DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines

Patrick Damme[1], Marius Birkenbach[10], Constantinos Bitsakos[6], Matthias Boehm[1], Philippe Bonnet[9], Florina Ciorba[12], Mark Dokter[1], Pawel Dowgiallo[8], Ahmed Eleliemy[12], Christian Faerber[8], Georgios Goumas[6], Dirk Habich[11], Niclas Hedam[9], Marlies Hofer[2], Wenjun Huang[3], Kevin Innerebner[1], Vasileios Karakostas[6], Roman Kern[1], Tomaž Kosar[13], Alexander Krause[11], Daniel Krems[2], Andreas Laber[7], Wolfgang Lehner[11], Eric Mier[11], Marcus Paradies[3], Bernhard Peischl[2], Gabrielle Poerwawinata[12], Stratos Psomadakis[6], Tilmann Rabl[5], Piotr Ratuszniak[8], Pedro Silva[5], Nikolai Skuppin[3b], Andreas Starzacher[7], Benjamin Steinwender[10], Ilin Tolovski[5], Pınar Tözün[9], Wojciech Ulatowski[8], Yuanyuan Wang[3b], Izajasz Wrosz[8], Aleš Zamuda[13], Ce Zhang[4], Xiao Xiang Zhu[3b]

[1] Know-Center GmbH/TU Graz, Austria; [2] AVL List GmbH, Austria; [3] DLR, [3b] DLR/TU Munich, Germany; [4] ETH Zurich, Switzerland; [5] HPI/Uni Potsdam, Germany; [6] ICCS/NTUA, Greece; [7] Infineon, Austria; [8] Intel, Poland; [9] ITU Copenhagen, Denmark; [10] KAI GmbH, Austria; [11] TU Dresden, Germany; [12] University of Basel, Switzerland; [13] University of Maribor, Slovenia

ABSTRACT

Integrated data analysis (IDA) pipelines—that combine data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring—become increasingly common in practice. Interestingly, systems of these areas share many compilation and runtime techniques, and the used—increasingly heterogeneous—hardware infrastructure converges as well. Yet, the programming paradigms, cluster resource management, data formats and representations, as well as execution strategies differ substantially. DAPHNE is an open and extensible system infrastructure for such IDA pipelines, including language abstractions, compilation and runtime techniques, multi-level scheduling, hardware (HW) accelerators, and computational storage for increasing productivity and eliminating unnecessary overheads. In this paper, we make a case for IDA pipelines, describe the overall DAPHNE system architecture, its key components, and the design of a vectorized execution engine for computational storage, HW accelerators, as well as local and distributed operations. Preliminary experiments that compare DAPHNE with MonetDB, Pandas, DuckDB, and TensorFlow show promising results.

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 [4], natural sciences, remote sensing, transportation, health-care, and finance [88], 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 [3, 66] and data analysis of simulation outputs [12]. These complex end-to-end analysis requirements create a trend towards *integrated data analysis (IDA) pipelines* that jointly utilize 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 [24, 71]; flexible data-parallel computation frameworks like Spark [87], Flink [20], or Dask [73]; distributed ML systems like TensorFlow [1] or PyTorch [65]; 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 [83], decades of data mining and advanced analytics, in-DBMS ML [50], array databases like SciDB [82], and more recently, data management around ML systems (e.g., TensorFlow TFX [9]), and HPC-inspired (e.g., topology-aware) data management and query processing [13]. However, an open system infrastructure for seamlessly developing and running IDA pipelines is still missing, and at the same time, new challenges emerge.

HW Challenges: Interestingly, data management, HPC, and ML systems share many compilation and runtime techniques; and together stress every hardware 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 [46] 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

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

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.

Contributions: The DAPHNE project[1] 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 [51] 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 paper, 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 requirements and opportunities in Section 2.
- *System Architecture:* Subsequently, we describe the overall MLIR-based system architecture (Section 3), language abstractions (Section 4), and selected compiler and runtime components (Section 5), including aspects of extensibility.
- *Vectorized Execution:* We further introduce our central, vectorized (a.k.a. tiled) execution engine for fused pipelines of frame and matrix operations, which can utilize heterogeneous HW devices, and computational storage in Section 5.5.
- *Experiments:* Finally, we share promising preliminary results of an *early prototype*, comparing DAPHNE on four basic IDA pipelines with baseline systems in Section 6.

2 IDA PIPELINES

Integrated data analysis pipelines are complex, often multi-phase workflows of ETL (extraction transformation loading), ML training/scoring, numerical computation, simulations, and data analysis. In order to raise awareness of this trend towards IDA pipelines, we briefly describe representative DAPHNE use cases, and summarize requirements and opportunities of such IDA pipelines.

2.1 Example Use Cases

We describe selected use cases—representative for a broader range of applications—from earth observation, semiconductor manufacturing, and vehicle development.

Earth Observation (DLR): Local climate zones (LCZs) classification categorizes patches of satellite images for modeling climate-relevant surface properties (e.g., surface imperviousness and structure) [80, 89]. 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 [91], the DLR team materialized and published a labeled dataset, called `So2Sat LCZ42` [89, 90] 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 expert 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 [41, 42] 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 on the global change process of urbanization).

Semiconductor Manufacturing (Infineon): Ion implantation changes the physical, chemical, and electrical properties of target substrates such as silicon wafers. Special equipment accelerates dopants in an electric field onto these wafers. Every recipe change requires ion-beam tuning to meet specific requirements. In order to avoid expensive timeouts (after 15min) of unsuccessful tuning, a prediction model estimates the tuning success. For data preparation, raw implant log files are scanned, parsed, and stored in a multi-table database. These tables are joined and exported—with different time horizons—into CSV-files of 79 categorical and 2,468 numerical features. For model training, the data is pre-processed by removing low-variance and highly-correlated features, train/test splitting, missing value imputation, one-hot encoding, and normalization. Finally, a random forest classifier is trained via extensive hyper-parameter tuning and cross validation, and evaluated by F1 measure, AUC (area under the ROC curve), and correlation measures.

Material Degradation (KAI): For investigating degradation of power semiconductors, accelerated stress tests are performed for different devices under test. Many devices are tested simultaneously and the results are stored as waveforms (time series of voltage and current measurements). Over the past years, KAI stored millions of electrical signals per semiconductor technology, stored as TDMS files. Using physics-based models, the microscopic degradation in the thin metal layers are simulated and lifetime under different parameters of mechanical stress are estimated and serve as input for the subsequent reliability analysis and degradation modeling. The analysis pipeline includes reading the TDMS files, slicing out pulses, computing the power waveform, waveform line simplification for data reduction with bounded error, and ingestion into a databases via a REST API; the simulation component then reads from this database, performs FEM (finite element method) simulations in a separate HPC cluster, and stores the results back.

Vehicle Development (AVL): In the context of automotive vehicle development, we focus on two specific use cases. First, in PEM and SOFC fuel cells, a hydrogen (H2) flow gets supplied from a fuel tank and accelerated via a nozzle. In a mixing chamber of the ejector, this primary and recirculated streams are mixed. Depending on the geometric variables of the ejector and the operating conditions, a certain entrainment ratio of recirculated and fresh flows, and suction pressure can be achieved. The ejector geometry optimization iteratively predicts geometry variants (7 key and 17 side design parameters) via a behavioral model (trained via non-linear or weighted least-squares) [28], and performs CFD (computational

[1]Project website: https://daphne-eu.eu/. The initial open source release of the DAPHNE system prototype is planned for 03/2022.

fluid dynamics) simulations on the ejector mesh for verification and result post-processing. This process is repeated until the target properties and constraints are met. Besides mesh simplification and parallelization, the challenge is to minimize expensive CFD simulations, which shares characteristics with neural architecture search [68, 92]. Second, during vehicle development, many target key performance indicators (e.g., fuel consumption, vehicle mass, aerodynamic drag) are predicted, and repeatedly validated via simulations and hardware-in-the-loop tests. Consolidating the heterogeneous metadata and measurements along this process allows to build prediction models for individual KPIs, interactions among KPIs, and maturity/confidence of KPIs. The current analysis pipeline collects data in JSON files, and trains GPR (Gaussian process regression) models [31], but the main challenge is collecting these heterogeneous measurements and simulation data.

