D3.5
Final Compiler Prototype

DAPHNE

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

Version 1.3

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 document presents the final DAPHNE compiler prototype. As described in the compiler design and overview [D3.4], the DAPHNE compiler is based on MLIR [LA+21], a framework for domain-specific compilers, to facilitate a cost-effective development of our domain-specific language, reuse of compiler infrastructure, and good extensibility. Along with this document, we share a snapshot of the final compiler prototype's source code, which is also a part of the open-source DAPHNE prototype repository on GitHub. After comments on the artifact access (Section 1) and environment setup (Section 2), we showcase the final compiler prototype by means of three demonstration scenarios. The first scenario (Section 3) walks through some of the most decisive steps of DAPHNE's optimizing compilation chain. It is mostly based on the demonstration that accompanied the extended compiler prototype [D3.3], but significantly expanded in the parts we worked on in the past 1.5 years, i.e., extensibility and MLIR-based code generation for CPU. The second scenario (Section 4) focuses in more detail on extensibility and code generation. The third scenario (Section 5) presents our work on sparsity-exploiting operator fusion. For each scenario, we present experimental results that emphasize the capabilities of the DAPHNE compiler. After a brief overview of other finished and ongoing work in the DAPHNE compiler (Section 6), we conclude the document with an overview of the final compiler prototype's source code (Section 7).

D3.5 Final Compiler Prototype			
WP3 – DSL Abstractions and Compilation			
Type of document	D	Version	1.3
Dissemination level	PU	M48 (Nov 2024)	
Lead partner	TUB		
Author(s)	Patrick Damme (TUB), Philipp Ortner (TUB), Matthias Boehm (TUB), DAPHNE Development Team		
Reviewer(s)	Florina M. Ciorba (UNIBAS), Dimitrios Tsoumakos (ICCS), Jonas H. Müller Korndörfer (UNIBAS), Quentin Guilloteau (UNIBAS)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Populated document as a copy of D3.3.	Patrick Damme (TUB)
V1.1	Revised text from the perspective of M48, restructured the document, expanded demo scenario, added second demo scenario including experiments.	Patrick Damme (TUB)
V1.2	Added third demo scenario.	Patrick Damme (TUB)
V1.3	Addressed feedback by internal reviewers, re-ran experiments with final code, various little improvements	Patrick Damme (TUB)

Table of Contents

List of Figures

List of Abbreviations

Abbreviation	Meaning
AVX	Advanced Vector eXtensions
CG	Conjugate Gradient
CPU	Central Processing Unit
CSV	Character-Separated Values
DS	Direct Solve
GPU	Graphics Processing Unit
IR	Intermediate Representation
JIT	Just-In-Time
JSON	JavaScript Object Notation
LLVM	Low-Level Virtual Machine
MLIR	Multi-Level Intermediate Representation
PNMF	Poisson Nonnegative Matrix Factorization
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
SVE	Scalable Vector Extensions

1 Artifact Access

The final compiler prototype is publicly accessible as a snapshot of the DAPHNE prototype under the **link** https://daphne-eu.know-center.at/index.php/s/t9Q24w8f85Tfia7 (~3.1 MB).

This snapshot is a copy of the DAPHNE open-source repository at https://github.com/daphne-eu/daphne (commit 741faf4f20b0fbc2515baede3da186146601c644; Nov 26, 2024).

Furthermore, the artifact contains the directories `scenario1`, `scenario2`, and `scenario3` with files specific to the three demonstration scenarios in this document, such as the full DaphneIR outputs shown in shortened form in this document, the scripts for the experiments, and the raw experimental result.

2 Environment Setup

We recommend using a GNU/Linux system (e.g., Ubuntu 24.04) for following the demonstration scenarios. After downloading the file `daphne-d3.5-v1.1.zip` from the link above, the following commands need to be executed in a terminal from the same directory to set up the environment for going along with the demonstration scenarios in the next sections. The commands can be copied separately into the terminal. Commands that span multiple lines use \ (backslash) at the line endings. Alternatively, the DAPHNE documentation in `doc/GettingStarted.md`[1] contains instructions for setting up DAPHNE (but for ease of use, it is recommended to follow the instructions below). As a fallback, this deliverable contains all outputs and scripts shown in this document in the directories `scenario1`, `scenario2`, and `scenario3`, such that it can also be understood without running the system.

```
# Unpack the deliverable artifact and cd into it.
unzip daphne-d3.5-v1.1.zip
cd daphne-d3.5-v1.1

# Pull a container image with pre-built dependencies.
docker pull daphneeu/daphne-dev

# Run bash in the container for an interactive environment.
./containers/run-docker-example.sh

# Build DAPHNE in the container (should take only a few minutes).
./build.sh --no-deps
```

[1] The documentation can also be found only at https://daphne-eu.github.io/daphne/GettingStarted/.

