediates, and outputs, (b) the way the inputs need to be split and the outputs need to be combined, (c) the operations in the pipeline, and (d) the available accelerator devices including their characteristics, especially the capacity of the device memory. In general, we offload a pipeline if the involved data is neither too small (dominant offloading overhead) nor too large (cannot be processed on the device) and all necessary operations are supported on the target device. While the details of the decision are developed in cooperation with WP7, the DAPHNE compiler provides all necessary infrastructure for easily integrating complex decision-making strategies. When offloading to a device is chosen, the compiler adds an MLIR attribute to the pipeline as a hint for later lowering passes. The decision for a hardware accelerator is not exclusive, i.e., one pipeline can be executed on multiple heterogeneous accelerators in parallel, at the granularity of tasks in the vectorized engine [D2.2]. In later lowering steps, one MLIR function is generated per accelerator and all function pointers are passed to the `vectorizedPipeline` runtime kernel.

3.3.9 Memory Management

Motivation. DAPHNE manages memory automatically, such that DaphneDSL users do not need to think about it manually. Memory management is generally accomplished by a collaboration between the DAPHNE compiler and the DAPHNE runtime, whereby the compiler is required for its global view on the program, and the runtime for its ability to cover dynamic aspects, which are not predictable at compile-time. In the following, we highlight two important features related to memory management in DAPHNE.

Garbage collection. DAPHNE data objects (matrices and frames) are created by runtime kernels. The task of garbage collection is to ensure that every data object and its underlying data buffers are freed exactly once, to avoid memory leaks and double frees. To this end, DAPHNE employs reference counters at the level of both data objects and data buffers. Initially,

the reference counter of a newly created data object or data buffer is one. The DAPHNE runtime offers the `decRef` and `incRef` kernels to manipulate an object's reference counter. The `decRef` kernel decreases the reference counter by one; if the counter becomes zero, the data object is destroyed, which triggers decreasing the underlying data buffers' reference counters as well, leading to their deallocation unless they are shared with other data objects. The `incRef` kernel simply increases an object's reference counter by one. The DAPHNE compilation chain contains a dedicated compiler pass that inserts special operations that are later lowered to calls to these kernels at the right points. This pass finds all SSA values by iterating over all blocks of each function in the IR, processing all block arguments, walking the operations in each block, and processing each result value of an operation.

- *Ensure free at least once:* For each SSA value to process, the compiler traverses the definition-use chain to find the last user of the value in the current block and inserts a `daphne.decRef` operation directly after that operation. For unused SSA values, the `daphne.decRef` operation is inserted directly after the value's defining operation for operation results and directly in the beginning of the block for block arguments, respectively. The only exceptions are SSA values that are used as operands to block terminators (such as `daphne.return` at the end of a function, or `scf.yield` at the end of a loop's body), because these values are passed on to another block and keep existing there.
- *Ensure free at most once:* Additionally, each time an SSA value is passed as an operand to an operation that contains a region where the value effectively becomes a block argument, the compiler inserts a `daphne.incRef` operation right before the use of the value. That way, it compensates for the `daphne.decRef` operation that will be created for the block argument inside the nested block. For instance, a `daphne.incRef` operation is needed before a value is passed to a user-defined function or to a loop operation.

In combination, these two strategies ensure that each data object is freed exactly once.

Update-in-place optimization and reuse of allocations. By default, all runtime kernels create a fresh data object (matrix or frame) for each of their results. The only exceptions are kernels that can directly reuse their input data object, e.g., sorting an already sorted matrix. For more efficient memory usage, DAPHNE supports an update-in-place optimization that allows a kernel to reuse one of its input data objects for its result if certain conditions are met. The benefits of that are (1) saving the allocation of a new object, and (2) improving the cache behavior, since load and store operation will access the same memory locations, thereby increasing the hit rates in the CPU cache and the translation look-aside buffer. In the simplest case, the following must hold: (a) the input and output objects have the same logical and physical data type, value type, and shape, (b) the input object is not used again later in the IR, (c) the input's underlying data buffers are not shared with any other data objects that are still in use, and (d) the kernel's access pattern ensures that individual elements of the input data are overwritten only when they are not needed anymore for the calculation. This is, for instance, the case for elementwise unary and binary operations. While criteria (a) and (b) can be determined at compile-time (assuming that all required types and properties could be inferred), criteria (c) and (d) can only be determined at run-time. Thus, for each DaphneIR operation supporting in-place updates, the DAPHNE compiler checks, for each operand, if it is

still used afterwards and attaches this information as MLIR attributes to the operation. These attributes later become parameters to the runtime kernels. Then, it is the kernels' task to evaluate the remaining criteria and to overwrite the input if applicable.

