D2.2 Refined System Architecture

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

Version 1.4

PUBLIC

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

Document Description

This report on the refined system architecture summarizes the design of the DAPHNE system infrastructure. The described design decisions resulted from continuous, detailed discussions with all project partners from use cases and benchmarking efforts (WP8 and WP9), over system architecture, DSL abstractions and compilation (WP2 and WP3), to runtime integration and scheduling (WP4 and WP5), as well as computational storage and hardware (HW) accelerators (WP6 and WP7). In order to make this system architecture self-contained, it includes the overall DAPHNE motivation, a requirement analysis, a description of the overall system architecture and its key components, the design of a new vectorized execution engine as a central orchestration component for heterogeneous HW and distributed processing as well as fine-grained fusion, parallelism, and scheduling, and the design of extensibility as a cross-cutting aspect of the architecture. Essential parts of this design have already been implemented and are available as open-source software on GitHub.

This refined system architecture is largely based on the initial system architecture described in deliverable D2.1 as well as the language design specification described in deliverable D3.1. Furthermore, this deliverable reflects on several lessons learned during the project, and describes several aspects of the system in more detail than before.

D2.2 Refined System Architecture			
WP2 – System Architecture			
Type of document	R	Version	1.4
Dissemination level	PU		
Lead partner	KNOW		
Author(s)	Patrick Damme (KNOW), Matthias Boehm (KNOW)		
Reviewer(s)	Pınar Tözün (ITU), Dirk Habich (TUD)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial draft: copied and adapted/extended contents from deliverable D2.1; minor/no changes in Sections 1, 2, 4, 6; significant changes in Section 3; Section 5 still empty	Patrick Damme
V1.1	Draft of Section 5, more on context objects in Section 3.3	Patrick Damme
V1.2	Minor cleanups, added deployment environments, and more references	Matthias Boehm
V1.3	Incorporated feedback from Pınar Tözün and Dirk Habich. In that context, also added Section 6 on the refinements.	Patrick Damme
V1.4	Finalized references	Patrick Damme

1 Introduction

Modern data-driven applications in many domains deal with increasingly large and heterogeneous data collections, as well as a variety of machine learning (ML) models for cost-effective automation and improved analysis results. Examples include ML-assisted manufacturing, biomedical engineering [A17], natural sciences, remote sensing, transportation, health-care, and finance [Z+19], which often include data access via open formats, data pre-processing and cleaning, ML model training and scoring, HPC libraries and custom codes, but also ML-assisted simulations [A+19b, P+21], and data analysis of simulation outputs [BDS14]. These complex end-to-end analysis requirements create a trend towards *integrated data analysis (IDA) pipelines* that jointly utilize techniques from data management (DM), high-performance computing (HPC), and ML systems.

Deployment Challenges: Developing and deploying such IDA pipelines is, however, still a painful process of integrating different systems and related developers, programming paradigms, resource managers, and data representations. Common tools include local or distributed analytical database systems [D+16, RM20]; flexible data-parallel computation frameworks like Spark [Z+12], Flink [C+15], or Dask [R15]; distributed ML systems like TensorFlow [A+16] or PyTorch [P+19]; domain-specific systems and libraries; and custom application codes. Integrating DM+ML, HPC+ML, DM+HPC for improving productivity and/or performance are old problems though. Examples go back to Jim Gray's work on the Sloan Digital Sky Survey [S+00], decades of data mining and advanced analytics, in-DBMS ML [KBY17], array databases like SciDB [S+11], and more recently, data management around ML systems (e.g., TensorFlow TFX [B+17]), and HPC-inspired (e.g., topology-aware) data management and query processing [BKS20]. However, an open system infrastructure for seamlessly developing and running IDA pipelines is still missing, and at the same time, new challenges related to hardware, productivity, and utilization emerge.

HW Challenges: Interestingly, data management, HPC, and ML systems share many compilation and runtime techniques; and together stress every HW aspect of storage, computation, and networking. Accordingly, these systems are strongly impacted by HW challenges such as the end of Dennard scaling and the end of Moore's law, which ultimately lead to dark silicon (not all parts of a chip can be powered simultaneously) [JP13] and increasing specialization at device level (CPUs, GPUs, FPGAs, ASICs), storage level (computational memory/storage, storage hierarchies), and workload level (data types and sparsity). Similar to – and triggered by – the trend to IDA pipelines, the underlying HW environment of DM/HPC/ML systems converges as well. This HW specialization in turn leads to increasing heterogeneity and thus, even larger productivity and utilization challenges for pipelines across DM, HPC, and ML systems. Although it might appear overly ambitious, we argue that it is time for building a dedicated system infrastructure – albeit utilizing existing compilation frameworks and runtime libraries – that can mitigate these challenges jointly.

Contributions: The DAPHNE project sets out to build an open and extensible system infrastructure for integrated data analysis pipelines. For good integration and extensibility, we base this infrastructure on MLIR [LA+21] as a multi-level, LLVM-based intermediate representation backed by multiple organizations and communities. This approach allows a seamless integration with existing applications and runtime libraries (e.g., BLAS/LAPACK,

collective operations, task scheduling, DNN operations, compression, I/O, and column-vector primitives), while also enabling extensibility for specialized data types, hardware-specific compilation chains, and custom scheduling algorithms. In this report, we share the motivation and design of the overall DAPHNE system, including the following technical contributions:

- **IDA Pipelines:** We first make a case for IDA pipelines by example of real-world use cases, and then summarize their main characteristics, challenges, and opportunities by requirements on related system infrastructure in Section 2.
- **System Architecture:** Subsequently, we describe the overall MLIR-based architecture, data representations, and major design decisions in Section 3.
- **Vectorized Execution:** We further introduce a vectorized (tiled) execution engine for compiled operator pipelines of frames and matrices; heterogeneous HW devices, and computational storage in Section 4.
- **Extensibility:** Moreover, we discuss the design for extensibility in the DAPHNE system architecture in Section 5.

After that, we explicitly highlight the refinements compared to the initial system architecture in Section 6. Finally, we conclude this report in Section 7.

2 Requirements Analysis

Integrated data analysis pipelines consist of complex, often multi-phase workflows of ETL (extraction transformation loading) processes, ML training/scoring, numerical computation or simulation, and query processing and data analysis. In order to raise awareness of this trend towards IDA pipelines, we briefly describe a representative real-world use case, and summarize common characteristics, challenges, and opportunities.

2.1 Example Use Case

We describe a selected example use case from the area of earth observation, but it is representative for a much broader range of applications. Deliverable D8.1 [D8.1] describes the DAPHNE use cases – from earth observation, semiconductor manufacturing, material degradation, and vehicle development – as well as their initial IDA pipelines in more detail.

Earth Observation: Local climate zones (LCZs) classification categorizes patches of satellite images for modeling climate-relevant surface properties (e.g., surface imperviousness and structure) [SO12, Z+19b]. This use case leverages the Sentinel-1 synthetic aperture radar data and Sentinel-2 optical images (obtained by the European Space Agency as part of the Copernicus initiative), where one year of global data is already in the range of 4 PB. For training LCZs classifiers, the DLR team materialized and published a labeled dataset, called `So2Sat LCZ42` [Z+19b, Z+20] that consists of 400,673 pairs of Sentinel-1/Sentinel-2 image patches (32x32) and LCZ labels. The labels were hand-annotated by 15 experts in a month, and verified by 10 experts casting votes for a subset of the dataset, yielding a high confidence of 85%. Together, the train, test, and validation data account for ~55.1 GB in HDF5 format. The training pipeline includes Sentinel-2 pre-processing steps as well as training a ResNet20 [H+16, H+16b]

classifier. However, the main challenge is applying the scoring pipeline efficiently at peta-byte scale: reading the data from complex storage hierarchies, applying pre-processing, quantization into fixed-point representations, forward pass of ResNet20, materialization and subsequent spatio-temporal data analysis (e.g., for research of global urbanization).