3 Scenario 1: DAPHNE Compilation Chain / Linear Regression

3.1 Running Example: Linear Regression Model Training

In the first demonstration scenario, we showcase some of the most important (but not all) steps of the DAPHNE compilation chain. To this end, we make use of DaphneDSL scripts that train a linear regression model on real data from a CSV file on secondary storage. We have already used this example for demonstrating the initial and extended compiler prototypes [D3.2, D3.3]. We decided to include this demonstration scenario again to provide a comprehensive and self-contained view of the final compiler prototype. Consequently, this section is mostly an updated copy of the demonstration scenario of the extended compiler prototype [D3.3], where we have significantly expanded on the topics of lowering DaphneIR operations to kernel calls as well as on the newly introduced MLIR-based code generation backend. DAPHNE still supports two methods for the task of linear regression model training: The direct solve (DS) method and the conjugate gradient (CG) method (the initial compiler prototype [D3.2] showed only the DS method). The DS method solves the task by a closed form computation (typically most efficient on a small number of features), while the CG method is an iterative numerical algorithm (typically most efficient on a high number of features). The respective scripts have been translated to DaphneDSL from SystemDS [BA+20] (https://github.com/apache/systemds) and can be found in Appendix 1 as well as in the files `scripts/algorithms/lmDS_.daph` and `scripts/algorithms/lmCG_.daph`. Note that each of these scripts defines a single function, which can be imported into any other DaphneDSL script, thereby offering reusable high-level building blocks for integrated data analysis pipelines. In the following, we focus on the DS method, which we invoke via the DaphneDSL script `scripts/algorithms/lmDS.daph` that imports this function and calls it with the loaded input data. We will come back to the CG method in the micro benchmarks in Section 3.4.

Before we can run these DaphneDSL scripts, we need to download the small real-world data set we want to train the linear regression model on. The data set can be prepared by the following commands (or by executing `scenario1/data.sh` from the root directory of the artifact).

```
# Download a small real data set and slightly pre-process it, such that DAPHNE can use it.
mkdir data
wget https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv \
  -O data/wine.csv
# These sed commands were tested on Ubuntu GNU/Linux. They may not work as expected on
# other platforms. As a fallback, one can apply these changes manually in a text editor.
sed -i '1d' data/wine.csv        # remove the first line (header)
sed -i 's/;/,/g' data/wine.csv   # replace ; by , (column delimiter)
echo '{"numRows": 4898, "numCols": 12, "valueType": "f64", "numNonZeros": 58776}' \
  > data/wine.csv.meta
```

3.2 Executing Linear Regression Model Training

To execute the DS method on real input data, a user can call DAPHNE as follows (assuming the present working directory is the root of the daphne/ source tree):

```
bin/daphne \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=true
```

This command invokes the DAPHNE command-line interface through the executable `bin/daphne` and tells it to execute the script `script/algorithms/lmDS.daph`. The complete script can also be found in Appendix 1. The remaining arguments are passed to the DaphneDSL parser as script arguments: `XY` is the path to the input file that contains the feature matrix `X` and the labels vector `y`, `reg` is a regularization constant for L2-regularization and should be set to non-zero for highly dependent, sparse, or numerous features (we set it to a default value here); `icpt` stands for intercept and indicates whether a shifting and scaling of the input features in `X` should be performed; finally, the boolean flag `verbose` indicates whether to print detailed output during the script execution. As we set `verbose` to `true`, the console output shows a few informative statistics on the calculated model as well as the model itself:

```
Calling the Direct Solver...
Computing the statistics...
AVG_TOT_Y, 5.877909
STDEV_TOT_Y, 0.885639
AVG_RES_Y, 0.000000
STDEV_RES_Y, 0.751357
DISPERSION, 0.564537
R2, 0.281870
ADJUSTED_R2, 0.280254
R2_NOBIAS, 0.281870
ADJUSTED_R2_NOBIAS, 0.280254

RESULT
DenseMatrix(12x1, double)
0.0655125
-1.86318
0.0220869
0.0814792
-0.247322
0.00373283
-0.000285785
-150.274
0.68631
0.631463
0.193487
150.183
```

3.3 Steps of DAPHNE's Optimizing Compiler

Overview. To illustrate what the DAPHNE compiler does to make this script work, we have a look at selected steps of the compilation chain [D3.4]. To this end, we utilize DAPHNE's explanation feature, which prints the DaphneIR at chosen stages. Figure 1 displays an overview of the DAPHNE compiler in the context of the DAPHNE system architecture and highlights the steps of the compilation chain that we will have a closer look at in this section. In particular, these are: (0) the initial DaphneIR after DaphneDSL parsing, (1) initial simplifications (e.g., general programming language rewrites), (2) type and property inference, (3) physical operator selection, (4) the generation of fused operator pipelines for vectorized execution, (5) memory management, (6) lowering DaphneIR operations (a) to pre-compiled kernels and (b) by code generation, and (7) lowering to LLVM and JIT-compilation.