2.2 Requirements and Opportunities

Main characteristics of IDA pipelines are the composition of complex workflows of data extraction and pre-processing, ML training and scoring (with multiple models), numerical computation and simulations; human intervention and large project teams; as well as query processing of inputs, intermediates, and simulation outputs. Related emerging application trends include data cleaning during exploratory query processing (e.g., imputation and deduplication) [5, 19], ML-assisted data cleaning and augmentation [23, 25, 29], ML-assisted simulation [67], graph extraction and processing pipelines [76], and increasing use of vectorized array programs for graph- and tree-based problems [61, 75]. The following list identifies key requirements on system infrastructure supporting such IDA pipelines:

- Seamless high-level APIs and DSLs (DM, HPC, ML; operations and primitives; mini-batch and batch; SQL)
- Interoperability through common frame/matrix operations
- Extensible infrastructure (data types, kernels, metrics, scheduling) with externalized multi-level compilation
- Local, distributed, and out-of-core datasets; with dense, sparse, and irregular (ragged, nested) formats, and support for heterogeneous, multi-modal input data formats
- Integration with resource management, programming models, and specialized DM, HPC, ML libraries
- Awareness and utilization of storage hierarchies, computational storage, and heterogeneous accelerators
- Fine-grained operator fusion and parallelism, with code generation and data/plan partitioning across devices and nodes

Opportunities: 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; holistic optimization; as well as improved scheduling and resource utilization in shared cluster environments.

3 SYSTEM ARCHITECTURE

DAPHNE is an open and extensible system infrastructure for developing and executing IDA pipelines. In this section, we share the design of the overall system architecture and its key components.

The DAPHNE system architecture is shown in Figure 1. DAPHNE is built from scratch in C++, but utilizes MLIR [51] as a multi-level,

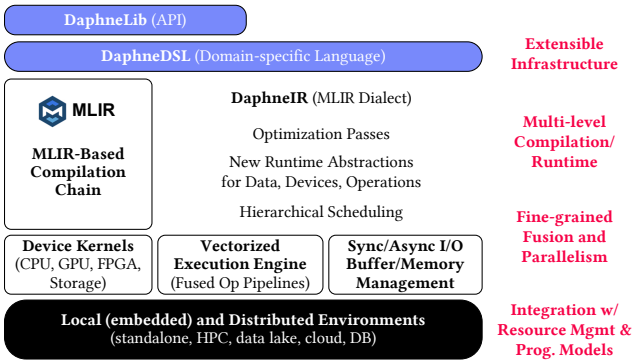


Figure 1: DAPHNE System Infrastructure.

LLVM-based intermediate representation (IR) as well as existing runtime libraries such as BLAS, LAPACK, DNN kernels and collective operations. These libraries are augmented with more specialized, custom kernel implementations. Users specify their IDA pipelines in DaphneDSL (a language similar to Julia, PyTorch, or R) or DaphneLib (a Python API with lazy evaluation that creates DaphneDSL well). These DSL programs are then compiled—via a multi-level compilation chain—into executable runtime plans.

Extensibility: A major design decision is the focus on an extensible infrastructure allowing the registration of new data types, kernels, and scheduling algorithms in predefined extension hooks. Extensibility goes beyond recent work on combining variants (variability) of communication primitives [34]. We further allow sideway entries into the multi-level compilation chain for enforcing certain physical data types and kernels. These enforced physical properties are treated as constraints and the optimizing compiler respects and works around them. In contrast to purely declarative interfaces, this multi-level abstraction can simplify experimentation and system extensions while providing data independence and automatic optimization for unconstrained scripts.

Compilation Chain: The DaphneDSL scripts are converted by an ANTLR parser into DaphneIR, an MLIR dialect comprising control flow (via MLIR dialect SCF), frame/matrix data types, and operations. Additionally, we aim to integrate parsers for SQL and existing ML DSLs. After parsing, we apply various MLIR optimization passes such as common-subexpression elimination, and code motion, but also new passes such as algebraic simplifications, inference of interesting properties [44], and cost-based optimization. A common infrastructure also enables cardinality and sparsity estimation in a holistic manner. In contrast to other MLIR dialects, we lower frame and matrix operations to C++ kernels and use LLVM mainly for control flow and scalar operations.

Runtime Environment: During 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 instruction-level parallelism. Our vectorized execution engine further provides means of operator fusion, and a seamless integration of heterogeneous computing devices, computational storage, and distributed operations.

4 LANGUAGE ABSTRACTIONS

Common Abstractions: Tuning IDA pipelines for emerging hardware and changing data characteristics requires substantial manual effort and is often unsustainable in practice. Orchestrating IDA pipelines with specialized systems reduces the effort but causes overhead for boundary crossing (e.g., materialization of intermediates), static resource allocation (temporal and/or spatial underutilization), and lacks the ability of optimization and redundancy elimination (data and computation) if IDA pipeline primitives are mixed in repetitive or iterative computations. A necessary requirement for an open and extensible system infrastructure for entire IDA pipelines is a common language abstraction. Data management, HPC, and ML systems rely upon well-established language abstractions: in data management we have SQL, UDFs, procedures, and recently data frame abstractions; in HPC the focus is on application codes, programming models like OpenMP, MPI, and stencil computation, as well as libraries for numerical computation and simulation (e.g., FEM, CFD); in ML systems, we have UDF-based systems, linear algebra programs, and higher-level abstractions for DNNs, features, and ML algorithms. Despite these very different language abstractions, programming models, and objectives, there is common ground: all of these systems fundamentally work with combinations of data frames, and matrices/tensors.

Design Principles: The following design principles govern the individual design decisions on DAPHNE language abstractions:

- *Frame & Matrix Operations:* Many ML algorithms, query processing, and numerical computation can be expressed via frame and matrix operations. These coarse-grained operations preserve their semantics, and simplify the parallelization and lowering to kernels for emerging hardware.
- *Data Independence:* Instead of requiring users to specify data representations like dense, sparse, and compressed; or data locations like local CPU, local GPU, or distributed; users work with abstract data types and the system optimizes the IDA pipeline for data and deployment characteristics. This principle is crucial for composite DSL-based primitives.
- *Extensibility:* Data independence and automatic optimizations are great aspirations but face challenges regarding extensibility for new operations, data types, and hardware. Given increasing specialization across the SW and HW stack, DAPHNE aims for good extensibility to allow researchers to quickly experiment with new prototypes and extensions.

4.1 DaphneDSL: A Domain-specific Language

Overview: DaphneDSL is a domain-specific language inspired by ML systems and languages/libraries for numerical computation like Julia [11], Python NumPy [40], R [59], and SystemDS DML [17]. At a high-level, this DSL supports conditional control flow, typed and untyped functions; abstract data types of frames, matrices, and scalars; various built-in operations (i.e., functions and operators), and additional second-order language abstractions. A user creates a simple text file with a DSL program (e.g., `example.daphne`) and can parse, compile, and execute this program via `daphnec example.daphne`. DaphneDSL is parsed into DaphneIR using ANTLR4, for which we provide a DSL grammar file and generate the respective parser in an offline manner on grammar updates.