2.2 Challenges and Opportunities

The main characteristic of many IDA pipelines is the composition of complex workflows including data pre-processing, ML training and scoring (often with multiple models), numerical computation and simulations, human intervention and large project teams, as well as query processing of input data and intermediates. The following list identifies key requirements on related system infrastructure:

- **Seamless high-level APIs and DSLs** (DM, HPC, ML; operations and primitives for common computation tasks such as data cleaning, feature transformations, SQL query processing, ML algorithms, and model debugging; mini-batch and batch training)
- **Extensible infrastructure** (data types, kernels, metrics, scheduling; with externalized multi-level compilation for early adoption by researchers, HW vendors, platform developers, and performance engineers)
- **Interoperability between frame and matrix operations** (seamless data conversion, shared data structures and operations, common zero-copy slicing and data access)
- **Integration with resource management, programming models, and specialized libraries** (seamless integration of existing DM, HPC, ML libraries for reuse and interoperability; integration with resource management for improved resource sharing; timely adoption of, and integration with, new deployment models and infrastructure)
- **Local and Distributed Data Representations** (local, distributed, and out-of-core datasets; with dense, sparse, and irregularly ragged or nested data formats; and heterogeneous, multi-modal input formats)
- **Heterogeneous Hardware** (awareness and utilization of storage hierarchies, computational storage with sync and async I/O, and heterogeneous accelerators with a spectrum of interfaces from high-level kernel abstractions to increasing specialization)
- **Fine-grained operator fusion and parallelism** (operator fusion and code generation for workload characteristics and heterogeneous HW, with data and plan partitioning across devices and nodes for full utilization of available hardware resources)

With system infrastructure addressing these requirements, new opportunities arise. Examples include tightly integrated ML-assisted simulations; materialization decisions for late data augmentation during ML training, and query processing of simulation outputs; as well as improved scheduling and resource utilization in shared cluster environments.

3 DAPHNE Architecture

DAPHNE is an open and extensible system infrastructure for developing and executing integrated data analysis pipelines [D+22]. In this section, we share the design of the overall system architecture and its key components. DAPHNE is both, a stand-alone executable as well

as a shared library, which can be integrated into other programs (e.g., for efficient data transfer). Since March 2022 DAPHNE is available as open-source software under Apache 2.0 license on GitHub (https://github.com/daphne-eu/daphne), where is it continuously extended.

3.1 Architecture Overview

The DAPHNE system architecture is shown in Figure 1. DAPHNE is built from scratch in C++ (for seamless integration with HW specialization), but utilizes MLIR [LA+21] as a multi-level, LLVM-based intermediate representation (IR) as well as existing runtime libraries such as BLAS, LAPACK, and DNN kernels as well as collective operations. These libraries are augmented with more specialized, custom kernel implementations. Users specify their IDA pipelines in the DaphneDSL (a language similar to Julia, PyTorch, or R) or DaphneLib (a high-level Python API with lazy evaluation that internally compiles DaphneDSL scripts as well). These scripts are then compiled – via a multi-level compilation chain – into executable runtime plans, which can be executed in a local or various distributed environments. Furthermore, the system is equipped with a suite of tools, e.g., for getting insights into the compiler and runtime.

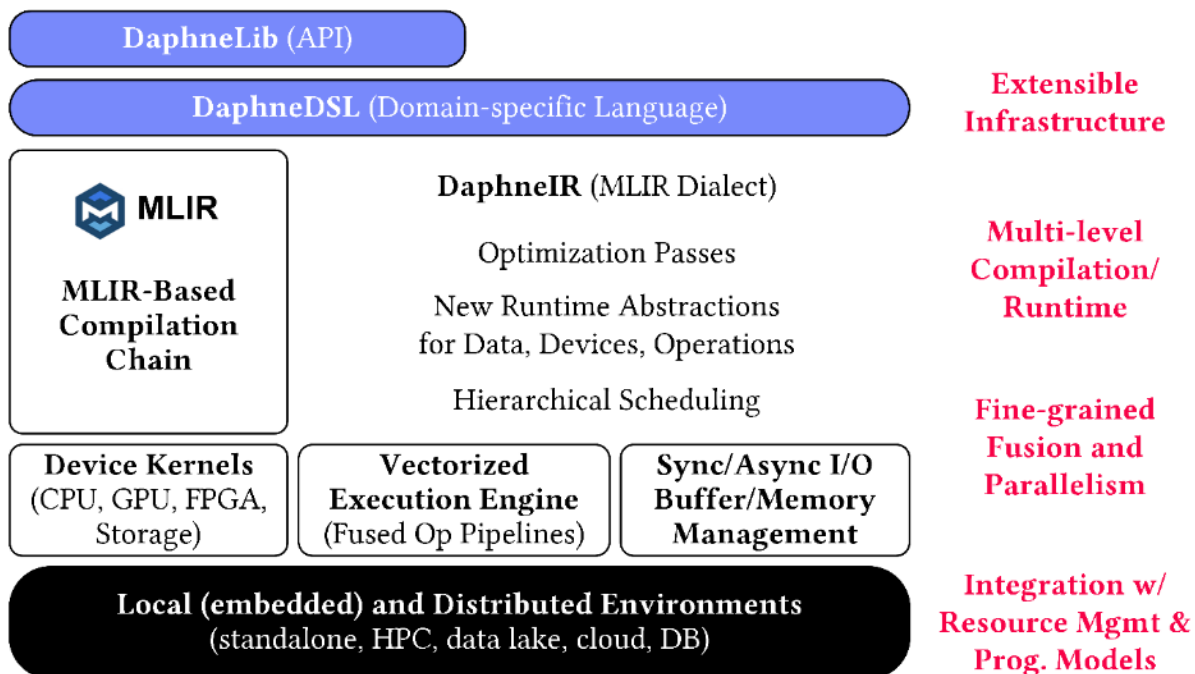


Figure 1: DAPHNE System Infrastructure.

User Frontend: The main entry point to the system is DaphneDSL, a domain-specific language inspired by ML systems as well as languages and libraries for numerical computation like Julia [BK+12], Python NumPy [HM+20], R [MH+12], and SystemDS DML [BA+20]. DaphneDSL is based on three design principles:

- **Frame and matrix operations:** DaphneDSL offers hundreds of operations from relational algebra and linear algebra as well as various aggregation and statistical

functions. These high-level operations retain the semantics for optimizations in the compiler and simplify parallelization and lowering to kernels for emerging HW.

- **Data independence:** Users only work with abstract frame, matrix, and scalar data types instead of specifying data representations like dense, sparse, and compressed; or data locations like local CPU, local GPU, or distributed. The compiler and runtime optimize the IDA pipeline for the given data and deployment characteristics. Combining these data types with value types such as SI8, SI32, SI64, UI8, UI32, UI64, FP32, FP64 (i.e., various integer and floating-point types) as well as strings, provides a simple, powerful and extensible type system.
- **Extensibility:** Given the expectation of increasing specialization across the software and hardware stack, DAPHNE's design and language abstractions aim for good extensibility to allow researchers to quickly experiment with new prototypes and extensions of system infrastructure for new environments. More details on extensibility at API and DSL as well as compiler and runtime level are described in Section 5.