3.3.10 Pre-compiled Kernels and Code Generation

Motivation. So far, the compilation chain has worked on DaphneIR's domain-specific operations. As we are approaching the end of the compilation chain, these operations must be lowered to some executable representation. To this end, the DAPHNE compiler offers multiple alternative routes. By default, each DaphneIR operation is lowered to a special operation called `daphne.callKernel`, which represents a call to a pre-compiled kernel function. Alternatively, for most operations, the DAPHNE compiler can generate code on-the-fly, whereby multiple code generation back ends are supported. In fact, these different routes can be taken for each operation individually, whereby it is especially meaningful to treat all operations within a fused pipeline the same way. In the following, we present these approaches in more detail.

Pre-compiled kernels. Lowering DaphneIR operations to pre-compiled kernels is the default behavior, in contrast to other MLIR-based systems []. Kernels are hand-written C++ functions that perform coarse-grained operations, i.e., they process entire matrices or frames (or views thereof within a vectorized pipeline). The advantages of this approach are manifold: (1) implementing kernels in a high-level language like C++ is much more convenient than implementing a code generation back end, for most developers, (2) all kinds of hardware accelerators typically offer some kind of C/C++ interface, which can directly be used in a C/C++ kernel, (3) the effort for low-level kernel code optimization and compilation is spent just once in advance. Writing kernels in C/C++ offers a high degree of flexibility. That way, kernels can target a multitude of hardware devices, e.g., CPUs (scalar and data-parallel SIMD processing), GPUs, and FPGAs; a multitude of hardware abstractions, e.g., CUDA [SK10], TVL [UP+20], and oneAPI [R+21]; and various existing libraries of highly optimized device kernels, e.g., BLAS and LAPACK. Pre-compiled kernels are also used for calls into the DAPHNE runtime in general.

Mapping operations to pre-compiled kernels. The DAPHNE compiler is informed of the available pre-compiled kernels via the *kernel extension catalog*. This catalog contains essential information on each kernel including the DaphneIR operation it belongs to, the combination of input/output data/value types it is specialized for (most standard kernels are implemented in a type-generic way using C++ templates, but concrete instantiations of these templates are pre-compiled), the hardware back end it targets, the shared library file containing the kernel, and (optionally) cost models. This catalog is populated from configuration entries at system start-up or at DaphneDSL compile-time from dedicated DaphneDSL commands. DAPHNE's compilation chain contains a dedicated pass that rewrites DaphneIR operations to the `daphne.callKernel` operation. For each DaphneIR operation, the compiler searches the kernel extension catalog for corresponding registered kernels. These kernels are filtered by the hardware back end. Then, the compiler selects the kernel that best matches the input/output data/value types of the operation. Ideally, there is a pre-compiled kernel for this combination. However, due to the exponential number of variants, it is infeasible to pre-compile all combinations of data/value types for all inputs/outputs of an operation (in terms of both pre-compilation time and kernels library footprint). If there is no ideal match, the compiler chooses the kernel that incurs the least cost for cast operations on the operands and results. As the

casts shall not lead to a loss of precision, we generally pre-compile all kernels for the largest floating-point and integer types, i.e., `f64` and `si64`, for all operands and results. Additional specializations for other type combinations can be added as desired. During the lowering to LLVM, each `daphne.callKernel` operation becomes an LLVM function call, and the required shared libraries containing the kernels are linked during JIT-compilation.

Code generation. As an alternative to pre-compiled kernels, the DAPHNE compiler also supports on-the-fly code generation for individual DaphneIR operations and entire fused pipelines. For this purpose, multiple code generation back ends can be used and easily added to the compiler. In the following, we describe the three code generation back ends we are currently supporting or aiming for.