Table 1: Basic Frame/Matrix Built-in Operations.

Operation Group	Examples
Matrix Multiply	gemm, syrk, conv2d
Element-wise Ops	unary, binary, ternary, n-ary
Aggregation and Stats	sum(), rowSums(), colSums(), median()
Data Generation	random data, sequences, sampling
DNN Layers/Optimizers	lstm(), softmax(), sgd, adam, adagrad
Matrix Reorganization	transposition, slicing and insert, reshaping
Frame Reorganization	dedup, sorting, renaming, casting
Set Operations	union, intersect, diff, cartesian product
Relational	selection, projection, join, group-by
Read and Write	csv, matrix market, parquet, arrow, hdf5

Data and Value Types: DaphneDSL differentiates data and value types. Supported data types include frames (a table with columns of potentially different value types), matrices (homogeneous value type) and scalar values, but in the future we will likely extend this by tensors and named/unnamed lists to group instances of such data types and access by name and/or position (e.g., `params["lr"]`, or `params[7]`). Value types specify the representation of individual values and currently we support: `SI8`, `SI32`, `SI64`, `UI8`, `UI32`, `UI64`, `FP32`, `FP64` (integer and floating point). In addition, we also support strings but currently only for scalars. The combination of data and value types yields powerful data representations such as `Matrix<FP32>` and `Frame<SI32, SI8, FP64>`.

Basic Built-in Operations: The supported frame and matrix operations include both relational algebra and linear algebra as well as various aggregation and statistical functions. In detail, there are 100s of relevant operations in the categories shown in Table 1. At language level, these operations process abstract data types.

Example 1. ***Example DSL Program:*** *The following DaphneDSL program computes the connected components of a co-author graph.*

```
G = readCOO("./AuthorCOO.csv");  // n-by-n boolean matrix
n = nrow(G);     // get the number of vertexes
maxi = 100;
c = seq(1, n);   // init n-by-1 matrix of vertex IDs
diff = inf;      // init diff to +Infinity
iter = 1;
// iterative computation of connected components
while(diff>0 & iter<=maxi) {
  u = max(rowMaxs(G * t(c)), c); // neighbor propagation
  diff = sum(u != c);            // # of changed vertexes
  c = u;                         // update assignment
  iter = iter + 1;
}
```

We read a CSV file in coordinate format of row indexes, column indexes, and values (ones) into an expanded (likely sparse) matrix, where each row and column (i.e., vertex or node) refers to an author and non-zero cells (i.e., edges) refer to co-author relationships. We then initialize the state of each vertex with a unique ID and iteratively propagate the current state to all neighbors (here, `G*t(c)` performs a matrix/row-vector element-wise multiplication with broadcasting of the transposed vector). The new vertex states are computed as the maximum IDs received from neighbors and the current state [56]. That way, the maximum vertex ID per component propagates through the entire subgraph, and once a fixpoint is reached, we terminate and obtain the assignment of nodes to connected components.

Control Flow and Function Calls: For expressiveness, we also support conditional control flow with loops, branches, and function calls. The basic control flow constructs are mapped to the existing MLIR dialect SCF (structured control flow). We support for, while-do, and do-while loops, but will further add parfor (parallel for) loops [14, 35, 77] as an entry point for parallelization. Branches use an if-elseif-else syntax, and both loops and branches allow for arbitrary nesting levels. In order to build a hierarchy of DSL-based primitives, we aim to support—inspired by Python type hints, Julia type assertions, and Rust function signatures—both typed and untyped functions. For example, consider making the DSL script for connected components a built-in function `components()`:

```
def components(G, maxi, verbose) { ... }
def components(G: matrix<ui1>, maxi: ui32, verbose: ui1)
  -> matrix<ui64> { ... } // one output matrix of UI64
```

These alternatives provide good flexibility (untyped functions are compiled on demand according to types at a call site) as well as typing (for data and/or value types) if the types are known during development. For example, the function `components()` assumes Boolean input graphs (1 bit integers) and can make this explicit. Multiple function returns also require multi-assignments such as `X,Y = foo(Z)`. Arguments are passed by position or name, were the latter allows arbitrary argument orderings and defaults. For both DSL-based functions and built-in operations, we aim to allow, similar to Julia [10], multiple dispatch where function calls are dispatched to the most specific type combination of inputs. Only top-level function declarations are supported, but namespaces allow the packaging of function libraries without conflicts.

Scoping and Type Polymorphism: Related to control flow, we use bounded scoping from traditional programming languages. If an intermediate variable is created in a certain nested scope, it is deleted at the end of this scope. This scoping is in contrast to R's unbounded scoping, which is useful in the absence of variable declarations. However, via simple matrix/frame constructors, we can easily overcome the need for type declarations. Variable shadowing is not supported, so an assignment of X overwrites the outer scope's variable but in a function-local manner. Furthermore, DaphneDSL has copy-on-write semantics by default, where assignments like `A=B`, and function calls are copy-by-reference, but any modification like `B[i,]=C`, implicitly copies B, performs the partial update and assigns the new intermediate to B, while A remains unmodified. This approach is equivalent to R's copy-on-write semantics, whereas other languages like Julia use update-in-place by default and require users to perform explicit `A=copy(B)` operations if implicit updates to multiple objects are unintended. Internally, the DAPHNE compiler and runtime then help to avoid unnecessary copies (e.g., via update-in-place flags and/or reference counting). Finally, DaphneDSL provides limited type polymorphism in terms of non-polymorphic data types but polymorphic value types. This approach excludes variable assignments of conflicting data types and simplifies runtime plan generation.

Higher-level Built-in Operations: Besides the basic built-in operations, we further aim to provide higher-level built-in operations. This includes DSL-based functions and second-order functions. DSL-based functions are composite functions (e.g., ML algorithms or DNN layers) written in DaphneDSL that are registered in packages, which can be imported in other DaphneDSL scripts. Second-order functions take functions as arguments, and include built-in functions for executing SQL queries on registered frames, primitives like parameter servers for data-parallel mini-batch training, and user-defined functions with different data bindings:

```
// (a) SQL query processing
registerView("XTab", X);           // X:= [SI32, SI8, FP64]
Y = sql("SELECT DISTINCT a, b FROM XTab"); // Y:= [SI32, SI8]
// (b) primitives for mini-batch training
Mp = paramserv(model=M, features=X, labels=y,
               upd=updateGrad, agg=updateModel, utype=ASP,
               freq=BATCH, epochs=200, batchsize=128, ...);
// (c) user-defined functions (axis: 0 cell, 1 row, 2 column)
Y = map(X, foo);      // DSL function foo on every cell of X
Y = map(X, "v -> v.length + 1");  // C++ with pre-defined env
```

These primitives are either compiled to frame and matrix operations, or are mapped to dedicated infrastructure that repeatedly calls the passed function arguments. For example, the parameter server [27] (or similar distributions strategies [36]) establishes temporary workers, repeatedly runs gradient and model updates and, after termination, returns the model and thus, acts as a stateless function.