DaphneDSL further supports conditional control flow in the form of branching and loops with arbitrary nesting levels, typed and untyped functions, as well as additional second-order language abstractions (e.g., for SQL processing, parameter servers and mini-batch training, and user-defined functions with different data bindings). To facilitate user productivity, DaphneDSL offers a rich hierarchy of DSL-based primitives, e.g., for data cleaning, feature transformation, ML algorithms, model debugging, and (mini-)batch training, which can be imported into a user script. For a seamless integration into typical workflows and environments of data scientists, ML and data researchers, as well as experts of specific application domains, DAPHNE offers DaphneLib as a Python API. Python is currently undoubtedly the primary entry point to ML and other data systems. DaphneLib allows calling basic (i.e., primitive operations from linear and relational algebra, such as transposition and selection) and higher-level (i.e., DSL-based functions for typical ML and other algorithms, such as `components()`, and second-order functions, such as `map()`) DaphneDSL built-in functions on DAPHNE matrices or frames. These inputs can be created from pandas data-frames or NumPy arrays, whereby we strive for highly efficient data transfer via shared memory or inter-process communication. Similar to PySpark [ZC+12] and Dask [R15], DaphneDSL uses lazy evaluation, i.e., operations merely build up an internal DAG representation of the data flow, while evaluation is triggered explicitly. On evaluation, a DaphneDSL script is assembled and executed, thereby reusing the entire DAPHNE compilation chain with all related optimization passes. More details on DAPHNE's user frontend can be found in deliverable D3.1 [D3.1].

Compilation Chain: DaphneDSL scripts are converted by an ANTLR parser into MLIR, specifically, DaphneIR as an MLIR dialect. MLIR [LA+21] is a customizable compiler infrastructure for reuse and low-cost domain-specific compilers. DaphneIR defines matrix and frame data types, logical and physical frame and matrix operations, and various traits (e.g., for schema, type, and shape inference). Conditional control flow is supported by integrating with the existing MLIR dialect SCF (structured control flow). After parsing, we apply various MLIR compiler passes. These passes continuously lower the IR from high-level, abstract operations and data types over multiple levels of operator specialization (e.g., local/distributed operations, device placement on CPUs/GPUs/FPGAs, choice of physical kernels for specific environments and devices), as well as data specialization (e.g., DenseMatrix/CSRMatrix representations) down

to MLIR's LLVM dialect, which can directly be just-in-time compiled and executed. In contrast to other MLIR dialects, we lower frame and matrix operations to C++ kernels and only use LLVM for control flow and scalar operations. At the same time, this design still allows the implementation of selected kernels in LLVM, or other MLIR dialects, if beneficial. Additionally, there are optimization passes rewriting the IR to improve the program's behavior in terms of runtime, memory consumption, and others. In particular, we apply MLIR programming language rewrites (e.g., common subexpression elimination, constant propagation, constant folding, branch removal, code motion/loop hoisting, function inlining/unrolling), type and property inference (e.g., data and value types, shape/dimensions, schema, sparsity/cardinality, symmetry), inter-procedural analysis (analysis of function call graphs, propagation of types, dimensions, properties), algebraic simplification rewrites from linear and relational algebra, operator ordering (e.g., join ordering/enumeration, matrix multiplication chain optimization, sum-product optimizations, data-flow-graph linearization), the generation of fused operator pipelines, and memory management (update-in-place, reuse of allocations, garbage collection). Passes for optimization and lowering can be interleaved and repeatedly executed. The joint system infrastructure also enables cardinality [BH+07, MNS09] and sparsity [SB+19] estimation in a holistic manner. More information on the initial compiler design can be found in deliverable D3.1 [D3.1], and it will be extended in deliverable D3.4 by M36.

Runtime: At runtime, the kernels are executed sequentially and produce materialized intermediates in memory with copy-on-write semantics and operator-level synchronization barriers. Besides this basic execution model, DAPHNE will adopt hierarchical scheduling mechanisms for ML pipelines; task-parallel loops and operations; data-parallelism across nodes, devices, NUMA nodes, and cores; as well as vector-instruction-parallelism. Our vectorized execution engine – as described in Section 4 – further provides means of operator fusion, and a seamless integration of heterogeneous computing devices, computational storage, and distributed operations.

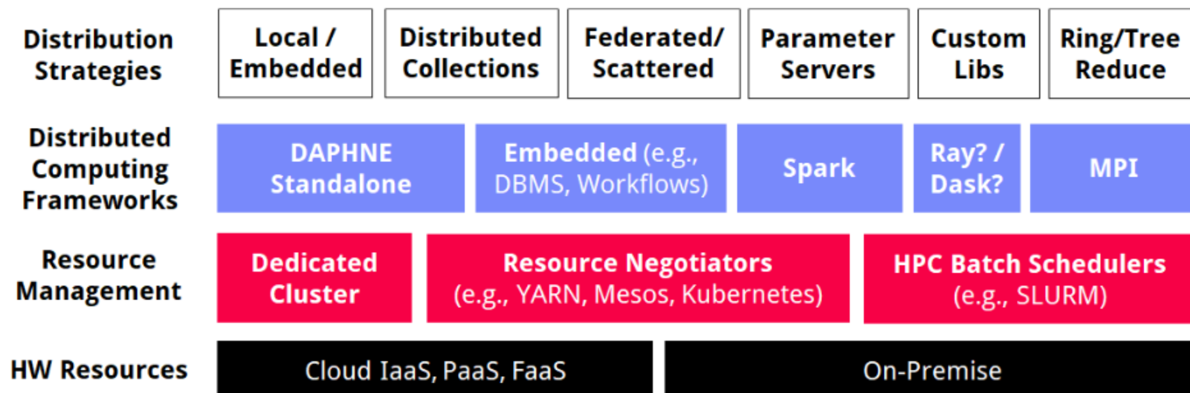


Figure 2: Deployment Environments and Integration.

Deployment Environments: As already alluded to in the overall system architecture (see Figure 1), we aim to enable the deployment of the DAPHNE system in a variety of environments

in terms of hardware resources, cluster resource managers, and distributed computing frameworks. Figure 2 gives an overview of selected relevant dimensions. First, hardware resources might be dedicated on-premise or cloud resources, where the latter include Infrastructure as a Service (e.g., compute instances, object storage), Platform as a Service (e.g., provisioned Spark clusters), and Function as a Service (e.g., elastic, state-less task execution). Second, relevant resource management strategies include statically provisioned clusters, resource managers like YARN [V+13, C+19], Mesos [H+11], and Kubernetes [B+16], as well as HPC schedulers like SLURM [Li12]. Similar to the integration with different resource managers, we also aim to support different storage abstractions such as local file systems and raw devices; distributed file systems like HDFS, Lustre (as used by DLR), and CephFS (as used by UM); as well as object storage such as S3 and OpenStack Swift. Third, besides DAPHNE's standalone distributed runtime, we foresee integrations of DAPHNE components in distributed computing frameworks like Spark [Z+12], Ray [M+18], and Dask [R15], as well as database management systems and workflow orchestration systems like AirFlow. There are use cases for running DAPHNE embedded in UDFs of such frameworks, but also to use these frameworks for distributed operations in DAPHNE. From an architecture perspective, we will devise appropriate abstractions in terms of extension hooks to enable extensibility at these different levels as well as to separate concerns of other DAPHNE components. Example abstractions are so-called distribution primitives for managing distributed collections of tiles and federated data objects (see data representations in Section 3.2), parameter servers for mini-batch training, and reusable primitives for broadcasting, prefetching, and various aggregations. So far, our implementation efforts centered around the DAPHNE standalone distributed runtime, dedicated on-premise and HPC clusters and resource managers, as well as key distribution primitives based on gRPC as a communication library. However, we are already actively exploring the integration of MPI and NCCL collective operations (e.g., reduce-all) as well.