MLIR code generation. As the DAPHNE compiler is based on MLIR, it is only natural to support MLIR-based generation of low-level code for individual DaphneIR operations. Via MLIR code generation, we can target CPUs, or any hardware device supported by the MLIR/LLVM compiler stacks (e.g., GPUs via MLIR dialects like `gpu`, `nvvm`, and `spirv`).

The basic idea consists in generating MLIR operations to express the semantics of the DaphneIR operations at a lower abstraction level (MLIR's *progressive lowering* principle). For instance, most DaphneIR operations result in a pair of loops from MLIR's `affine` dialect, that iterate over the elements of a matrix or frame, operations from MLIR's `arith` or `math` dialects to operate on individual elements, as well as operations from MLIR's `memref` dialect for loading and storing individual values from and to memory. While this resembles the default lowering approach in most MLIR-based systems, DAPHNE solves some special challenges to ensure the byte-compatibility of pre-compiled and generated kernels as well as the interoperability of generated kernels with DAPHNE's C++ data structures like `DenseMatrix`, `CSRMatrix`, and `Frame`. For this purpose, we introduce dedicated DaphneIR operations and corresponding runtime kernels for converting (in both directions) between, e.g., DAPHNE's `DenseMatrix` and MLIR's `StridedMemref` types, which are essentially meta data operations not copying any data.

The potential benefits of MLIR-based code generation are manifold, and mainly relate to instruction-level optimizations exploiting knowledge from the overall DaphneIR program, which is not available during the pre-compilation of isolated kernels. These benefits include (1) inlining UDFs in `map()` operations, where a pre-compiled `map` kernel can only call a function pointer, (2) the utilization of known constant operands for further optimizations, e.g., `x % y` can be rewritten to the more efficient `x & (y - 1)` if y is a power of two, (3) fine-grained operator fusion can be achieved through MLIR loop fusion on the affine loop generated for individual DaphneIR operations, where intermediates can be passed through CPU registers similar to code-generating DBMSs [N11], and (4) kernel code can be generated for any combination of input/output data/value types as actually required, whereby casts take place only at the value level.

CUDA code generation. Given DAPHNE's focus on heterogeneous hardware accelerators and sparsity exploitation, we also include a CUDA-based code generation back end targeting CUDA-compatible GPUs. Following the ideas of [BB+14], hand-written templates of CUDA C++ code with textual placeholders are applied to patterns of DaphneIR operations detected by the DAPHNE compiler. The resulting source code is compiled with CUDA's `nvrtc` run-time

compiler, which produces Nvidia's Parallel Thread Execution (PTX) IR, which is finally compiled by the Nvidia driver. The generated CUDA C++ code can use DAPHNE's runtime data structures directly. We primarily apply CUDA code generation for sparsity exploitation across chains of operations to avoid dense intermediates. More details on CUDA code generation can be found in deliverable D7.3 [D7.3].

eBPF code generation. To enable offloading the data processing to computational storage devices such as the Daisy board [D6.3], we are also planning to support the generation of eBPF bytecode on the host, which is transferred to and executed by the device. eBPF code can be interpreted or JIT-compiled. That way, sequences of operations can be pushed down to computational storage to reduce the data volume early on, e.g., through quantization, filtering, or input file parsing.

3.3.11 Lowering to LLVM and JIT-Compilation

Motivation. After the DAPHNE compiler has performed all custom and domain-specific optimizations on the IR, the final step of the compilation chain is to lower the IR to a dialect that can be compiled to executable code. We choose MLIR's `llvm` dialect for that purpose.

Lowering to LLVM. On the one hand, only a few distinct DaphneIR operations still exist after lowering to pre-compiled kernels and code generation. We apply custom rewrite patterns to lower these operations to `llvm` dialect operations. For instance, for each `daphne.callKernel` operation, we generate the declaration of the respective kernel function (to be linked with during JIT-compilation) and replace the operation by an `llvm.call` operation. Moreover, for the `daphne.vectorizedPipeline` and `daphne.map` operations, we need to retrieve a function pointer to the function(s) they must call at run-time, which we postpone until now since it can only be done at the LLVM abstraction level. On the other hand, we employ various existing MLIR conversions to progressively lower operations from existing MLIR dialects like `scf`, `affine`, `arith`, `math`, and `memref` to the `llvm` dialect. After that, the entire program is represented in MLIR's `llvm` dialect. Thus, it is now possible to generate actual LLVM IR and JIT-compile it to obtain the executable code, which is all done by existing MLIR and LLVM facilities.

