D2.1 Initial System Architecture

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

Version 1.1

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 initial system architecture summarizes the preliminary 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 as well as fine-grained fusion, parallelism and scheduling, and a discussion and separation from related work. Essential parts of this design have been concurrently implemented in a private prototype of the DAPHNE system, which we intend to release as an open source project early 2022.

As the project continues, the requirements analysis, system architecture, and prototype system will incrementally evolve as needed. Updates of this initial system architecture will be accordingly reflected in the refined system architecture via D2.2 (M21).

D2.1 Initial System Architecture			
WP2 – System Architecture			
Type of document	R	Version	1.1
Dissemination level	PU		
Lead partner	KNOW		
Author(s)	Matthias Boehm (KNOW)		
Reviewer(s)	Pınar Tözün (ITU), Vasileios Karakostas (ICCS)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial write-up, summarizing broader design discussions and an early prototype of the system architecture	Matthias Boehm
V1.1	Incorporated comments by Pınar and Vasileios; additional details, figures and explanations	Matthias Boehm

1 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, and execution plans 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 report, we make a case for IDA pipelines, describe the overall 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.

2 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 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 [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 [L+20] 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 3.
- **System Architecture:** Subsequently, we describe the overall MLIR-based architecture, data representations, and major design decisions in Section 4.
- **Vectorized Execution:** We further introduce a vectorized execution engine for compiled operator pipelines of frames, matrices, and tensors; heterogeneous HW devices, and computational storage in Section 5.

3 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.

3.1 Example Use Cases

We describe a selected example use case from the area of earth observation, but it is representative for a much broader range of applications. The report D8.1 [SB+21] (which is concurrently developed at the time of this writing) describes the DAPHNE use cases – from earth observation, semiconductor manufacturing, material degradation, and vehicle development – as well as their initial IDA pipelines in much 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 a 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)

3.2 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.

4 DAPHNE Architecture

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

4.1 Overall System Architecture

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 [L+20] 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.

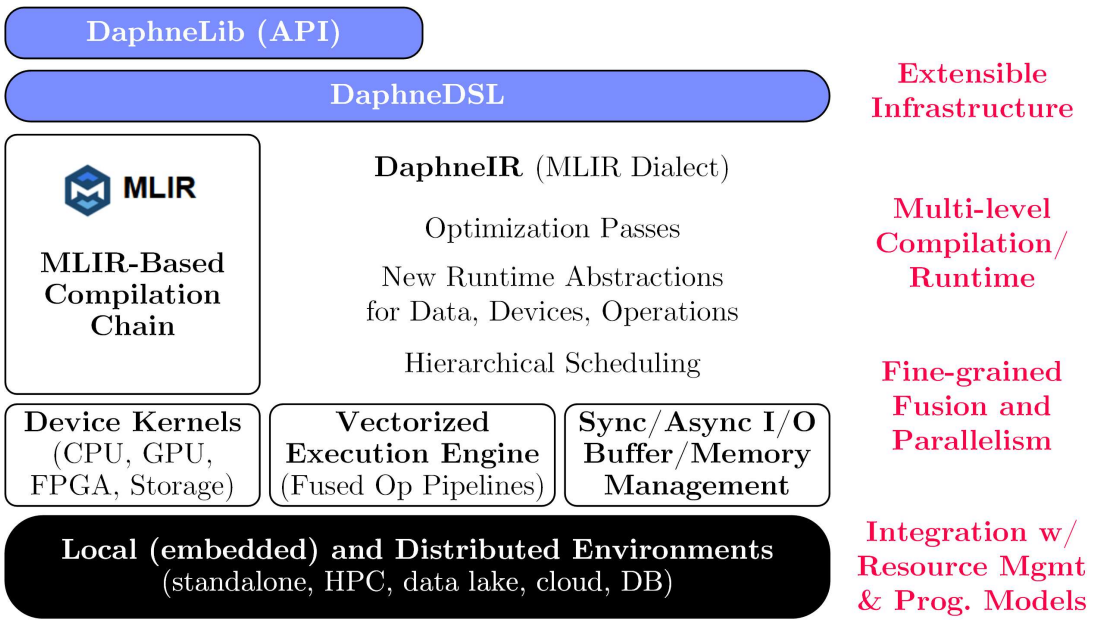


Figure 1: DAPHNE System Infrastructure.

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 [G+21]. 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 more declarative interfaces, this multi-level abstraction can simplify experimentation and extensions while providing data independence and automatic optimization for unconstrained scripts. Related research directions include appropriate abstractions – for essential extension points – which limit potential increases in system complexity, and ensure low overhead and high performance. An example is an operator abstraction for new kernels that exposes interesting properties, which in turn, can be used by more broadly applicable simplification rewrites.