External Libraries: The integration of external libraries is very important for enabling an incremental adoption of DAPHNE. This integration focuses on two main aspects, low overhead data exchange with UDFs and libraries, as well as exposing the tuning knobs (e.g., degree of parallelism) of libraries to optimization and hierarchical scheduling of IDA pipelines. While the basic integration is through UDFs with materialized intermediates, we aim to support zero-copy data formats like Apache Arrow, chunked data transfers as known from R-Integrations [26, 39], and careful tuning of buffer management [74]. Additionally, we aim to annotate the called UDFs in special script-level scopes to expose tunable parameters such as the degree of parallelism, and other MPI/OpenMP/BLAS library configurations. Exposing and utilizing these parameters during optimization will allow for seamless scheduling and better utilization of hardware resources in complex, composite IDA pipelines.

4.2 DaphneLib: A Python API

Overview: Python is currently undoubtedly the main entrance to ML systems, but increasingly often also to query processing and numerical computations. Accordingly, we provide DaphneLib as a simple user-facing Python API that allows calling individual basic and higher-level DAPHNE built-in functions. The overall design follows similar abstractions like PySpark [87] and Dask [73] by using lazy evaluation, but on evaluation, creates and executes a DaphneDSL script, reusing the entire compilation chain.

Lazy Evaluation: The entry point for DaphneLib is a DaphneContext that can create DAPHNE matrices or frames from pandas data-frames or NumPy arrays. These matrices and frames are essentially metadata objects with references to leaf data. Subsequent operations are directly invoked on these metadata objects. For example, our `components()` function can be called as follows:

```
dc = DaphneContext()
G = dc.from_numpy(npG)
G = (G != 0)
c = components(G, 100, True).compute()
```

While Spark differentiates transformations and actions (where actions trigger computation), Dask [73] provides an explicit compute()

function. In order to make our API easily understandable, we follow this design of explicit triggers. In the example above, we create a DAPHNE matrix from a NumPy array, convert it to a Boolean matrix (`G!=0`), and call the `components()` function. All these operations only build a local dependency graph of operations, where metadata objects refer to data or operation nodes, and subsequent operations extend the dependencies of their inputs. On `compute()`, we then traverse the dependency graph in a depth-first manner and construct a DaphneDSL script. Each node adds a line of DSL script, and the depth-first traversal with memorization ensures an ordering by data dependencies without unnecessary redundancy if a node is reachable over multiple paths. The resulting DaphneDSL script is parsed, compiled, and executed through daphnec, and the results are converted to NumPy arrays or pandas data-frames. In addition to all basic and high-level built-in functions and operations, we also provide a function for executing DaphneDSL directly. Given this seamless integration, users can then mix and match DAPHNE computations with other Python libraries.

4.3 Extensibility

The design principle of extensibility is of utmost importance to enable low-effort exploratory experimentation and custom extensions for new data types, operations, and hardware. At language and configuration level, there are multiple aspects of extensibility, all of which require a discussion of different personas and deployments. DAPHNE aims for deployments with different distribution strategies, different distributed computing frameworks, different resource managers, as well as different on-premise and cloud hardware resources. In this context, we see the personas of internal/external developers, users, and infrastructure administrators.

Extension Catalog: Our initial design centers around an extension catalog that allows registering dedicated artifacts in the form of shared libraries. The catalog also registers the type of extension (e.g., kernels for existing operations, or data types), traits and properties, as well as cost functions provided by developers. Based on this metadata, these extensions are represented in DaphneIR and thus, included in various optimization passes such as shape inference, and operator selection. The concrete use of extensions can be further influenced both at script level (mostly by users) or configuration files (mostly by administrators).

Extensibility at DSL Level: At DaphneDSL level, we will provide means for programmatically obtaining and setting configurations and topology information (e.g., get/setNumThreads(32)). These configurations include available devices, degree of parallelism, memory budgets, but also different garbage collection and scheduling algorithms. Depending on the place of invocation, these configurations affect the default configuration for the current scope and child-scopes (e.g., called functions). Script-level extensibility also includes dedicated built-in functions for affecting data representations, data placement, and operator placement:

```
X = sparse(Y);
X = compress(Y);
X = device(Y, "/GPU:0");
X = device(Y, ["/GPU:0","/GPU:1"], round_robin);
X = Y @_gpu Z; // matmult on GPU
```

All these decisions are made via basic built-in functions. This approach provides clear data-flow semantics and allows extensibility. For example, a developer might create a new compressed data type and operations. By registering and invoking a shared library with a custom `compressXYZ()` operation, the data can be brought into this representation, and subsequent function calls on this data object are then dispatched to the specialized operator implementations. Such script-level decisions lose data dependence 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.

5 COMPILER AND RUNTIME

As shown in Figure 1, the DaphneDSL scripts are then compiled into executable runtime plans. In this section, we give a more detailed overview of the compiler and selected runtime components.

5.1 Compiler Overview

MLIR Background: The ANTLR parser converts a DaphneDSL program into an MLIR-based intermediate representation. MLIR [51] is a customizable compiler infrastructure for creating low-cost domain-specific compilers. Programs are represented in static single assignment (SSA) form—permitting only a single assignment to otherwise immutable variables—and then lowered to LLVM. Its basic concepts are modules, functions, and regions that can contain sequences of blocks, which in turn contain sequences of operations. The basic philosophy is that everything, even loops with complex body programs, are operations and everything is customizable. Using MLIR allows for reuse of basic infrastructure and various optimization passes as a library, as well as future extensibility by other MLIR dialects. For control structures like branches and loops, we already use the SCF (structured control flow) dialect.

DaphneIR: The DAPHNE MLIR dialect, called DaphneIR, defines the types, operations, and various traits (e.g., for schema, type, and shape inference) in so-called TableGen [84] records, from which C++ code is automatically generated. Operations produce values of a certain type. For example, the following snippets show the TableGen specifications of basic scalar value types, the matrix multiplication operation, and the shape inference interface:

```
def SIScalar : AnyTypeOf<[SI8, SI32, SI64], "signed int">;
def UIScalar : AnyTypeOf<[UI8, UI32, UI64], "unsigned int">;
def IScalar : AnyTypeOf<[SIScalar, UIScalar], "int">;
def FPScalar : AnyTypeOf<[F32, F64], "float">;
def NumScalar : AnyTypeOf<[IScalar, FPScalar], "numeric">;
def Daphne_MatMulOp : Daphne_Op<"matMul", [
  DeclareOpInterfaceMethods<VectorizableOpInterface>,
  NRowsFromIthArg<0>, NColsFromIthArg<1>
]> {
  let arguments = (ins MatrixOf<[NumScalar]>:$lhs, ...rhs);
  let results = (outs MatrixOf<[NumScalar]>:$res);
}
def InferShapeOpInterface : OpInterface<"InferShape"> {
  let description = [{
    Interface to infer the shape(s) of the data object(s)
    returned by an operation.
  }];
  let methods = [
    InterfaceMethod<
      "Infer the shape(s) of the output data object(s).",
      "std::vector<std::pair<ssize_t, ssize_t>>",
                                        "inferShape", (ins)>
];}
```

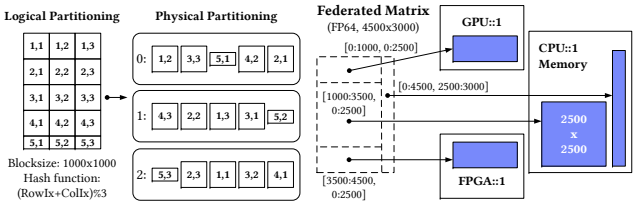
NumScalar refers to any supported integer or floating point type, and is used to parametrize the value types of operation inputs/outputs.