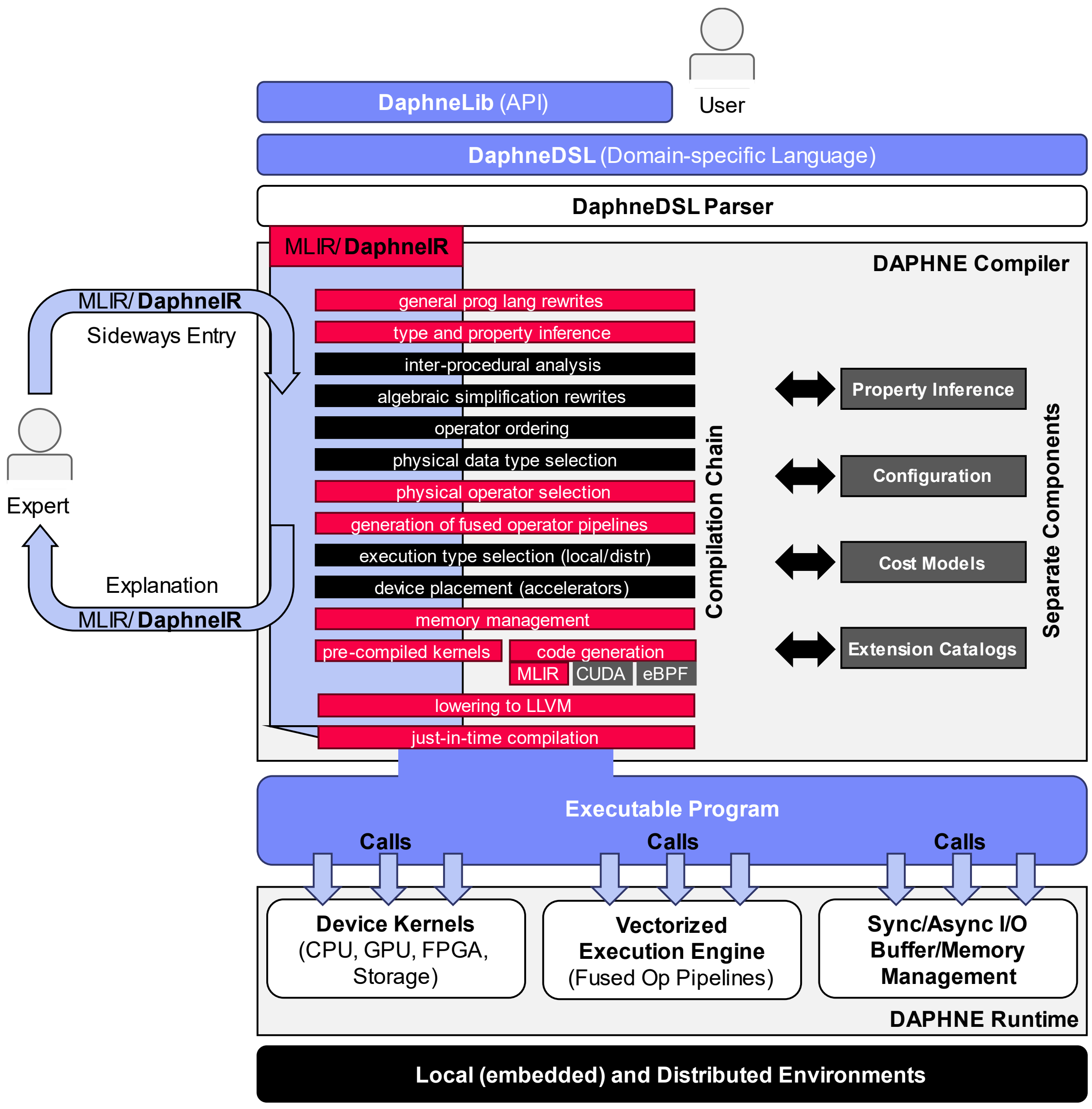
User
DaphneLib (API)
DaphneDSL (Domain-specific Language)
DaphneDSL Parser
MLIR/DaphneIR
DAPHNE Compiler
MLIR/DaphneIR
Sideways Entry
Expert
Explanation
MLIR/DaphneIR
general prog lang rewrites
type and property inference
inter-procedural analysis
algebraic simplification rewrites
operator ordering
physical data type selection
physical operator selection
generation of fused operator pipelines
execution type selection (local/distr)
device placement (accelerators)
memory management
pre-compiled kernels
code generation
MLIR
CUDA
eBPF
lowering to LLVM
just-in-time compilation
Compilation Chain
Property Inference
Configuration
Cost Models
Extension Catalogs
Separate Components
Executable Program
Calls
Calls
Calls
Device Kernels (CPU, GPU, FPGA, Storage)
Vectorized Execution Engine (Fused Op Pipelines)
Sync/Async I/O Buffer/Memory Management
DAPHNE Runtime
Local (embedded) and Distributed Environments

Figure 1: The DAPHNE compiler in the context of the system architecture [D3.4]. The compilation steps shown in this section are highlighted in red.