Tooling: While the components for the specification, optimization, and execution of IDA pipelines are the core of the DAPHNE architecture, there is also a need for tooling around this core. In the following, we describe two important tools for explanations and profiling.

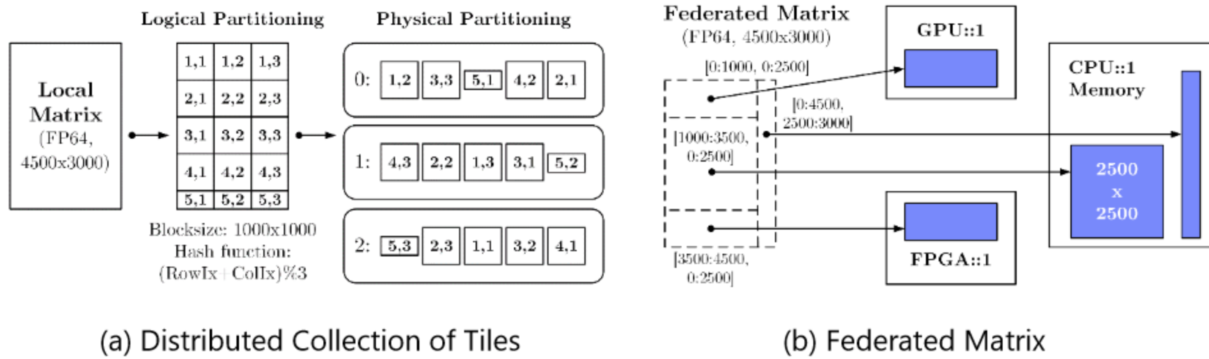
- **Plan Explanations:** As a means to understand the complex decisions made by the compiler in its various passes, DAPHNE ships with tools for investigating the IR at any level. Similar to the well-known `EXPLAIN` statement from SQL, this tool can be used to understand compiler decisions and estimated properties such as data characteristics and costs. For instance, a user can invoke `daphne` with the `--explain` flag, which takes the names of one or multiple compiler passes as a parameter. Upon that, the system prints the IR after the application of each specified pass. By default, the IR is displayed in MLIR's standard IR notation, examples of which can be found in deliverable D3.1 [D3.1]. This notation reveals which operations are performed on which data in which order, as well as the data and value types of inputs and outputs, where already known. Additional information is attached to operations as MLIR attributes (e.g., estimated cost) and to input/output types as properties (e.g., shape and sparsity). An early version of this plan explanation tool has been shown in deliverable D3.2 [D3.2] on the compiler prototype. This tool will be valuable for both the development of the compiler as well as the deployment and configuration of IDA pipelines by users. In the future, we aim to extend the tool by means of workload summarization and visualization.

- **Monitoring/Profiling:** This suite of tools provides insights into the execution behavior of the system at runtime. As a consequence of the hierarchical nature of DAPHNE, approaches for monitoring can be applied at various levels. For instance, tailored scripts applying external and widely used tools like `perf` to DAPHNE can be used to get access to hardware performance counters and to find out where time is spent. Furthermore, tools like Intel VTune can be used to instrument the DAPHNE source code, but target only Intel hardware. A deeper and more pointed integration can be achieved by using frameworks like PAPI, likwid, or OpenHPC directly. In addition to tool-based and library-based profiling, time measurements will be done natively in the DAPHNE prototype. While each of the former fits the local execution of DAPHNE, dedicated tools can be applied for the distributed and multi-device settings. For instance, gRPC comes with profiling support (e.g., Google cloud profile) and MODA can be used for profiling MPI. GPU vendors like NVIDIA also offer HW-specific tools, such as the Nsight tool suite and the DCGM monitoring tool. We see two major uses for monitoring/profiling: (1) it enables developers or administrators to identify performance bottlenecks, and (2) it can be used by the system itself to adapt its behavior, e.g., in terms of advanced, utilization-aware scheduling algorithms. To this end, DAPHNE will provide summary statistics of executed kernels and functions (frequency and total runtime) as well as compiler- and runtime-specific metrics.

3.2 Data Representation

DAPHNE's basic data types are frames, matrices, and scalars, where a frame is a bag (unordered multiset) of tuples with a schema, and a matrix is a two-dimensional array. Each matrix, scalar or frame-column has a value type (e.g., FP32, BF16, UI8, STR). At a later point in time, we may add tensors as multi-dimensional arrays of homogeneous value type. At DSL level, users deal with these abstract data types, and the compiler systematically lowers operations to kernels that produce local or distributed physical data structures.

Local Data Structures: DAPHNE's core data structures are dense or sparse matrix formats. Both use row-major representations: a dense linearized one-dimensional array, and a compressed sparse row (CSR) [S94] format of row offsets, column-index, and value arrays. While matrices are homogeneous arrays of a particular value type, frames have a schema and thus, require the handling of multiple value types. Given common analytic workload characteristics, our frames rely on a column-oriented storage implemented via a dense matrix per column or column group. This composition allows the reuse of matrix operations as frame operations. Finally, for zero-copy indexing (e.g., slicing or vectorized execution), each matrix can specify a view window on top of a potentially larger array.



(a) Distributed Collection of Tiles (b) Federated Matrix

Figure 3: Examples of Distributed Data Structures.

Distributed Data Structures: The distributed matrix and frame representations are then composed from the local data structures. We support the following two abstractions that give a great balance of flexibility and control:

- **Distributed Collections of Tiles:** A matrix is divided into fixed-size blocks and stored as a collection of block-indexes and blocks [KBY17]. By default, such a bag is unordered but can be partitioned (hash, range) or sorted. Figure 3.a shows an example 4500-x-3000 matrix, organized as a collection of squared 1K-x-1K blocks and hash-partitioned into three partitions, which can then be stored and processed in a distributed manner.
- **Federated Matrices/Frames:** A federated matrix is a virtual matrix whose individual parts (identified by index ranges, and address information for accessing remote data) are stored as local or distributed data at a federated site [B+21] or device. Figure 3.b shows again an example 4500-x-3000 matrix that is federated across host and device memory, where the federated metadata comprises index ranges and pointers.

Both of these abstractions are amenable to data-parallel computation, but they have different tradeoffs regarding distribution, load balancing, sparsity, and direct access.