The InferShapeOpInterface specifies an `inferShape` method that returns a vector of matrix dimensions (pair of rows and columns, for each result). For matrix multiplication, we take the rows from the left- and columns from the right-hand-side. Additional traits exist for operator fusion (vectorization), distributed operations, and type inference. During parsing, we instantiate the individual operations, blocks, and regions to obtain a DaphneIR program, and apply optimization passes for rewrites, shape inference, and lowering of the SCF dialect. Finally, this representation is further lowered to the MLIR-LLVM dialect, including LLVM function calls to specific kernels (e.g., `ewNeq` and `sumAll` for `sum(u != c)`), and ultimately compiled to hardware-specific instructions.

Optimization Passes: The DAPHNE compilation is based on MLIR optimization passes for enrichment by inferred properties and lowering. This lowering descends from high-level, abstract operations and data types to multiple levels of operator specialization (e.g., local/distributed operations, device placement, choice of physical kernels), as well as data specialization (e.g., DenseMatrix/CSRMatrix representations). Optimization passes for rewrites and lowering can be interleaved and repeatedly executed according to known dependencies. Important categories of optimization passes include (so far, only partially implemented):

- MLIR Programming Language Rewrites (e.g., CSE, constant folding, branch removal, code motion, function inlining)
- 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 (e.g., many peephole optimizations for relational/linear algebra operations)
- Operator Ordering (e.g., join/matmult ordering/enumeration, sum-product optimizations, operator scheduling)
- Operator Fusion (selection of fused operators, vectorization/tiling, and splitting/merging strategies)
- Memory Management (update-in-place, reuse of allocations)
- Execution Type Selection (local vs distributed, distributed caching/partitioning)
- Device Placement (e.g., CPU/GPU/FPGA, multiple devices)
- Physical Operator Selection (e.g., different join/group-by, matrix multiplication, and matrix-vector operators)

Additional Compiler Components: There are additional compiler components that are used by several passes and during runtime. First, many advanced rewrites and reordering require cost estimation. We aim to provide a cost estimation component including cardinality and sparsity estimation (with different cost functions and summary statistics) for blocks of operations and entire sub-programs. Costing sub-programs needs to reason about loops or branches, and compute aggregated costs. Second, data-dependent operators, UDFs, and control flow can create unknown shapes and properties [45], which would result in inefficient fallback plans. A recompilation component [15] aims to adaptively recompile sub-programs at natural or artificial block boundaries according to the actual sizes of intermediates. Third, various runtime strategies would benefit from compiler assistance. Examples include compiler-assisted compression and reuse [69], which both leverage workload characteristics to make more informed choices during runtime.



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

Figure 2: Distributed/Multi-device Data Representations.

5.2 Data Representations

DAPHNE's basic data types are frames, matrices, and scalars. Each matrix, scalar or frame-column has a 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 that are the inputs/outputs of kernels.

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) format of row offsets, column-index and value arrays. While matrices are homogeneous arrays, frames have a schema and thus, require the handling of value types. Given common analytic workload characteristics, our frames rely on 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—similar to NumPy [40]—a view window on a potentially larger array.

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 fine-grained control if needed:

- *Distributed Collection of Tiles:* A matrix is divided into fixed-size blocks and stored as a collection of block-indexes and blocks [50]. By default, such a bag is unordered but can be partitioned (hash, range), or sorted.
- *Federated Matrix/Frame:* A federated matrix is a virtual matrix whose individual parts (identified by index ranges) are stored as a local or distributed data structure at a particular federated site [8] or device.

Figure 2 shows an example $4{,}500 \times 3{,}000$ matrix. Both distributed representations are amenable to data-parallel computation, but they have different tradeoffs regarding distribution, load balancing, sparsity, and direct access. The collection-of-tiles splits the matrix into squared blocks of fixed size, and applies hash partitioning if needed. The host does not maintain the location of individual tiles but distributed joins and aggregations apply, and tiles are always aligned. In contrast, the federated matrix keeps metadata of physical data at the host, and stores device-local data as a local data structure. This approach avoids overheads for ultra-sparse matrices, allows more effective placement, and yields efficient kernel operations.

5.3 Local and Distributed Runtime

The compiler produces an execution plan with calls to C++ host kernels for local, distributed, or accelerator operations. Our kernels

make heavy use of C++ templates for both value types and combinations of dense and sparse inputs. Since we support hundreds of operations that require specialization, we automatically generate the template instantiations. For n-ary operations with mixed types, the compiler injects casts, some of which (e.g., casting an FP32 frame-column to an FP32 matrix) are no-ops.

Definition 1. ***Kernel:*** *A kernel is an implementation of an IR operation (or registered user-defined kernel) that operates on instantiated and materialized data types. Most kernels are stateless (except memory allocation) and deterministic. Stateful kernels are allowed as well (e.g., implementing configuration management and setup/tear-down of context objects for device/cluster initialization and cleanup).*

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 devices or frameworks are local kernels themselves that add state to the global context. This approach simplifies the integration of new accelerators by registering such kernels in the extension catalog.

Distributed Runtime: We aim for an integration with different distributed programming models and resource managers. As a first step, we are building the DAPHNE standalone distributed runtime with dedicated worker processes, simple RPC communication (using gRPC), and a basic integration with SLURM as a common HPC resource manager. The host kernels of distributed operations then bring data into a distributed representation if needed and spawn distributed jobs in the form of MLIR snippets that can be compiled in an architecture-aware manner at the individual workers. In the future, we aim to further integrate MPI and device-specific collective operations (e.g., NVIDIA NCCL), and embedded deployments in different HPC, cloud, and DB environments.

5.4 Accelerators and Storage

Most HW accelerators like GPUs, FPGAs, and near-SSD compute have a cache hierarchy and high-bandwidth memory. In hybrid runtime plans that utilize heterogeneous hardware, a data object might be partitioned or replicated across devices. For flexibility—e.g., in programs with conditional control flow—we keep this data-location information in runtime data structures. Specifically, matrices and frames reference the host data (which can be a `nullptr`), and/or data on HW accelerators, computational storage devices, and distributed workers. 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 initiates implicit (stream & discard) or explicit (copy & retain) data transfer. Additionally, the compiler can inject `prefetch` and `broadcast` directives to overlay anticipated transfers with other operations. These primitives nicely generalize to different HW accelerators, the distributed runtime, and computational storage [7, 54].

Example 2. ***Near-SSD Quantization:*** *For the DLR inference workload from Section 2.1, we might broadcast the quantization boundaries (and parts of the trained model) via the mentioned primitives 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 to the CPU or other HW accelerators by 4x.*

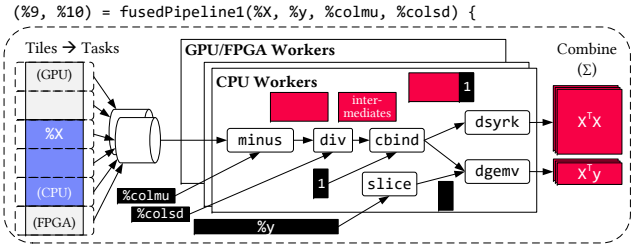


Figure 3: Vectorized Execution of Compiled Operator Pipelines (with multi-device data and task placement)**.**

5.5 Vectorized (Tiled) Execution

Basic runtime plans of kernels with materialized intermediates offer good performance and simplify debugging. Although this model is commonly used in ML systems and column stores, it suffers from several limitations. Materializing intermediates has large temporary memory and memory-bandwidth requirements, multi-threaded kernels create synchronization barriers per operator, and device placement of operators is too coarse-grained. 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.