Initial DaphneIR after DaphneDSL parsing. The initial, unoptimized IR produced by the DaphneDSL parser can be inspected by:

```
bin/daphne --explain parsing \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that we also set verbose to false to avoid some unnecessary output by the script. At this stage, the IR is rather lengthy (~430 lines) and not very comfortable to look at. The complete IR obtained by the command above can be found in `scenario1/ir_01_parsing.txt`.

Initial simplification. The DAPHNE compiler chain starts by applying a first round of straightforward simplifications, including general programming language rewrites such as constant folding, common sub-expression elimination, and a few reorderings (e.g., moving constants to the top of the IR). The IR after this stage can be viewed by:

```
bin/daphne --explain parsing_simplified \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

The complete output can be found in `scenario1/ir_02_parsing_simplified.txt`. Now the IR has been shortened to ~330 lines, and we can have a first look at it.

```
IR after parsing and some simplifications:
module {
  func.func @"lmDS-1"(%arg0: !daphne.Matrix<?x?xf64>, %arg1: !daphne.Matrix<?x?xf64>, ...) ->
!daphne.Matrix<?x?xf64> {
    ...
    "daphne.return"(%64) : (!daphne.Matrix<?x?xf64>) -> ()
  }
  func.func @main() {
    ...
    %7 = "daphne.constant"() {value = "data/wine.csv"} : () -> !daphne.String
    %8 = "daphne.read"(%7) : (!daphne.String) -> !daphne.Matrix<?x?x!daphne.Unknown>
    ...
    %11 = "daphne.sliceCol"(%8, ...) : (!daphne.Matrix<?x?x!daphne.Unknown>, ...) ->
!daphne.Matrix<?x?x!daphne.Unknown>
    ...
    %14 = "daphne.sliceCol"(%8, ...) : (!daphne.Matrix<?x?x!daphne.Unknown>, ...) ->
!daphne.Matrix<?x?x!daphne.Unknown>
    %15 = "daphne.generic_call"(%11, %14, ...) {callee = "lmDS-1"} :
(!daphne.Matrix<?x?x!daphne.Unknown>, !daphne.Matrix<?x?x!daphne.Unknown>, ...) ->
!daphne.Matrix<?x?xf64>
    ...
  }
}
```

The IR consists of a single module containing two functions. First, the function `lmDS-1` is the imported `lmDS` function from DaphneDSL. Second, `main` is the entry point to the program and collects all DaphneDSL statements that are not part of any DaphneDSL function. The high-level steps in the main function are: (1) read the file `data/wine.csv` as a `daphne.Matrix`, (2) extract the feature matrix (X in DaphneDSL) and labels vector (y in DaphneDSL) using the `daphne.sliceCol` operation, and (3) call the function `lmDS-1` with these two. Furthermore, we can see that at this stage of the IR, information on the value types and shapes (the number of rows and the number of columns of a matrix or frame) of the matrices is still unknown, indicated by the type `!daphne.Matrix<?x?x!daphne.Unknown>`.

Type and property inference. One of the next steps in the compilation chain is the inference and propagation of data types and value types as well as interesting data properties (such as the shape, i.e., the number of rows and the number of columns of a matrix or frame). We interleave the inference with constant folding and other simplification rewrites, which are expressed as canonicalizations in MLIR. The reason is that type/property inference and constant propagation cyclically depend on each other: e.g., `nrow(X)` can only be constant-folded once the shape of `X` is known, and the shape of `fill(1.0, n, 1)` can only be inferred once the constant n is known. The IR after this stage can be viewed by:

```
bin/daphne --explain property_inference --no-ipa-const-propa \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that by `--no-ipa-const-propa`, we turn off a feature of inter-procedural analysis, to which we will come back shortly. The complete IR still has a length of ~310 lines and can be found in `scenario1/ir_03_property_inference.txt`.