Metadata and Placement: Both local and distributed data structures also store metainformation, including the shape, sparsity, symmetry, sort order, and data placement. Some of these might already be known at compile-time due to inference passes. However, especially in presence of data-dependent operations and conditional control flow, this information might only become available at runtime. Thus, storing the metadata in runtime data structures provides a maximum degree of flexibility. Metainformation can be exploited by kernels, e.g., to choose a more efficient algorithm for sorted data. The data placement is a particularly important property for both the local and distributed runtime. In hybrid runtime plans that utilize distributed workers and/or heterogeneous hardware within a single worker, a data object might be partitioned or replicated across cluster nodes and/or devices. Specifically, matrices and frames reference the host data (which can be a `nullptr`) and/or data on distributed workers, HW accelerators, and computational storage devices. The nature of such a reference depends on the respective backend, and could be comprised of, e.g., an IP address, port, and node-local key for some distributed backend, or a CUDA device identifier and a device-internal pointer for hardware accelerators integrated via CUDA. Furthermore, for

federated data representations, the metadata at the coordinator also includes the detailed information about data partitions (row/column ranges) and data location per partition.

3.3 Local and Distributed Runtime

Kernels: The compiler generates an execution plan with calls to C++ host kernels for local, distributed, or accelerator operations. Our kernels make heavy use of C++ metaprogramming for both value types and combinations of dense and sparse inputs. This approach allows us to automatically generate template instantiations for the operations that require specialization. By default, this is done while building the DAPHNE system. More precisely, the type combinations to pre-compile per kernel are given by a configuration file. Since pre-compiling all possible combinations of input and output data and value types for each kernel would result in an unacceptable build time and binary footprint, we apply a number of mitigation strategies. Most importantly, we pre-compile only selected, frequently used kernels. Initially, we will focus, e.g., on kernels with homogeneous input data and value types and dedicate more combinations, e.g., of dense and sparse inputs and outputs, to more costly operations like matrix multiplication. The set of pre-compiled kernels will evolve over time as we gain insights into the kernel usage in the DaphneDSL/DaphneLib implementation of the use case IDA pipelines from WP8. The compiler restricts itself to using these pre-compiled kernels and injects casts to adapt the input and output types accordingly. Some of these casts, e.g., casting an FP32 frame-column to an FP32 matrix) are no-ops. Furthermore, we plan to investigate the compilation of the necessary kernel specializations on-the-fly during the compilation of a DaphneDSL script. This could be done either by means of a C++ compiler generating a complementary shared library of additional kernels from the C++ sources, or by means of generating the kernel code in low-level MLIR operations, akin to other systems building on MLIR, whereby both have their individual advantages and disadvantages.

Context Objects: Access to distributed runtimes and HW accelerators is encapsulated in a context object that is passed to individual kernels. The initializers of specific contexts are local kernels themselves that add state to the global context. This approach simplifies the integration of new accelerators via shared libraries of kernels and optimization passes. Besides that, context objects are also used to pass the user configuration to kernels. Finally, the context also provides information on the current level within a nested structure of (parfor) loops, including information on the kind and degree of parallelism available in that context.

Distributed Runtime: We aim for an integration with different distributed programming models and resource managers (see Section 3.1). As a first step, we are building the DAPHNE standalone distributed runtime with dedicated worker processes, simple communication, and a basic integration with SLURM as a common HPC resource manager. For the communication part, we focus on two distributed backends. On the one hand, we already have a running RPC-based prototype, which utilizes gRPC `send` and `receive` primitives for individual data transfers between coordinator and worker nodes. On the other hand, we are currently developing an MPI-based prototype, which utilizes OpenMPI collective operations, such as broadcast, scatter, and `all-reduce`. Subsequently, we will focus on improving performance by devising advanced MPI distribution primitives. Both backends have integration points in the compiler and runtime

as well as the vectorized execution engine. Most importantly, the operator fusion performed by the compiler determines the code to be executed by each node. This code is sent to the workers in the form of an MLIR snippet, which can be compiled in an architecture-aware manner at the individual workers. The distribution of the data and spawning of distributed jobs is done at the beginning of a fused pipeline. While DAPHNE will support multiple distributed communication frameworks (e.g., RPC and MPI), only one of them can be used per invocation of the DAPHNE system. In the future, we aim to further integrate with external distributed backends (e.g., Spark and Dask), device-specific collective operations (e.g., NVIDIA NCCL), and embedded deployments in different HPC, cloud, and database environments. Accordingly, we keep the initial design generic enough to allow such extensions.

Memory Management and Garbage Collection: DAPHNE automatically manages the memory required for the data objects in a DaphneDSL script. The system ensures that there are no memory leaks or double frees. Moreover, we aim at reusing allocated buffers and existing data as much as possible to avoid unnecessary allocation and copying. For instance, when creating a view into a segment of a larger matrix using right indexing in DaphneDSL, both the view and the original matrix share the same underlying data with copy-on-write semantics. Moreover, the same data object could be referenced by different variables in a DaphneDSL script, potentially even depending on conditions evaluated at runtime. Thus, the system manages memory at two levels: At the level of (shallow) data objects like `Frame` or `DenseMatrix`, reference counters are employed to track active uses. The compiler inserts operations to increase and decrease the counters on certain SSA (static single assignment) values, while the runtime takes care of freeing a data object once its counter becomes zero. However, the object's underlying data might still be shared with other objects. Thus, the runtime also employs reference counting at the level of individual (large) data buffers. Sharing a data buffer between two data objects increases the reference counter of a buffer, while the destruction of an object decreases the counter of the underlying data buffers. A data buffer is released once its counter becomes zero, but instead of giving it back to the operating system immediately, we could still keep it in a buffer pool to serve upcoming memory requests. We expect this optimization to be especially useful in the context of the vectorized execution engine, elementwise (and other shape-preserving) operations, and iterative algorithms with constant size of intermediates.

3.4 Accelerators and Storage

In this section, we summarize how hardware accelerators and storage are integrated into the overall system. More detailed designs of managed storage tiers, and near-data processing, as well as the integration and full utilization of HW accelerators can be found in the dedicated deliverables D6.1 [D6.1] and D7.1 [D7.1].

Data Transfer: Most HW accelerators like GPUs, FPGAs, and near-SSD compute devices have a cache hierarchy and high-bandwidth memory. As discussed in Section 3.2, each DAPHNE frame and matrix has associated metadata including information on the placement of ranges of rows and/or columns on accelerator devices. This allows us to keep track of how a data object is partitioned or replicated over host memory and the memories of multiple devices in

hybrid runtime plans that utilize heterogeneous hardware. For example, a compiled GPU operation is called through its host kernel, which first invokes primitives to make the inputs available in GPU memory. If the data is already on the GPU, there is no additional transfer, and otherwise the primitive utilizes implicit (stream and discard) or explicit (copy and retain) means of data transfer. Additionally, we allow the compiler to inject `prefetch` and `broadcast` directives to overlay anticipated transfers with other operations. These distribution primitives nicely generalize to HW accelerators, the distributed runtime, and computational storage [BD21, LB21]. For example, for the DLR inference workload from Section 2.1, we might broadcast the quantization boundaries (and/or parts of the trained model) to the near-SSD CPU or FPGA, stream FP32 data from the SSD's flash chips, quantize the data in batches to UINT8, and thus, reduce the PCIe data transfer by 4x.