Vectorized Task Execution: Figure 3 shows the integration of vectorized operator pipelines into execution plans. Similar to LLVM loops, a vectorized pipeline has multiple inputs, multiple outputs, and an IR body. Additionally, we specify split (e.g., row slicing) and combine (e.g., concatenate or aggregate) functions for inputs and outputs. Here, we perform matrix standardization $((\mathbf{X} - \text{colMeans}(\mathbf{X}))/\text{colSds}(\mathbf{X}))$, append a column of ones for the intercept, and compute $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ as part of a closed-form linear regression algorithm. The input matrix $\mathbf{X}$ is federated across CPU, GPU, and FPGA memory, and vectorized execution creates tasks for aligned row partitions (similar to morsels [30, 53]) and appends them to one or multiple (e.g., device-specific) task queues.

Definition 2. ***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 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 for each data item (sub-ranges). Vectorization is achieved through these input data bindings, which also affect the size of pipeline intermediates.*

Worker threads then dequeue and execute tasks, and combine the results with worker-local aggregation. HW accelerator workers are CPU threads that launch the accelerator kernels per task.

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 3 can be invoked at row granularity (where the BLAS matrix

multiplications dsyrk and dgemv can be specialized to an outer product dger and daxpy), minimizing the size of intermediates for minus, div, and cbind. 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. The operations inside a fused operator pipeline are regular kernel calls with view-based indexing of inputs, which allows reusing kernels, and exploiting sparsity of both, inputs and pipeline intermediates.

Multi-device Scheduling: As shown in Figure 3, vectorized execution also seamlessly integrates HW accelerators and scheduling. For CPU kernels, we leverage single-operator pipelines as the default multi-threading approach. In this framework, we will further explore different task partitioning and scheduling strategies, single and multiple task queues (e.g., device-specific with task stealing), data-locality-aware scheduling, and means of runtime adaptation. Finally, vectorized execution also nicely integrates with computational storage, where operator pipelines might be executed on near-SSD CPUs or FPGAs; and the task queues can connect asynchronous I/O and subsequent computation pipelines.

Code Generation: Vectorized execution also simplifies code generation. Instead of interpreting vector kernels, 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 [85] for matrices and frames; for GPUs, we compile CUDA C++ code and call CUDA libraries; and for FPGAs, we will use OneAPI DPC++, T2S [79], and hand-crafted kernels. Similarly, but largely unexplored, for computational storage, we aim to compile eBPF byte-code programs [54].

5.6 Extensibility

Besides the extensibility features at script and configuration level—which allows registering new runtime kernels—we further aim at extensibility of DaphneIR and the optimizing compiler.

DaphneIR: Our DaphneIR dialect defines types, operations, and various traits in TableGen [84] records, but also reuses existing MLIR dialects like SCF. One developer-centric direction for extensibility is the extension of the DaphneIR dialect. Common use cases are adding new operations of an existing category (e.g., a new unary element-wise 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 might add existing or new MLIR dialects and integrate them with the rest of the system by changing the DAPHNE infrastructure internally. 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 composing existing 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 them for a given IR program under certain conditions.

Sideways Entry in Multi-level Compilation: The regular invocation of daphnec (by a user or through the Python API) takes a DaphneDSL script and then compiles and executes this script. In order to aid debugging and understanding, an explain flag allows to print the DaphneIR at different states of compilation. Similar to the use of kernels at script level, which are treated as constraints, we will extend daphnec to take valid DaphneIR instead of DaphneDSL as program specification as well. 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 daphnec, which performs the remaining lowering and runs the final executable plan.

6 EXPERIMENTS

Our experiments study a DAPHNE prototype with a preliminary compilation chain, templated kernels, vectorized execution engine, CSV reader, distributed runtime, and basic GPU integration.

6.1 Experimental Setting

HW&SW Environment: We ran the experiments on a single node with two Intel Xeon Gold 6238 CPUs @ 2.2-2.5 GHz (56 physical/112 virtual cores, 7.7 TFLOP/s), 768 GB DDR4 RAM at 2.933 GHz balanced across 6 memory channels per socket, 2×480 GB SATA SSDs (system/home), and 12×2 TB SATA SSDs (data). This node has a PCIe-connected NVIDIA Tesla T4 GPU with 8.1 TFLOP/s and 16 GB memory. Finally, we use Ubuntu 20.04.1, MLIR and LLVM as of 05/2021, openBlas 0.3.15, CUDA 11.4.1, and cuDNN-8.2.2.

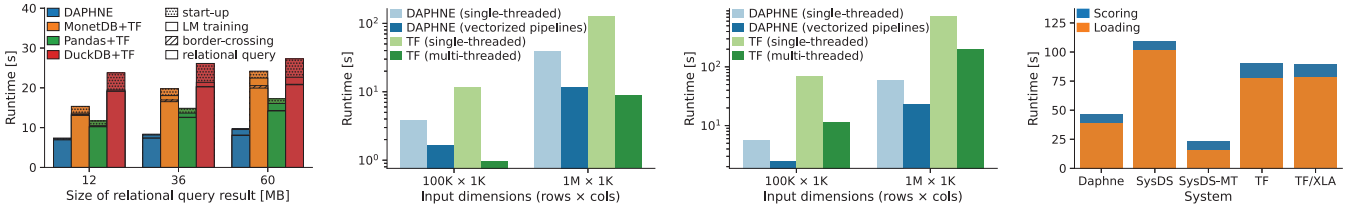
IDA Pipelines and Data: The tested workloads are four simple IDA pipelines: P1 (TPC-H query processing, standardization, and linear regression), P2 (earth observation ResNet20 scoring), K-means clustering (with 20 iterations), and connected components from Example 1. First, for pipeline P1, we use the TPC-H data generator with scale factor SF=10, processing the query

$$\gamma_{\text{CID,C.*,sum(TPrice)}}(\sigma_{\text{MktSeg}=p}(C) \bowtie_{\text{CID}} \sigma_{Date \in [3a]}(O)) \quad (1)$$

(filters on Customers/Orders, and a group-join [58]), following by one-hot encoding, normalization and linear regression as discussed in Section 5.5. Baselines are MonetDB, Pandas, and DuckDB [71], each combined with TensorFlow (TF) 2.6 [1]. Second, pipeline P2 is the ResNet20 scoring pipeline from the earth observation use case—described in Section 2.1—applied to the 4.9 GB test data in FP32, where we compare with TF, TF XLA [52], and SystemDS [17].

6.2 Simple IDA Pipelines

P1 – Query Processing and LM: For a fair comparison, DAPHNE and all baselines read from CSV, with different loading (e.g., MonetDB load into temporary tables) and transfer strategies, and all systems run single-threaded. Figure 4(a) shows the results with varying query output size (selectivity of predicate p). Due to the group-by aggregation, boundary crossing and model training has only minor performance impact. However, the larger the passed intermediates, the more DAPHNE benefits from integrated execution. Figures 4(b) and 4(c) show additional micro-benchmarks for normalization and LM regression (closed-form algorithm, part of P1) as well as K-means clustering on synthetic data. Vectorized execution already yields speedups of 3x with room for improvements.



(a) P1: Query and LM (b) LM Micro-Benchmark (c) K-means Micro-Benchmark (d) P2: ResNet20 Scoring

Figure 4: Preliminary Experimental Results of Selected IDA Pipelines and Micro-Benchmarks.

P2 – ResNet20 Scoring: Figure 4(d) shows the P2 results on the T4 GPU. We trained the ResNet20 model with the Adam optimizer for 100 epochs on `So2Sat LCZ42` [90]. TF and TF XLA (both with GPU) show similar performance, which is dominated by I/O, while SystemDS' [17] (with GPU and read into FP64 and conversion to FP32) is about 1.2x slower, but 4x faster with multi-threaded I/O. DAPHNE's basic CSV reader and GPU integration already show the best single-threaded performance. Vectorized read pipelines and task placement have potential for solid improvements.