```
IR after inference:
module {
  func.func @"lmDS-1-1"(%arg0: !daphne.Matrix<4898x11xf64>, %arg1: !daphne.Matrix<4898x1xf64>, ...) ->
!daphne.Matrix<?x1xf64> {
    ...
    %46 = scf.if %44 -> (!daphne.Matrix<4898x?xf64>) {
      %69 = "daphne.colBind"(%arg0, %38) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
      %70 = "daphne.cast"(%69) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<4898x?xf64>
      scf.yield %70 : !daphne.Matrix<4898x?xf64>
    } else {
      %69 = "daphne.cast"(%arg0) : (!daphne.Matrix<4898x11xf64>) -> !daphne.Matrix<4898x?xf64>
      scf.yield %69 : !daphne.Matrix<4898x?xf64>
    }
    ...
    "daphne.return"(%68) : (!daphne.Matrix<?x1xf64>) -> ()
  }
  func.func @main() {
    ...
    %9 = "daphne.constant"() {value = "data/wine.csv"} : () -> !daphne.String
    %10 = "daphne.read"(%9) : (!daphne.String) -> !daphne.Matrix<4898x12xf64:sp[1.000000e+00]>
    %11 = "daphne.sliceCol"(%10, %7, %1) : (!daphne.Matrix<4898x12xf64:sp[1.000000e+00]>, si64, si64) -
> !daphne.Matrix<4898x11xf64>
    %12 = "daphne.sliceCol"(%10, %1, %0) : (!daphne.Matrix<4898x12xf64:sp[1.000000e+00]>, si64, si64) -
> !daphne.Matrix<4898x1xf64>
    %13 = "daphne.generic_call"(%11, %12, ...) {callee = "lmDS-1-1"} : (!daphne.Matrix<4898x11xf64>,
!daphne.Matrix<4898x1xf64>, ...) -> !daphne.Matrix<?x?xf64>
    ...
  }
}
```

In this step, we apply both intra-procedural and inter-procedural analyses. In terms of intra-procedural analysis, the value types and shapes inside the main function have now become known by propagating the known value types and shapes of the input data over the involved operations. That way, the shapes of the inputs to the `lmDS-1` function have become known, too. This knowledge enables inter-procedural analyses. In particular, the `lmDS-1` function has been specialized for the given input shapes. This is visible in the slightly changed function name (`lmDS-1-1`) and in the fact that the shapes of the parameters are known now. Note that, being unused, the original `lmDS-1` function has been removed. Inside of the `lmDS-1-1` function, the DAPHNE compiler performs intra-procedural analysis again. Thus, properties like shapes are known inside this function as well. Nevertheless, especially in the presence of conditional control flow, the shape of a data object might not be unambiguously known. As an example, consider the intercept mode of `lmDS`. In `lmDS_.daph`, line 56ff, a column of ones is appended to the feature matrix if the intercept mode is 1 or 2, but not if the intercept mode is 0. The number of columns after this if-statement is either 11 or 12 with the given input data. In such a case of disagreement, the DAPHNE compiler conservatively assumes the number of columns to be unknown after the branching. Therefore, the shape of the result of the `scf.if` operation is `4898x?`. To illustrate this situation, we explicitly added the DAPHNE compiler flag `--no-ipa-const-propa` above.

However, by default, DAPHNE also propagates compile-time constants into functions. To see the effect, we next invoke DAPHNE by:

```
bin/daphne --explain property_inference \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that we omitted the flag `--no-ipa-const-propa`. The complete IR can be found in `scenario1/ir_04_property_inference_alternative.txt`, as well as in Appendix 2. Now, the constant arguments `icpt` (1) and `verbose` (false) have been inserted while specializing the `lmDS-1` function to obtain the `lmDS-1-1` function. In combination with constant folding, the knowledge of the constants unlocked a range of traditional compiler optimizations DAPHNE directly inherits from MLIR, most importantly (in this case) branch removal. We can see that the entire body of the `lmDS-1-1` function is now a *purely sequential program* since all conditional control flow depending on the intercept and verbosity level has been resolved at compile-time. This does not only make the IR more human-readable at ~60 lines but can also make the program execution more efficient by unlocking further optimization opportunities as we will see later. As a result of the full intra- and inter-procedural analysis, the shape of the output of the `lmDS-1-1` function is now also known to be `12x1` (for the intercept 1).

```
IR after inference:
module {
  func.func @"lmDS-1-1"(%arg0: !daphne.Matrix<4898x11xf64>, %arg1: !daphne.Matrix<4898x1xf64>, ...) ->
!daphne.Matrix<12x1xf64> {
    ...
    "daphne.return"(%38) : (!daphne.Matrix<12x1xf64>) -> ()
  }
  func.func @main() {
    ...
    %10 = "daphne.read"(%9) : (...) -> !daphne.Matrix<4898x12xf64:...>
    %11 = "daphne.sliceCol"(%10, %7, %1) : (!daphne.Matrix<4898x12xf64:...>, si64, si64) ->
!daphne.Matrix<4898x11xf64>
    %12 = "daphne.sliceCol"(%10, %1, %0) : (!daphne.Matrix<4898x12xf64:...>, si64, si64) ->
!daphne.Matrix<4898x1xf64>
    %13 = "daphne.generic_call"(%11, %12, ...) {callee = "lmDS-1-1"} : (!daphne.Matrix<4898x11xf64>,
!daphne.Matrix<4898x1xf64>, ...) -> !daphne.Matrix<12x1xf64>
    ...
  }
}
```