Device Kernels: We decided to implement device kernels in C/C++ (as opposed to lowering them to low-level MLIR/LLVM operations), in order to retain full control and because latest generation HW accelerators can usually be programmed that way, while we would have to wait for the respective MLIR/LLVM extensions for the lowering approach. One challenge connected to the implementation of device kernels is supporting a wide range of devices without having to reimplement kernels for every single device. Thus, we mainly focus on generic kernel implementations by adopting one of the following two approaches: First, general-purpose programming frameworks (e.g., OpenMP [DM98], OpenCL [M+11], CUDA [SK10], SYCL [RL16], and oneAPI [R+21]) are widely applicable since they are not specialized for a single domain or class of applications. However, they fail to capture the domain-specific, high-level semantics of HW accelerators by operating on a comparably low level of abstraction, e.g., reasoning about individual loops. Second, domain-specific programming frameworks (e.g., TVL [U+20], Weld [P+18], Voodoo [P+16], and Sierra [Le+14]) can be used to overcome these limitations, since they provide programming language abstractions for operations specific to a single domain. Furthermore, they can be mapped to the available hardware and the desired performance independently of the actual source code. To sum up, we will focus on using CUDA [SK10] for NVIDIA GPUs, oneAPI [R+21] and T2S [S+19] for FPGAs, as well as plain C++ kernels (with auto-vectorization) and TVL [U+20] for SIMD extensions on CPUs. Nevertheless, the DAPHNE system architecture is general enough to embrace other programming frameworks for hardware accelerators. Supporting all these approaches in one system also enables us to compare them with each other in terms of development time and kernel performance.

4 Vectorized Execution Engine

Basic runtime plans of kernels with materialized intermediates offer good performance and simplify debugging. Although this model is commonly used in most ML systems and many column stores, it suffers from several limitations. Materializing intermediates has large temporary memory and memory-bandwidth requirements, multi-threaded kernels create too fine-grained synchronization barriers per operator, and device placement of operators is too coarse-grained. In order to address these limitations, we introduce a vectorized execution engine for compiled operator pipelines of frames and matrices, which allows fine-grained operator fusion and parallelism across HW devices. This vectorized execution engine is also the

central means for parallelism in both the local (e.g., multi-threading and multi-device) and distributed (e.g., multiple worker nodes) runtime.

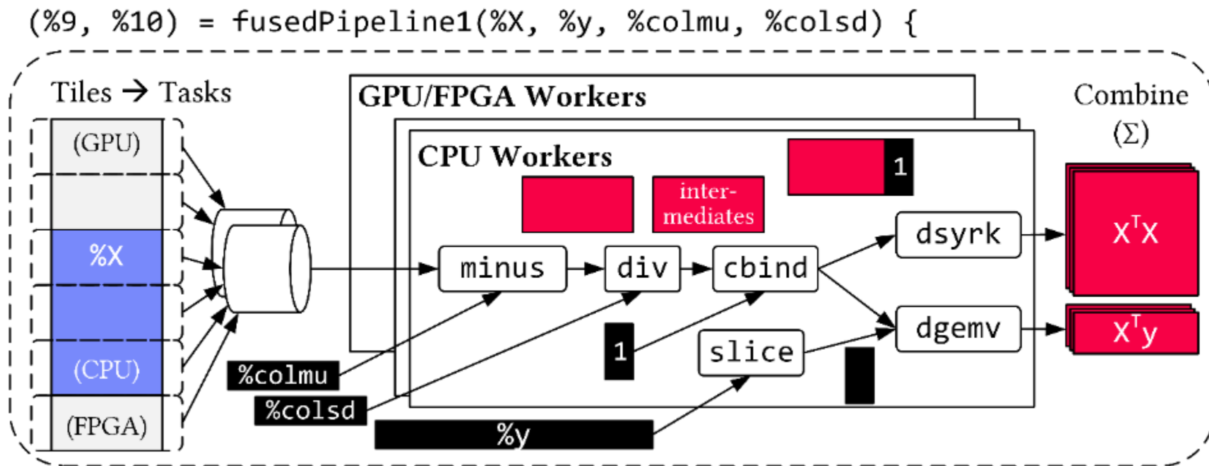


Figure 4: Vectorized Execution of Compiled Operator Pipelines
(with multi-device data and task placement).

Vectorized Task Execution: Figure 4 shows the basic integration of vectorized operator pipelines into execution plans. Similar to LLVM loops, such a vectorized pipeline has multiple inputs, multiple outputs, and an IR body. Additionally, we specify split (e.g., row slicing) and combine (e.g., row-bind concatenation or aggregate) functions. In this example, we perform matrix standardization (`(X-colMeans(X))/colSds(X)`), append a column of ones for intercept computation, and compute X^TX and X^Ty as part of a close-form linear regression algorithm. The input matrix X is federated across CPU, GPU, and FPGA memory, and vectorized execution creates tasks for aligned row partitions (similar to morsels [L+14]) and appends them to one or multiple (device-specific) task queues.

Vectorized Task: A task comprises its input data, an operator pipeline (graph) with a specific input data binding (scalar, row, or tile), outputs, and a combiner. The inputs and outputs can be zero-copy views (index ranges) or specific buffers, where the task size refers to the length of the range (e.g., number of rows). If the task size is greater than the data binding, the pipeline is invoked sequentially for each data item in the task.

Worker threads then read from the queue, execute the tasks, and combine the results (with worker-local aggregation if needed). Any HW accelerator worker is implemented as a CPU thread that launches the actual accelerator kernels. In the future, we will also explore the concept of standing accelerator kernels that consume tasks directly.

Fused Operator Pipelines: By controlling the task size, we can ensure bounded memory requirements and fit intermediates into the device caches. That way, the entire operator pipeline behaves like a dedicated, hand-crafted kernel. A task is the unit of scheduling with potential worker contention on shared task queues and outputs, and random access to the start of the task data. The more tasks (or the smaller the task size), the higher the overhead but

the better for load balancing. Separating task size from data binding provides additional flexibility. For example, the pipeline in Figure 4 can be invoked at row granularity (for which we could specialize the matrix multiplications `dsyrk` and `dgemv` to an outer product `dger` and `daxpy`), minimizing the size of intermediates. However, with sufficiently many features (e.g., >1000) every row's outer product and accumulation would flush the last-level cache. Instead, a tiling with multiple rows allows more efficient, cache-conscious operations.

Multi-device Scheduling: As shown in Figure 4, the vectorized execution allows a seamless integration of HW accelerators and scheduling. For CPU kernels, we leverage single-operator pipelines as the default multi-threading, which applies to many element-wise and aggregation operations. In this framework, we are further exploring different task partitioning and scheduling strategies, single and multiple task queues (e.g., device-specific with task stealing), and data-locality-aware scheduling, and runtime adaptation. Finally, vectorized execution also nicely integrates with computational storage where operator pipelines can be executed, for instance, on near-SSD CPUs or FPGAs; and the task queues can also connect asynchronous I/O and subsequent computation pipelines.

Code Generation: Vectorized execution also simplifies code generation. Instead of interpreting vector kernels sequentially, we can compile device-specific kernels for different workers, but reuse the split and combine infrastructure. Code generation allows fine-grained specialization, sparsity exploitation, and exploitation of reconfigurable devices like FPGAs. For CPU pipelines, we use MLIR which leverages LLVM for scalar data bindings, and vectorized kernels or libraries like BLAS and TVL [U+20] for matrices and frames; for hardware accelerators, we use the hand-crafted device kernels described in Section 3.4. Similarly, but largely unexplored, for computational storage, we aim to compile eBPF byte-code programs [LB21].

5 Extensibility

The overall design principle of an open and extensible system architecture is of utmost importance to enable low effort exploratory experimentation and custom extensions for new data types, operations, and hardware. We aim to support extensibility in terms of configurations of internal behavior, but also in terms of extensions for custom operations (at script level and kernels), custom data formats (e.g., vendor-specific binary formats), new data types (e.g., compressed types), and various extensions of the compiler and runtime system (e.g., additional IR dialects, new optimization passes, custom kernels and data types, and scheduling algorithms).