Additional Experiments: We conducted additional experiments with FPGA kernels (Intel FPGA PAC D5005 with Stratix 10SX FPGA and 32 GB memory), hybrid CPU+GPU vectorized pipelines, vectorized sparse pipelines, and distributed operations with up to 100 workers on the Vega[2] Supercomputer. While still in an early stage and several integration challenges remain, we observed promising results and can draw several conclusions. First, vectorized execution with reuse of kernels makes multi-device scheduling feasible, and simplifies multi-threaded pipelines with sparse intermediates. Second, hybrid local+distributed or multi-device execution requires a careful planning of data/task placement and related data transfers. Third, vectorized execution in distributed operations is a promising direction for exploiting both inter- and intra-node parallelism.

7 RELATED WORK

System infrastructure for IDA pipelines is related to a wide variety of areas. We specifically discuss the context of modern system support for IDA pipelines, trends of HW accelerator integration, and vectorized (tiled) execution, and extensibility.

Systems for IDA Pipelines: The trend toward IDA pipelines is currently handled with a combination of existing systems including standalone and embedded DBMS like DuckDB [71], ML systems like TensorFlow [1] or PyTorch [65], data-parallel computation frameworks like Spark [87], Flink [20], or Dask [73] (often with collections of tiles of an overall matrix or frame), and variety of specialized systems or libraries (e.g., for graph processing and time series analysis). Furthermore, ML systems are extended with basic data processing (e.g., TFX [9]), DBMS are extended with ML capabilities (e.g., via UDFs or lambda functions) [50], data-parallel frameworks aim to provide a unified environment [87], compilation frameworks like MLIR [51] or CVM [60] provide common compiler infrastructure, and HPC techniques are increasingly adopted across these systems [13]. However, these integrated systems often rely on separate libraries and data representations for query processing, ML, and HPC; the integration of HW accelerators is not holistic; and the handling of numerical HPC codes is limited.

[2]Vega Supercomputer (Maribor): https://doc.vega.izum.si/general-spec/

HW Accelerator Integration: The spectrum of hardware acceleration ranges from CPUs with SIMD, over GPUs and FPGAs, to custom ASICs and focuses on tradeoffs regarding reconfiguration capabilities, performance, and energy efficiency [64]. Other dimensions include custom data types, sparsity exploitation (e.g., via operator fusion [16], or HW support [63]), and near-data processing (e.g., on SSDs with FPGAs attached [6, 7]). Existing work largely relies on manual or heuristic operator placement, but there is work on reinforcement learning for multi-device operator placement [57], and new link technologies significantly influence these decisions [55, 72]. Recent work applies self-scheduling across devices [30], and data partitioning according to expected device performance [37], for fully utilizing available devices. At systems level, mostly HW-vendor-provided libraries (e.g., BLAS, DNN, but also frame operations [62]) are used for CPU, GPU, and partially FPGA operations, while FPGAs and ASICs are often integrated via compilation frameworks like TensorFlow XLA [52], TVM [22], T2S [79], EVEREST [70], or target-specific compilers [64]. DAPHNE as a compiler and runtime system aims to improve the productivity, extensibility, and performance of utilizing multiple heterogeneous devices.

Vectorized Execution: Vectorized execution is a heavily overloaded term including (1) computation via coarse-grained (vectorized) array operations, (2) SIMD (vector) instruction parallelism, and (3) vector-at-a-time (vectorized) query processing à la MonetDB/X100 [18]. Interestingly, all three interpretations apply to DAPHNE: the system is optimized for data analysis and linear algebra on frames and matrices, the kernels and LLVM compiler exploit SIMD and SPMD parallelism, and the central vectorized execution engine processes batches of data. Besides ML systems, recent work on vectorized array operations include tensors for data processing [48], decision tree predictions in Hummingbird [61], slicing finding in SliceLine [75], and maximum inner-product search in Maximus [2], which all cleanly map complex algorithms to vectorized array operations. Furthermore, vectorization is related to fused and compiled operator pipelines in SystemDS [16] and Tuplex [78] as well as work on morsel-driven query processing [30, 53].

Extensibility: Providing extensibility for functionality and performance has been investigated in different systems and at different abstraction levels. First, in the context of database management systems, there are great surveys of work on extensibility [21]. Abstract data types and user-defined functions/aggregates were introduced in PostgreSQL [81] and are now widely used in practice. Such UDFs have also been used to integrate ML into DBMS [33] and HPC OpenMP applications in DBMS [86], but UDFs are often treated as black boxes and thus, not subject to optimization (unless specifically handled [43]). Additional means of extensibility include query optimizer generators [38], extensible cardinality estimation [47],

interfaces for new storage methods (a.k.a. storage managers, or engines) with well-defined interfaces for create/drop relation, insert, delete, update, and get/getNext operations, but also persistently stored modules like attachments or triggers. Second, several ML systems also provide means of extensibility. DNN frameworks like Caffee, PyTorch, and TensorFlow make it easy to add layers and optimizers. AutoML systems like MLBase defined catalogs for registering new ML algorithms with their cost functions [49]. Additional work also focuses on extensibility of system internals. Examples include TensorFlow distribution strategies for mini-batch training [36], TVM code generation for new hardware backends [22], and the Flashlight library for extensibility by custom modules and kernels [32]. DAPHNE is inspired by these existing works and aims to build an open and extensible infrastructure for IDA pipelines.

8 CONCLUSIONS

We described the 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 an MLIR-based compilation chain, frame and matrix representations, HW accelerators and computational storage, hierarchical 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 and GPUs show promising results. In the next few years, we will build out this infrastructure and tackle research challenges across the different levels from resource management, device kernels, I/O, buffer management, and vectorized execution, over compilation, operator and pipeline scheduling, to seamless extensibility and customization of IDA pipelines, as well as extensibility of system internals.

ACKNOWLEDGEMENTS

The DAPHNE project is funded by the European Union's Horizon 2020 research and innovation program under grant agreement number 957407 from 12/2020 through 11/2024. We further thank Aristotelis Vontzalidis (ICCS/NTUA), Dževad Ćoralić (TU Graz), and Thomas Krametter (KAI) for their valuable contributions—after the initial paper submission—to the DAPHNE distributed runtime, Python API, and use case implementation, respectively.