3.4 Configurability and Extensibility

Configurability. The default DAPHNE compilation pipeline defines a reasonable set of features in a sensible order and is usable on all DaphneDSL scripts. Nevertheless, users may optionally modify the pass pipeline in certain ways. First, individual non-essential compiler features like individual optimizations can be turned on or off by means of command-line arguments and configuration files, while it is still guaranteed that a valid executable program is generated in the end. Second, the start and end points of the compilation chain can be modified, which is especially useful for testing and debugging purposes as well as for explanation and sideways entry described below. Third, expert users, developers, and researchers, may define a completely custom order of the existing compiler passes or even add their own compiler passes via means for extensibility.

Explanation. The DAPHNE compilation chain consists of numerous passes, many of which make complex decisions to rewrite the IR. Thus, understanding the IR at any stage of the pipeline is important for testing and debugging. Inspired by the SQL `EXPLAIN` command supported in most DBMSs, DAPHNE offers a `--explain` command-line argument that can be used to print the IR after selected passes. This feature has been used extensively in the demonstrator deliverable on the extended compiler prototype [D3.3]. It can be used for debugging and fine-tuning an IDA pipeline but is also employed for automated test cases of the DAPHNE code base, where we check the IR before and after the pass under test for certain conditions.

Sideways entry. DaphneDSL scripts (possibly generated internally by DaphneLib, DAPHNE's Python API) are the main entry point into the DAPHNE compilation chain. However, the DAPHNE compiler also offers means for sideways entry by accepting MLIR files, which contain a textual representation of the IR, as input and starting the compilation chain at a given point. This feature is intended for developers and researchers, especially in combination with the explanation feature. An IR fragment obtained via explanation can be manually changed and reinserted into the system to try the effect of certain rewrites. Besides that, a form of sideways entry is used in the worker nodes of the distributed runtime, which receive IR fragments from the coordinator node. These fragments are further optimized and compiled at the worker.

Compiler hints. The DAPHNE compiler facilitates user productivity by making many low-level decisions automatically, e.g., on the physical data types, kernels, and device placement. However, expert users may optionally use compiler hints in DaphneDSL [D2.2], e.g., to enforce a certain physical data representation, kernel, or device. These hints are materialized as MLIR attributes attached to the DaphneIR operations in the initial IR produced by the DaphneDSL parser. The DAPHNE compiler respects these hints as much as possible.

4 Related Work

Optimizing compilers in related systems. In the fields relevant to IDA pipelines, there are several systems that also have optimizing compilers. In terms of languages, systems, and libraries for numerical computations, Julia [BE+17] has its own domain-specific IR, while it relies on LLVM as a compilation back end, Dask [R15] is a drop-in replacement for numpy with lazy evaluation, which builds an operator DAG and optimizes it, and Mojo[4] is a recently introduced superset of the Python language, which has an MLIR-based compilation stack and focuses on efficiency through multi-threading and hardware accelerators. Regarding ML systems, SystemDS [BA+20] (formerly SystemML [GK+11]) has its own domain-specific compiler with high-level operators and low-level operators as two abstraction levels. The design of SystemDS had a strong influence on DAPHNE. TensorFlow [AB+16] and PyTorch [PG+19] support MLIR-based compilation back ends. TVM [CM+18] compiles ML models to heterogeneous hardware systems. In the area of database management systems (DBMSs), the optimization of declaratively specified SQL queries has been an active research field for decades [SA+79]. Similarities to compiler systems have been explored, e.g., in MonetDB with its SSA-based IR for