On the one hand, each aspect of the system architecture can be extended directly in the DAPHNE source code. On the other hand, we anticipate that certain aspects (e.g., device kernels and physical data types) will be extended especially often, since they are at the heart of specialization. Therefore, we try to make their extensibility as simple as possible. In particular, we will allow users to implement their extensions outside the DAPHNE code base, register them with the system at runtime, and employ them manually at script level or automatically by the compiler, where possible.

Finding a good balance between the expressiveness and increase in code complexity as well as a possible negative impact on performance implied by any abstractions required for extensibility, are challenges we will have to face. Nevertheless, we expect these to be outweighed by the advantages of enabling more use cases and improving performance by means of tailored algorithms or data representations.

5.1 API and DSL Extensibility

Extension Catalog: The central component for extensibility is an extension catalog that will allow the registration of dedicated artifacts (such as kernels or data types) in the form of shared libraries. For kernels, the extension catalog stores information like the DaphneIR operation it represents (if applicable), the name of the kernel function to call from a compiled IDA pipeline, the input/output data/value type combinations it supports (e.g., dense/sparse), interesting data properties it targets (e.g., sorted, symmetric), and optionally cost models for use by the compiler (e.g., runtime, memory consumption). For physical data types, the extension catalog records the logical data type they represent (frame/matrix), their integration into the vectorized engine (e.g., preferred slicing axis), potential constraints for their usage (e.g., requiring a symmetric matrix), and optionally cost models for use by the compiler (e.g., physical size). Based on this metadata, these extensions are represented in DaphneIR and thus included in various optimization passes such as shape inference, operator selection, and physical representation selection. The concrete use of extensions can be further influenced both at script level (mostly by users or during experimentation) or via configuration files that influence the entire deployment and thus, potentially many users.

Extensibility at Script Level: DaphneDSL offers dedicated built-in functions for data representations, data placement, and operator placement. For instance, after registering a new compressed matrix representation `ComprMatrixXYZ`, a user could enforce this format to be used by invoking `Y = as.ComprMatrixXYZ(X)`. Here, we reuse DaphneDSL's cast syntax to express the representation. Furthermore, a user could ensure that Y is placed on a specific device by `Y = device(X, "/GPU:0")`. Finally, the placement of operations (such as a matrix multiply @) on devices can be dictated by `X = Y @_gpu Z`. Besides that, users can also specify concrete last-level kernels to use, e.g., `s = §my_sum_fpga_int16(X)`. All these script-level decisions loose data independence but are user choices and will be treated as constraints. The optimizing compiler then handles remaining operations around these fixed operators, and helps lowering everything to execution plans as needed (multi-level specification).

5.2 Compiler and Runtime Extensibility

DaphneIR: A developer-centric direction for extensibility is the extension of our DaphneIR dialect. Common use cases are adding new operations of an existing category (e.g., a new unary elementwise operation), adding a new category of operations (e.g., a specific quaternary operator), and adding new traits (e.g., as help for new optimization passes). Additionally, developers may add existing or new MLIR dialects and integrate them with the rest of the system by changing DAPHNE internally. That way, we will be able to benefit from the rich

ecosystem of dialects provided by the community. While some of these extensions can reuse most of the existing runtime operations, other require additional runtime kernels.

Compilation Chain: MLIR's approach of applying a sequence of optimization passes is already very modular and can reuse existing LLVM and MLIR passes. Adding new optimization passes or recomposing existing optimization passes into custom compilation chains is a natural direction for compiler extensibility. For example, having registered a new data type or kernel, an additional optimization pass may apply and choose these data types or kernels a given IR program under certain conditions.

Sideways Entry in Multi-Level Compilation: Normally, the invocation of `daphne` (by a user or through the Python API) takes a DaphneDSL script and then compiles and executes this script. In order to allow for debugging and understanding, the plan explanation tools described in Section 3.1 allow to print the DaphneIR at different states of compilation. We will extend `daphne` to accept valid DaphneIR instead of DaphneDSL as program specification. This flexibility allows researchers to obtain the generated execution plan, modify the plan slightly (e.g., to force certain sequences of local or distributed operations), and execute this plan through `daphne`, which performs the remaining lowering and runs the final executable plan. This route allows much more fine-grained modifications than the means for extensibility at script level described above, but should only be used if cases where the latter to not suffice.

Custom Kernels and Data Types: Users can implement new kernels for existing DaphneIR operations (e.g., to target a new hardware accelerator) by following a clearly defined C interface which is directly derived from the definition of the DaphneIR operation. Within such a kernel function, users have a high degree of freedom regarding implementation details, and only need to follow simple rules such as respecting the system's memory management. Likewise, new data types can be created by subclassing DAPHNE's existing frame and matrix types. Such new data types could immediately be used by existing kernels tailored for the super-classes, which are based on a generic `get/set` interface for individual elements. Alternatively, users may add custom kernels to process their new data type more efficiently. In general, we try to give users as much flexibility as possible, while providing useful library-style tools to simplify common performance engineering cases (e.g., lookup tables for converted values).

Advanced Scheduling Techniques: DAPHNE will also offer extension hooks for custom scheduling algorithms. Applying these in the local or distributed runtime will be possible by means of configuration at DSL level or via global configuration files.

6 Refinements to the Initial System Architecture

In Sections 3–5, we have focused on describing the refined system architecture as a whole. In this section, we explicitly highlight the development since the initial system architecture, which was presented in deliverable D2.1 [D2.1]. The initial system architecture already provided an overview of the architecture and described the most important components, including the MLIR-based compiler, local and distributed runtime, local and distributed data structures, selected aspects of HW accelerator and storage integration, as well as the vectorized execution engine and its relation to multi-device scheduling and code generation. During the

implementation of the DAPHNE prototype and the more detailed design of individual components of the architecture as presented in deliverables D3.1 [D3.1], D4.1 [D4.1], D5.1 [D5.1], and D7.1 [D7.1], the design decisions of the initial system architecture have turned out to work very well. Thus, we largely retain the initial system architecture, but refine and update it a several points. In the following, we describe the most important refinements we applied, and motivate them with lessons learned during our work with the initial system architecture.

More comprehensive overview of the system architecture. The architecture overview in Section 3.1 now captures the entire stack from the user frontend (which is mainly a summary of parts of deliverable D3.1 [D3.1]), over the compiler and runtime, to the deployment environment. For the first time, we also comment on tooling around this core of the system. The two examples we included are directly derived from early lessons learned while dealing with the architecture and prototype. Plan explanations are important while debugging and for sideways entry into the compilation chain, while monitoring and profiling are a requirement for more advanced adaptive scheduling techniques as well as for conducting micro benchmarks.

Vectorized engine as the central means for parallelism. Initially, we intended to use the vectorized engine only for the local runtime (multi-thread and multi-device). However, during the design and implementation of the distributed runtime, we realized that it can be seamlessly integrated into the framework of the vectorized engine: fused pipelines define the tasks for distributed workers and distribution primitives like broadcast, scatter, and all-reduce have conceptual counterparts in the split/combine steps of the local vectorized execution. Thus, defining the vectorized engine as the central means for parallelism allows us to share compiler and, partly, runtime infrastructure for the local and distributed parallel execution. In that context, the representation of interesting data properties, most importantly the data placement, plays a central role now, since it is required for the local vectorized as well as the distributed execution. Consequently, metadata and placement are now centrally discussed in Section 3.2 on the data representation, while they were previously only discussed in the context of HW accelerators.