Physical operator selection. After type and property inference have yielded more information on the intermediate results and simplification rewrites have simplified the IR, the DAPHNE compiler moves on to physical steps. Depending on the properties of the input data, some DaphneIR operations can be executed by specific physical operators for better efficiency than a default operator. The result of this operator selection can be viewed by:

```
bin/daphne --explain phy_op_selection \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

The complete IR can be found in `scenario1/ir_05_phy_op_selection.txt`. Compared to the previous step, there are two decisive changes: The bulk of the work in `lmDS` is done by two matrix multiplications (denoted by the @-operator) `A = t(X) @ X;` and `b = t(X) @ y;` in `lmDS_.daph`. Interestingly, for both there are more specialized physical operators available: `t(X) @ X` can be executed by a symmetric rank-k operation, and `t(X) @ y` does not require a matrix-matrix multiplication, but just a less expensive matrix-vector multiplication. The DAPHNE compiler successfully selects these physical operators.

```
IR after inference:
...
%19 = "daphne.colBind"(%arg0, %16) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
...
%33 = "daphne.transpose"(%19) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<12x4898xf64>
%34 = "daphne.matMul"(%33, %19, %0, %0) : (!daphne.Matrix<12x4898xf64>, !daphne.Matrix<4898x12xf64>,
i1, i1) -> !daphne.Matrix<12x12xf64>
%29 = "daphne.matMul"(%33, %arg1, %0, %0) : (!daphne.Matrix<12x4898xf64>, !daphne.Matrix<4898x1xf64>,
i1, i1) -> !daphne.Matrix<12x1xf64>
```

```
IR after selecting physical operators:
...
%18 = "daphne.colBind"(%arg0, %15) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
...
%31 = "daphne.ewMul"(%24, %6) : (!daphne.Matrix<12x1xf64>, f64) -> !daphne.Matrix<12x1xf64>
%32 = "daphne.syrk"(%18) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<12x12xf64>
%33 = "daphne.gemv"(%18, %arg1) : (!daphne.Matrix<4898x12xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<12x1xf64>
```

While such decisions could also be made by the `MatMul`-kernel at runtime in certain cases, doing it already at the compiler-level allows further optimization, e.g., w.r.t. the access pattern in vectorized execution, which we illustrate next and see again in Section 3.4.

Generation of fused operator pipelines for vectorized execution. As one of the next steps, DAPHNE routinely identifies sequences of DaphneIR operations for fine-grained operator fusion and vectorized/tiled execution in so-called vectorized pipelines. For this purpose, DaphneIR operations supporting vectorized execution implement a DAPHNE-custom MLIR interface to provide information on how each argument can be split and how each result can be combined. In a special *vectorization pass*, the DAPHNE compiler identifies these operations through their interface as well as producer-consumer-relationships between them through MLIR's means for querying the def-use-chains of values in the IR. In the beginning, each vectorizable operation constitutes a separate pipeline. Then, pipelines are greedily fused together if there is a consumer-producer-relationship between them and the result of the producing pipeline is combined along the same axis as the argument of the consuming pipeline is split. The output of this step can be viewed by:

```
bin/daphne --vec --explain vectorized \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that we added the --vec flag to turn on vectorization. The resulting IR can be found in `scenario1/ir_06_vectorized.txt`. Inside the body of the function `lmDS-1-1` we now find three `daphne.vectorizedPipeline` operations, the most interesting of which is shown below.

```
IR after vectorization:
...
func.func @"lmDS-1-1"(...) -> ... {
  ...
  %31:2 = "daphne.vectorizedPipeline"(%arg0, %15, %arg1, ...) ({
  ^bb0(%arg5: !daphne.Matrix<?x11xf64>, %arg6: !daphne.Matrix<?x1xf64>, %arg7:
!daphne.Matrix<?x1xf64>):
    %37 = "daphne.colBind"(%arg5, %arg6) : (!daphne.Matrix<?x11xf64>, !daphne.Matrix<?x1xf64>) ->
!daphne.Matrix<?x?xf64>
    %38 = "daphne.gemv"(%37, %arg7) : (!daphne.Matrix<?x?xf64>, !daphne.Matrix<?x1xf64>) ->
!daphne.Matrix<?x?xf64>
    %39 = "daphne.syrk"(%37) : (!daphne.Matrix<?x?xf64>) -> !daphne.Matrix<?x?xf64>
    "daphne.return"(%38, %39) : (!daphne.Matrix<?x?xf64>, !daphne.Matrix<?x?xf64>) -> ()
  }, {
  }) {combines = [3, 3], ..., splits = [1, 1, 1]} : (...) -> (!daphne.Matrix<12x1xf64>,
!daphne.Matrix<12x12xf64>)
}
```

Here, the concatenation (`colBind`) of the feature matrix and the intercept vector, the symmetric rank-k operation (`syrk`), and the general matrix-vector multiplication (`gemv`) have been fused together into one pipeline, since each of them can be vectorized by splitting the arguments into row segments and by combining the results through row segment concatenation. This pipeline scans over the large feature matrix once and processes cache-conscious chunks in parallel. The creation of vectorized pipelines constitutes one of the **most important connection points** between the DAPHNE compiler (WP3) and the runtime (WP4). Moreover, inside a pipeline, the shapes of the data objects may be unknown since they are subject to efficient runtime scheduling (WP5).