REFERENCES

[1] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.
[2] F. Abuzaid, G. Sethi, P. Bailis, and M. Zaharia. To Index or Not to Index: Optimizing Exact Maximum Inner Product Search. In *ICDE*, 2019.
[3] S. Agrawal, L. Barrington, C. Bromberg, J. Burge, C. Gazen, and J. Hickey. Machine Learning for Precipitation Now- casting from Radar Images. *CoRR*, abs/1912.12132, 2019.
[4] S. N. M. Albarqouni. *Machine Learning for Biomedical Applications*. PhD thesis, TU Munich, 2017.
[5] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3), 2015.
[6] Amazon. AQUA (Advanced Query Accelerator) for Amazon Redshift, 2021.
[7] A. Barbalace and J. Do. Computational Storage: Where Are We Today? In *CIDR*, 2021.
[8] S. Baunsgaard et al. ExDRa: Exploratory Data Science on Federated Raw Data. In *SIGMOD*, 2021.
[9] D. Baylor et al. TFX: A TensorFlow-Based Production- Scale Machine Learning Platform. In *SIGKDD*, 2017.
[10] J. Bezanson et al. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.
[11] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.
[12] S. Bhattacherjee, A. Deshpande, and A. Sussman. PStore: an efficient storage framework for managing scientific data. In *SSDBM*, 2014.
[13] S. Blanas, P. Koutris, and A. Sidiropoulos. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. In *CIDR*, 2020.
[14] M. Boehm et al. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7), 2014.
[15] M. Boehm et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.
[16] M. Boehm et al. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB*, 11(12), 2018.
[17] M. Boehm et al. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*, 2020.
[18] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.
[19] J. Cambronero, J. K. Feser, M. J. Smith, and S. Madden. Query optimization for dynamic imputation. *PVLDB*, 10(11), 2017.
[20] P. Carbone et al. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE DE Bull.*, 38(4), 2015.
[21] M. J. Carey and L. M. Haas. Extensible database management systems. *SIGMOD Rec.*, 19(4), 1990.
[22] T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018.
[23] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation strategies from data. In *CVPR*, 2019.
[24] B. Dageville et al. The Snowflake Elastic Data Warehouse. In *SIGMOD*, 2016.
[25] T. Dao, A. Gu, A. Ratner, V. Smith, C. D. Sa, and C. Ré. A kernel theory of modern data augmentation. In *ICML*, 2019.
[26] S. Das et al. Ricardo: integrating R and Hadoop. In *SIGMOD*, 2010.
[27] J. Dean et al. Large scale distributed deep networks. In *NeurIPS*, 2012.
[28] M. Deregnaucourt, M. Stadlbauer, C. Hametner, S. Jakubek, and H.-M. Koegeler. Evolving model architecture for custom output range exploration. *Mathematical and Computer Modelling of Dynamical Systems*, 21(1), 2015.
[29] L. Dong and T. Rekatsinas. Data integration and machine learning: A natural synergy. *PVLDB*, 11(12), 2018.
[30] K. Dursun, C. Binnig, U. Çetintemel, G. Swart, and W. Gong. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *PVLDB*, 12(12), 2019.
[31] D. Duvenaud, J. R. Lloyd, R. B. Grosse, J. B. Tenenbaum, and Z. Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *ICML*, 2013.
[32] Facebook. Flashlight: Fast and flexible machine learning in C++, 2021. https://github.com/flashlight/flashlight.
[33] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.
[34] S. Gan et al. BAGUA: Scaling up Distributed Learning with System Relaxations. *CoRR*, abs/2107.01499, 2021.
[35] G. E. Gévay, J. Quiané-Ruiz, and V. Markl. The power of nested parallelism in big data processing - hitting three flies with one slap -. In *SIGMOD*, 2021.
[36] Google. Inside TensorFlow: tf.distribute.Strategy, 2019. https://www.youtube.com/watch?v=jKV53r9-H14.

[37] M. Gowanlock, B. Karsin, Z. Fink, and J. Wright. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *DaMoN@SIGMOD*, 2019.

[38] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[39] P. Große, W. Lehner, T. Weichert, F. Färber, and W. Li. Bridging two worlds with RICE integrating R into the SAP in-memory computing engine. *PVLDB*, 4(12), 2011.

[40] C. R. Harris et al. Array programming with numpy. *Nat.*, 585, 2020.

[41] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[42] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, 2016.

[43] F. Hueske et al. Opening the black boxes in data flow optimization. *PVLDB*, 5(11), 2012.

[44] I. F. Ilyas, J. Rao, G. M. Lohman, D. Gao, and E. T. Lin. Estimating compilation time of a query optimizer. In *SIGMOD*, 2003.

[45] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, 2004.

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

[47] V. Josifovski, P. M. Schwarz, L. M. Haas, and E. T. Lin. Garlic: a new flavor of federated query processing for DB2. In *SIGMOD*, 2002.

[48] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi. Tensors: An abstraction for general data processing. *PVLDB*, 14(10), 2021.

[49] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

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

[51] C. Lattner et al. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR*, abs/2002.11054, 2020.

[52] C. Leary and T. Wang. TensorFlow, Compiled! (TensorFlow Dev Summit), 2017.

[53] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

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

[55] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *SIGMOD*, 2020.

[56] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[57] A. Mirhoseini et al. Device Placement Optimization with Reinforcement Learning. In *ICML*, 2017.

[58] G. Moerkotte and T. Neumann. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB*, 4(11), 2011.

[59] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language - objects and functions for data analysis. In *ECOOP*, volume 7313, 2012.

[60] I. Müller, R. Marroquín, D. Koutsoukos, M. Wawrzoniak, S. Akhadov, and G. Alonso. The collection virtual machine: an abstraction for multi-frontend multi-backend data analysis. In *DaMoN@SIGMOD*, 2020.

[61] S. Nakandala et al. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *OSDI*, 2020.

[62] O. O. Napoli, V. M. do Rosario, J. P. Navarro, P. M. C. e Silva, and E. Borin. Accelerating multi-attribute unsupervised seismic facies analysis with RAPIDS. *CoRR*, abs/2007.15152, 2020.

[63] NVIDIA. A100 Tensor Core GPU Architecture, 2020.

[64] K. Olukotun. "Let the Data Flow!". In *CIDR*, 2021.

[65] A. Paszke et al. PyTorch: An Imperative Style, High- Performance Deep Learning Library. In *NeurIPS*, 2019.

[66] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning Mesh-Based Simulation with Graph Networks. *ICLR*, 2021.

[67] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. In *ICLR*, 2021.

[68] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

[69] A. Phani, B. Rath, and M. Boehm. LIMA: fine-grained lineage tracing and reuse in machine learning systems. In *SIGMOD*, 2021.

[70] C. Pilato et al. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In *DATE*, 2021.

[71] M. Raasveldt and H. Mühleisen. Data Management for Data Science - Towards Embedded Analytics. In *CIDR*, 2020.

[72] A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. G. Anadiotis, and A. Ailamaki. Gpu-accelerated data management under the test of time. In *CIDR*, 2020.

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

[74] V. Rosenfeld, R. Müller, P. Tözün, and F. Özcan. Processing java udfs in a C++ environment. In *SoCC*, 2017.

[75] S. Sagadeeva and M. Boehm. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In *SIGMOD*, 2021.

[76] S. Sakr et al. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9), 2021.

[77] G. Sharma and J. Martin. Matlab®: A language for parallel computing. *Int. J. Parallel Program.*, 37(1), 2009.

[78] L. F. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD*, 2021.

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

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

[81] M. Stonebraker. The land sharks are on the squawk box. *Commun. ACM*, 59(2), 2016.

[82] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *SSDBM*, 2011.

[83] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. R. Slutz, and R. J. Brunner. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In *SIGMOD*, 2000.

[84] L. Team. TableGen Overview, 2021. https://llvm.org/docs/TableGen/.

[85] A. Ungethüm et al. Hardware-oblivious SIMD parallelism for in-memory column-stores. In *CIDR*, 2020.

[86] F. Wolf, I. Psaroudakis, N. May, A. Ailamaki, and K. Sattler. Extending database task schedulers for multi-threaded application code. In *SSDBM*, 2015.

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

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

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

[90] X. X. Zhu et al. So2Sat LCZ42: A Benchmark Data Set for the Classification of Global Local Climate Zones [Software and Data Sets]. *IEEE GRS Magazine*, 8(3), 2020.

[91] X. X. Zhu, C. Qiu, J. Hu, Y. Shi, Y. Wang, M. Schmitt, and H. Taubenböck. The urban morphology on our planet – global perspectives from space. *Remote Sensing of Environment*, 269:112794, 2022.

[92] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.