Prioritization of extensibility. Extensibility is a major goal of DAPHNE. To underline this, we dedicate an entire section (Section 5) to this cross-cutting aspect of the architecture. We summarize the most important points on extensibility that have previously been presented in deliverable D3.1 [D3.1], but also extend upon that, e.g., by commenting on how to add new kernels and data types.

More details on various components. Furthermore, we updated the description of the local and distributed runtime (Section 3.3) with early lessons learned regarding the pre-compilation of kernels, deeper insights into the co-existence of different distributed backends, and the design of a component for memory management and garbage collection. Moreover, in the context of accelerators and storage (Section 3.4), we clarified and further justified the way we implement and integrate device kernels. Due to its outstanding importance for the architecture, we decided to include a summary of the design presented in deliverable D7.1 [D7.1].

7 Conclusions

We described the refined overall architecture and key design decisions of the DAPHNE system infrastructure as an open and extensible system for integrated data analysis pipelines, comprising query processing, ML, and HPC. Major aspects are a domain-specific language for linear and relational algebra, an MLIR-based compilation chain, frame and matrix representations, HW accelerators and computational storage, multi-level scheduling, and a vectorized execution engine that allows for fine-grained fusion and parallelism across these heterogeneous components.

Preliminary experiments with selected ML pipelines on CPUs, GPUs, and FPGAs show promising results. In the next few years, we will continue building out this infrastructure and tackling research challenges across the different levels from resource management, device kernels, I/O and buffer management, and vectorized execution, over compilation, operator and pipeline scheduling, to seamless extensibility and customization for IDA pipelines.

7 References

[A17] S. N. M. Albarqouni. Machine Learning for Biomedical Applications: From Crowdsourcing to Deep Learning. PhD thesis, Technische Universitaet Muenchen, 2017.

[A+16] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. OSDI, 2016.

[A+19b] S. Agrawal et al. Machine Learning for Precipitation Nowcasting from Radar Images. CoRR, abs/1912.12132, 2019.

[B+16] B. Burns et al. Borg, Omega, and Kubernetes. Commun. ACM 59(5), 2016.

[B+17] D. Baylor et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. SIGKDD, 2017.

[B+21] S. Baunsgaard et al. ExDRa: Exploratory Data Science on Federated Raw Data. SIGMOD, 2021.

[BA+20] M. Boehm et al. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. CIDR, 2020.

[BD21] A. Barbalace and J. Do. Computational Storage: Where Are We Today? CIDR, 2021.

[BDS14] S. Bhattacherjee, A. Deshpande, and A. Sussman. PStore: an efficient storage framework for managing scientific data. SSDBM, 2014.

[BH+07] K. S. Beyer et al. On synopses for distinct-value estimation under multiset operations. SIGMOD, 2007.

[BK+12] Jeff Bezanson et al. Julia: A Fast Dynamic Language for Technical Computing. CoRR abs/1209.5145, 2012.

[BKS20] S. Blanas, P. Koutris, and A. Sidiropoulos. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. CIDR, 2020.

[C+15] P. Carbone et al. Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 2015.

[C+19] C. Curino et al. Hydra: a federated resource manager for data-center scale analytics. NSDI, 2019.

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

[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.

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

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

[D6.1] DAPHNE: D6.1 Report on Search Space Analysis, Automatic Capability Configuration, EU Project Deliverable, 11/2021.

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

[D8.1] DAPHNE: D8.1 Initial Pipeline Definition All Use Cases, EU Project Deliverable, 08/2021.

[D+16] B. Dageville et al. The Snowflake Elastic Data Warehouse. SIGMOD, 2016.

[D+22] P. Damme et al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. CIDR, 2022.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering, 5(1), 1998.

[H+11] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. NSDI, 2011.

[H+16] K. He et al. Deep Residual Learning for Image Recognition. CVPR, 2016.

[H+16b] K. He et al. Identity Mappings in Deep Residual Networks. ECCV, 2016.

[HM+20] Charles R. Harris et al. Array programming with NumPy. Nat. 585, 2020.

[JP13] R. Johnson and I. Pandis. The bionic DBMS is coming, but what will it look like? CIDR, 2013.

[KBY17] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. SIGMOD, 2017.

[L+14] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. SIGMOD, 2014.

[LA+21] C. Lattner et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO, 2021

[LB21] A. Lerner and P. Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. SIGMOD, 2021.

[Le+14] R. Leißa et al. Sierra: a SIMD extension for C++. WPMVP@PPoPP, 2014.

[Li12] Don Lipari. The SLURM Scheduler Design. User Group Meeting, 2012.

[M+11] A. Munshi et al. OpenCL programming guide. Pearson Education, 2011.

[M+18] P. Moritz et al. Ray: A Distributed Framework for Emerging AI Applications. OSDI, 2018.

[MH+12] Floréal Morandat et al. Evaluating the Design of the R Language - Objects and Functions for Data Analysis. ECOOP, 2012.

[MNS09] G. Moerkotte, T. Neumann, G. Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. PVLDB 2(1), 2009.

[P+16] H. Pirk et al. Voodoo – A Vector Algebra for Portable Database Performance on Modern Hardware. PVLDB 9(14), 2016.

[P+18] S. Palkar et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. PVLDB 11(9), 2018.

[P+19] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS, 2019.

[P+21] T. Pfaff et al. Learning Mesh-Based Simulation with Graph Networks. ICLR, 2021.

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

[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.

[RL16] R. Reyes and V. Lomüller. SYCL: Single-source C++ accelerator programming. In Parallel Computing: On the Road to Exascale (pp. 673-682). IOS Press, 2016.

[RM20] M. Raasveldt and H. Muehleisen. Data Management for Data Science - Towards Embedded Analytics. CIDR, 2020.

[S94] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994.

[S+00] A. S. Szalay et al. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. SIGMOD, 2000.

[S+11] M. Stonebraker et al. The Architecture of SciDB. SSDBM, 2011.

[S+19] N. K. Srivastava et al. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. FCCM, 2019.

[SB+19] J. Sommer et al. 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.

[SO12] I. D. Stewart and T. R. Oke. Local Climate Zones for Urban Temperature Studies. Bulletin of the American Meteorological Society, 93(12), 2012.

[U+20] A. Ungethüm et al. Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. CIDR, 2020.

[V+13] V. K. Vavilapalli et al. Apache Hadoop YARN: yet another resource negotiator. SoCC, 2013.

[Z+12] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI, 2012.

[Z+19] A. Zamuda et al. Forecasting Cryptocurrency Value by Sentiment Analysis: An HPC-Oriented Survey of the State-of-the-Art in the Cloud Era. High-Performance Modelling and Simulation for Big Data Applications, 2019.

[Z+19b] X. X. Zhu et al. So2Sat LCZ42: A Benchmark Dataset for Global Local Climate Zones Classification. CoRR, abs/1912.12171, 2019.

[Z+20] X. X. Zhu et al. So2Sat LCZ42: A Benchmark Data Set for the Classification of Global Local Climate Zones [Software and Data Sets]. IEEE Geoscience and Remote Sensing Magazine, 8(3), 2020.

[ZC+12] Matei Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI, 2012.