[4] https://www.modular.com/

columnar data processing [BK99]. Efficient data-centric code generation based on the LLVM compiler framework has first been investigated in HyPer [N11], which later also started supporting adaptive compilation by compiling long-running and interpreting short-running queries [KLN18]. Recently, query compilation for non-x86 architectures such as RISC architectures by ARM has gained attention [GB+23]. At same time, researchers have started looking for alternatives to LLVM to compile queries even faster [HD23b]. Like many of the systems mentioned above, the DAPHNE compiler also ultimately builds upon LLVM [LA04]. However, being focused on long-running analytical workloads, DAPHNE's requirements for top-speed compilation to machine code may not be as pressing as for transactional systems, even though dynamic recompilation in DAPHNE also demands high performance compilation. There are also systems whose optimizers can handle and optimize linear and relational algebra together [KK+19, JKG22], which is also a goal of DAPHNE.

Extensible optimizing compilers. The DAPHNE compiler can be extended by custom data and value types, kernels, and compiler passes. Extensibility has been a hot topic in database research in the 1980s and 90s [CH90]. For instance, the Volcano optimizer generator [GM93] could generate an entire DBMS optimizer from components specified succinctly by developers. Recently, the topic of extensibility gained new traction in research. User-defined operators [SN22] were presented as an approach to add new set-operators to an existing DBMS. The component-based DBMS mutable [HD23a] goes even a step further by making every component of the system interchangeable.

Individual techniques. The DAPHNE compiler adopts several techniques that have been presented in the literature before. For instance, *pipeline fusion* has been presented in the context of HyPer [N11] and *vectorized execution* has been pioneered by MonetDB/X100 [BKM08] and is still used in recent systems like DuckDB [RM19]. *Interesting properties* play an important role in the DAPHNE compiler. The Volcano optimizer generator [GM93] already allowed developers to define a custom set of interesting properties as an abstract data type (ADT). Sparso [RP+16] can exploit interesting properties of sparse matrices. Finally, there exist works on estimating sparsity in linear algebra programs [SB+19].

Extensibility of kernels, data types, and value types. As described in previous deliverables [D2.2, D3.1], DAPHNE (expert) users may extend DAPHNE by their custom kernels, data types, and value types [DB23]. The DAPHNE compiler is informed of these extensions through its extension catalogs. These catalogs store essential information on the custom kernels, data types, and value types, such as their name, shared library file, and more specific information and optional cost models. During physical data type selection, the DAPHNE compiler automatically selects data types from the catalog. Likewise, during the lowering of domain-specific DaphneIR operations to calls to pre-compiled kernels, these kernels are found in the catalog. In contrast to that, custom value types need to be applied consciously by the user, since they are a part of the IDA pipeline's semantics. However, in the future we could also investigate the automatic selection of custom value types, e.g., to explore the runtime-accuracy trade-off.

5 Conclusions

In this document, we have presented the design of DAPHNE's MLIR-based optimizing compiler. After a high-level overview of the compiler's role in the context of the overall DAPHNE system architecture, we have focused on the individual compiler components like the intermediate representation DaphneIR and various compiler features reflected in the compilation chain. Furthermore, we have commented on cross-cutting aspects of configurability and extensibility of the DAPHNE compiler. In the past three years since the project start, we have already implemented an initial and an extended prototype of the DAPHNE compiler, which have been demonstrated in earlier deliverables [D3.2, D3.3] and are part of the DAPHNE open-source repository[5]. Most of the compiler features mentioned above are already included in the current version of the DAPHNE compiler, and we are still actively working on making the compiler more feature-complete, robust, and efficient.

[5] https://github.com/daphne-eu/daphne

References

[AB+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: TensorFlow: A System for Large-Scale Machine Learning. OSDI 2016.

[BA+20] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, Sebastian Benjamin Wrede: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. CIDR 2020.

[BB+14] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, Yuanyuan Tian: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. IEEE Data Eng. Bull. 37(3): 52-62 (2014).

[BE+17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah: Julia: A Fresh Approach to Numerical Computing. SIAM Rev. 59(1): 65-98 (2017).

[BK99] Peter A. Boncz, Martin L. Kersten: MIL Primitives for Querying a Fragmented World. VLDB J. 8(2): 101-119 (1999).

[BKM08] Peter A. Boncz, Martin L. Kersten, Stefan Manegold: Breaking the memory wall in MonetDB. Commun. ACM 51(12): 77-85 (2008).