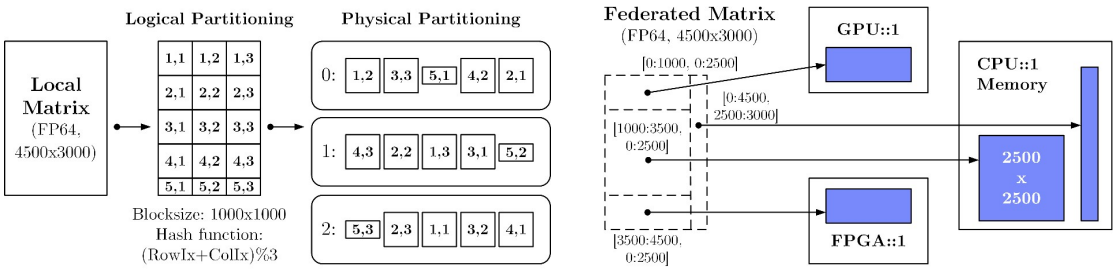
Compilation Chain: The DaphneDSL scripts are converted by an ANTLR parser into MLIR, specifically, DaphneIR as an MLIR dialect comprising conditional control flow, matrix and frame data types, as well as logical and physical frame and matrix operations. Additionally, we integrate a(n) SQL query parser that translates SQL into frame operations, and parsers for existing DSLs (like SystemDS' [BA+20] DML) in order to reuse DSL-based primitives for data cleaning, data preprocessing, ML algorithms, and model debugging. 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, and cost-based optimization. The joint system infrastructure also enables cardinality [BH+07, MNS09] and sparsity [SB+19] estimation in a holistic manner. 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.

Runtime Environment: 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 5 – further provides means of operator fusion, and a seamless integration of heterogeneous computing devices, computational storage, and distributed operations.

4.2 Data Representations

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, UINT8). 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, frames have a schema and thus, require the handling of 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 2: 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 2.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 2.b shows again an example 4500-x-3000 matrix that is federated across host and device memory, where the federated meta data 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.

4.3 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.

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.

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, 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. Accordingly, we keep the initial design generic enough to allow such extensions.

4.4 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 maximum flexibility – for example 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 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 3.1, 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. More detailed designs of managed storage tiers, and near-data processing, as well as the integration and full utilization of HW accelerators will follow in dedicated deliverables of WP6 and WP7.

5 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.

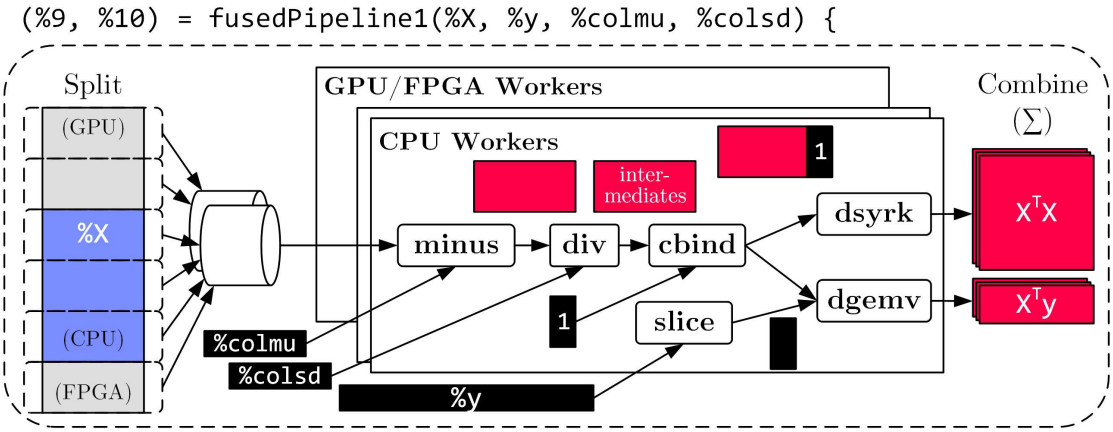


Figure 3: Vectorized Execution of Compiled Operator Pipelines
(with multi-device data and task placement).

Vectorized Task Execution: Figure 3 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.

Definition 1 - 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 for each data item.

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.

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 2 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 3, 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 will further explore 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 GPUs, we compile CUDA C++ code and call CUDA libraries; and for FPGAs, we use OneAPI DPC++, T2S [S+19], and hand-crafted kernels. Similarly, but largely unexplored, for computational storage, we aim to compile eBPF byte-code programs [LB21].

6 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 execution models.

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 [RM20], ML systems like TensorFlow [A+16] or PyTorch [P+19], data-parallel computation frameworks like Spark [Z+12], Flink [BZN05], or Dask [R15] (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 features for basic data processing (e.g., from TensorFlow to TFX [B+17]), DBMS are extended with ML capabilities (e.g., via UDFs or lambda functions) [KBY17], data-parallel frameworks aim to provide a unified environment [Z+12], compilation frameworks like MLIR [L+20] or CVM [M+20] provide common compiler infrastructure and HPC techniques are increasingly adopted

across these systems [BKS20]. 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 handling of numerical HPC codes is poor.

HW Accelerator Integration: The spectrum of hardware acceleration ranges from CPUs with SIMD, over GPUs and FPGA, to custom ASICs and focuses on different tradeoffs of reconfiguration capabilities and levels of performance or energy efficiency. Additional dimensions include custom data types, sparsity exploitation (e.g., via operator fusion [B+18], or HW support [N20]), and near-data processing (e.g., on smart SSDs [BD21], as used in (e.g., Amazon AQUA [AA21]). Existing work largely relies on manual or heuristic operator placement, but there is also work on using reinforcement learning for operator placement onto multiple heterogeneous devices [M+17], and new link technologies significantly influence these decisions [L+20b, R+20]. Recent work further applies self-scheduling schemes across devices [D+19], or partitions the data accordingly to expected device performance [G+19], for utilizing all available devices jointly. At systems level, mostly HW-vendor-provided libraries (e.g., BLAS, DNN, but also frame operations [N+20b]) are used for CPU, GPU, and partially FPGA operations, while FPGAs and ASICs are often integrated via dedicated compilation frameworks like TensorFlow XLA [LW17], TVM [C+18], EVEREST [P+21b], or target-specific compilers [O21]. 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 a la MonetDB/X100 [BZN05]. 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 a central component is our vectorized execution engine processing batches of data. Besides ML systems, recent work on vectorized array operations include tensors for data processing [K+21], decision tree predictions in Hummingbird [N+20], slicing finding in SliceLine [SB21], and maximum inner-product search in Maximus [A+19], which all cleanly map complex algorithms to vectorized array operations. Furthermore, vectorization is related to fused and compiled operator pipelines in SystemDS [B+18] and Tuplex [SYS21] as well as work on morsel-driven query processing [L+14, D+19].

7 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, 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 build out this infrastructure and tackle 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.

8 References

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

[A+19] F. Abuzaid et al. To Index or Not to Index: Optimizing Exact Maximum Inner Product Search. In ICDE, 2019.

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

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

[AA21] Amazon. AQUA (Advanced Query Accelerator) for Amazon Redshift, 2021.

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

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

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

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

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

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

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

[B+18] M. Boehm et al. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. PVLDB, 11(12), 2018.

[BZN05] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In CIDR 2005.

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

[C+18] T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In OSDI, 2018.

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

[D+19] K. Dursun et al. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. PVLDB, 12(12), 2019.

[G+21] S. Gan et al. BAGUA: Scaling up Distributed Learning with System Relaxations. CoRR, abs/2107.01499, 2021.

[G+19] M. Gowanlock et al. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In DaMoN@SIGMOD, 2019.

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

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

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

[K+21] D. Koutsoukos et al. Tensors: An abstraction for general data processing. PVLDB, 14(10), 2021.

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

[L+20] C. Lattner et al. MLIR: A Compiler Infrastructure for the End of Moore's Law. CoRR, abs/2002.11054, 2020.

[LW17] C. Leary and T. Wang. TensorFlow, Compiled! (TensorFlow Dev Summit), 2017. https://www.youtube.com/watch?v=kAOanJczHA0&feature=emb_logo.

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

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

[L+20b] C. Lutz et al. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In SIGMOD 2020.

[M+17] A. Mirhoseini et al. Device Placement Optimization with Reinforcement Learning. In ICML, 2017.

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

[MB11] G. Moerkotte and T. Neumann. Accelerating Queries with Group-By and Join by Groupjoin. PVLDB, 4(11), 2011.

[M+20] I. Mueller et al. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. In DaMoN@SIGMOD, 2020.

[N+20] S. Nakandala et al. A Tensor Compiler for Unified Machine Learning Prediction Serving. In OSDI, 2020.

[N+20b] O. O. Napoli et al. Accelerating Multi-attribute Unsupervised Seismic Facies Analysis With RAPIDS. CoRR, abs/2007.15152, 2020.

[N20] NVIDIA. A100 Tensor Core GPU Architecture, 2020. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[O21] K. Olukotun. Let the Data Flow!. In CIDR, 2021.

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

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

[P+21b] C. Pilato et al. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In DATE, 2021.

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

[R+20] A. Raza et al. GPU-accelerated data management under the test of time. In CIDR, 2020.

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

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

[SB21] S. Sagadeeva and M. Boehm. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In SIGMOD, 2021.

[SB+19] J. Sommer: MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. SIGMOD 2019.

[SYS21] L. F. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska. Tuplex: Data Science in Python at Native Code Speed. In SIGMOD, 2021.

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

[SB+21] B. Steinwender, M. Birkenbach, M. Hofer, D. Krems, A. Laber, N. Skuppin: DAPHNE D8.1 Initial Pipeline Definition of all Use Cases, 2021.

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

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

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

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

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