[CH90] Michael J. Carey, Laura M. Haas: Extensible Database Management Systems. SIGMOD Rec. 19(4): 54-60 (1990).

[CM+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018.

[DB+22] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina M. Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios I. Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaz Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz P. Wrosz, Ales Zamuda, Ce Zhang, Xiaoxiang Zhu: DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. CIDR 2022.

[DB23] Patrick Damme, Matthias Boehm: Enabling Integrated Data Analysis Pipelines on Heterogeneous Hardware through Holistic Extensibility. NoDMC@BTW 2023.

[EL+17] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen: SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. CIDR 2017.

[GB+23] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, Jana Giceva: Bringing Compiling Databases to RISC Architectures. Proc. VLDB Endow. 16(6): 1222-1234 (2023).

[GK+11] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, Shivakumar Vaithyanathan: SystemML: Declarative machine learning on MapReduce. ICDE 2011.

[GM93] Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993.

[HD23b] Immanuel Haffner, Jens Dittrich: A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. EDBT 2023.

[HD23a] Immanuel Haffner, Jens Dittrich: mutable: A Modern DBMS for Research and Fast Prototyping. CIDR 2023.

[JKG22] Michael Jungmair, André Kohn, Jana Giceva: Designing an Open Framework for Query Optimization and Compilation. Proc. VLDB Endow. 15(11): 2389-2401 (2022).

[KK+19] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, Volker Markl: An Intermediate Representation for Optimizing Machine Learning Pipelines. Proc. VLDB Endow. 12(11): 1553-1567 (2019).

[KLN18] André Kohn, Viktor Leis, Thomas Neumann: Adaptive Execution of Compiled Queries. ICDE 2018.

[LA04] Chris Lattner, Vikram S. Adve: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO 2004.

[LA+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, Oleksandr Zinenko: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO 2021.

[LY99] T. Lindholm and F. Yellin, Java Virtual Machine Specification, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[N11] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4(9): 539-550 (2011).

[PG+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala: PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS 2019.

[R15] Matthew Rocklin: Dask: Parallel Computation with Blocked algorithms and Task Scheduling. SciPy 2015.

[RM19] Mark Raasveldt, Hannes Mühleisen: DuckDB: an Embeddable Analytical Database. SIGMOD 2019.

[RP+16] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, Mikhail Smelyanskiy: Sparso: Context-driven Optimizations of Sparse Linear Algebra. PACT 2016.

[R+21] J. Reinders et al. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL (p. 548). Springer Nature, 2021.

[SA+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD 1979.

[SB+19] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, Peter J. Haas: MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. SIGMOD 2019.

[SK10] J. Sanders and E. Kandrot. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[SN22] Moritz Sichert, Thomas Neumann: User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. Proc. VLDB Endow. 15(5): 1119-1131 (2022).

[UP+20] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, Erich Focht: Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. CIDR 2020.

[D2.1] DAPHNE: D2.1 Initial System Architecture, EU Project Deliverable, 08/2021.

[D2.2] DAPHNE: D2.2 Refined System Architecture, EU Project Deliverable, 08/2022.

[D3.1] DAPHNE: D3.1 Language Design Specification, EU Project Deliverable, 11/2021.

[D3.2] DAPHNE: D3.2: Compiler Prototype, EU Project Deliverable, 02/2022.

[D3.3] DAPHNE: D3.3 Extended Compiler Prototype, EU Project Deliverable, 05/2023.

[D4.1] DAPHNE: D4.1 DSL Runtime Design, EU Project Deliverable, 11/2021.

[D4.1] DAPHNE: D4.3 Improved DSL Runtime Prototype and Overview, EU Project Deliverable, 11/2023.

[D5.1] DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks, EU Project Deliverable, 11/2021.

[D6.3] DAPHNE: D6.3 Prototype and Overview of Data Path Optimizations and Placement, EU Project Deliverable, 11/2023.

[D7.1] DAPHNE: D7.1 Design of Integration HW Accelerators, EU Project Deliverable, 05/2022.

[D7.3] DAPHNE: D7.3 Prototype and Overview Code Generation Framework, EU Project Deliverable, 11/2023.