At this point, we briefly come back to inter-procedural constant propagation again, as promised above. If we turn off this feature, the control flow cannot fully be resolved at compile-time, which limits the fusion opportunities. Indeed, the three operations `colBind`, `syrk`, and `gemv` end up in separate pipelines, which the interested reader can verify by running the following command or by viewing the complete IR, which can be found in the file `scenario1/ir_07_vectorized.txt`.

```
bin/daphne --vec --no-ipa-const-propa --explain vectorized \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Memory management. In close collaboration of WP3 and WP4, DAPHNE manages its memory usage and makes sure all allocations thereof are ultimately freed. DAPHNE's memory management concerns two levels: On the one hand, DAPHNE data objects like `DenseMatrix`, `CSRMatrix`, and `Frame` are *shallow objects* containing meta data and pointers to the underlying data buffers. On the other hand, the underlying data buffers contain the *actual data*. The data buffers (or ranges thereof) can reside on the host memory and/or the memories of hardware accelerators and remote nodes in a distributed setup. The existence of data buffers is always tied to a data object holding C++ `std::shared_ptr`'s to them, whereby *one data buffer can be shared by multiple data objects* (e.g., a `Frame` that was created from multiple `DenseMatrix`'s for its columns or a zero-copy view into a `DenseMatrix`). While the level of the data buffers is managed entirely by the DAPHNE runtime, the level of the data objects needs assistance from the DAPHNE compiler. Each data object has a reference counter that is initially one. The decisive challenge is to identify the point *when a data object is no longer needed* and can safely be freed. For this purpose, the DAPHNE compiler exploits its global view on the program to insert operations to increase the reference counter (`incRef`) each time an object is passed to a new scope (e.g., in a function call), and to decrease the reference counter (`decRef`) after the last use of a data object in each scope. Once the reference counter becomes zero, the object is freed. Since the extended compiler prototype [D3.3], we have also introduced reference counters for string scalars as these would otherwise cause memory leaks that can hurt long-running DaphneDSL scripts. The IR after this step can be viewed by:

```
bin/daphne --explain obj_ref_mgnt \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that, for better readability, we omit the `--vec` flag again. The complete IR can be found in `scenario1/ir_08_obj_ref_mgnt.txt`. In the `main` function, we can see that the matrix read from the CSV file is freed right after the two `sliceCol` operations which separate features from labels. As the feature matrix and labels are passed to the function `lmDS-1-1`, their references are increased to avoid double-frees, since the runtime objects also get memory-managed inside that function. Afterwards, their references are decreased to free them. The result of the function call is freed only after it has been printed to the console.

```
IR after managing object references:
...
func.func @main() {
  ...
  %15 = "daphne.read"(...) : (...) -> ...
  "daphne.decRef"(%0) : (...) -> ()
  %16 = "daphne.sliceCol"(%15, ...) : (...) -> ...
  %17 = "daphne.sliceCol"(%15, ...) : (...) -> ...
  "daphne.decRef"(%15) : (...) -> ()
  "daphne.incRef"(%16) : (...) -> ()
  "daphne.incRef"(%17) : (...) -> ()
  %18 = "daphne.generic_call"(%16, %17, ...) {callee = "lmDS-1-1"} : (...) -> ...
  "daphne.decRef"(%17) : (...) -> ()
  "daphne.decRef"(%16) : (...) -> ()
  ...
  "daphne.print"(%18, ...) : (...) -> ()
  "daphne.decRef"(%18) : (...>) -> ()
  ...
}
```

Lowering DaphneIR operations to pre-compiled kernels. By default, DAPHNE lowers all domain-specific operations (e.g., from linear algebra and relational algebra) to *calls to pre-compiled kernel functions* written in C++. This is done as one of the last steps of the compilation chain. The lowering to kernel calls is **one of the most decisive connections to the DAPHNE runtime** and WP4 [D4.4]. The IR after this step can be viewed by:

```
bin/daphne --explain kernels \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

The complete IR can be found in `scenario1/ir_09_kernels.txt`. As an example, the `daphne.syrk` operation has been lowered as follows:

```
IR after kernel lowering:
...
%37 = "daphne.call_kernel"(%23, %c54_i32, %19) {callee =
"_syrk__DenseMatrix_double__DenseMatrix_double"} : (!daphne.Matrix<4898x12xf64>, i32,
!daphne.DaphneContext) -> !daphne.Matrix<12x12xf64>
...
```

Here, `_syrk__DenseMatrix_double__DenseMatrix_double` is a specific kernel function that internally calls a BLAS routine. Since the extended compiler prototype [D3.3], we have added a unique id that we pass to each kernel call (`%c54_i32` in this example). This id is generated by the DAPHNE compiler and used by the DAPHNE runtime to map a kernel back to a DaphneDSL source code location. This information is used for expressive error messages (see Section 6). Moreover, the mapping from an DaphneIR operation to the name of the kernel function to call was still established through a naming convention for kernel functions. Meanwhile, we introduced a *kernel extension catalog* [D3.4], a DAPHNE-compiler-internal data structure that establishes an explicit mapping from DaphneIR operations and their argument/result types to the names of pre-compiled kernel functions. The kernel catalog is populated from JSON files of a specific format at DAPHNE start-up. In fact, even DAPHNE's built-in kernels are integrated through its extensibility mechanisms. The catalog file of the built-in kernels can be found in `lib/catalog.json` (after DAPHNE has been built once). The entry required to map the `daphne.syrk` operation above is the following:

```
[
  ...
  {
    "opMnemonic": "syrk",
    "kernelFuncName": "_syrk__DenseMatrix_double__DenseMatrix_double",
    "resTypes": ["DenseMatrix<double>"],
    "argTypes": ["DenseMatrix<double>"],
    "backend": "CPP",
    "libPath": "libAllKernels.so"
  },
  ...
]
```

The DAPHNE compiler performs a n:1 mapping (ignoring shapes and other data properties beyond the data/value type) of the argument/result types between MLIR types (such as `!daphne.Matrix<4898x12xf64>`, first argument of the `daphne.syrk` operation after physical operator selection above) to C++ types (such as `DenseMatrix<double>`, as specified in the catalog entry above). Besides the kernel function name, the entry also informs the DAPHNE compiler to link the program with the shared library `libAllKernels.so` (the library containing

DAPHNE's built-in kernels for CPU) during just-in-time compilation. We will see an example of adding a custom kernel in Section 4.

Lowering DaphneIR operations by low-level code generation. We described the design of code generation in general and MLIR-based code generation in particular in the compiler design and overview [D3.4]. As an alternative to lowering to calls to pre-compiled C++ kernels, code generation creates the low-level instructions necessary to perform a DaphneIR operation at compile-time. This approach can have numerous advantages, which we explained in detail in the DAPHNE compiler design and overview [D3.4]. Since the extended compiler prototype [D3.3], we have added an MLIR-based code generation backend, targeting CPU execution, to the DAPHNE compiler. At the time of writing, DAPHNE can generate code for an important subset of DaphneIR operations including elementwise binary operations (matrix-matrix, matrix-vector, and matrix-scalar for various op codes, e.g., addition, multiplication, etc.), full/row-wise/column-wise aggregation (for several op codes, e.g., sum, min/max, etc.), matrix multiplication, transpose, and `map()` (elementwise application of a scalar function) on dense matrices (see also Section 4) as well as selected examples of sparsity-exploiting code generation (see also Section 5).

DAPHNE's MLIR-based code generation backend tightly integrates with existing MLIR components and is comprised of multiple compiler passes, which can be grouped into two phases. In the first phase, DAPHNE-specific compiler passes lower DaphneIR operations to low-level MLIR operations from existing MLIR dialects like `linalg`, `affine`, `arith`, `memref`, etc. The IR after this step can be viewed by:

```
bin/daphne --mlir-codegen --explain mlir_codegen_daphneir_to_mlir \
        scripts/algorithms/lmDS.daph \
        XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

The complete IR can be seen in `scenario1/ir_10_mlir_codegen_daphneir_to_mlir.txt`.

```
IR after MLIR codegen (DaphneIR to MLIR):
#map = affine_map<(d0, d1) -> (d0, d1)>
...
  func.func @"lmDS-1-1"(...) -> ... {
    ...
    %6 = "daphne.constant"() {...} : () -> f64
    ...
    %24 = ... : (...) -> !daphne.Matrix<12x1xf64>
    ...
    %31 = "daphne.convertDenseMatrixToMemRef"(%24) : (!daphne.Matrix<12x1xf64>) -> memref<12x1xf64>
    %alloc = memref.alloc() : memref<12x1xf64>
    %intptr = memref.extract_aligned_pointer_as_index %alloc : memref<12x1xf64> -> index
    %32 = "daphne.convertMemRefToDenseMatrix"(%intptr, ...) : (index, ...) -> !daphne.Matrix<12x1xf64>
    linalg.generic {indexing_maps = [#map, #map], ...} ins(%31 ...) outs(%alloc : ...) {
    ^bb0(%in: f64, %out: f64):
      %42 = arith.mulf %in, %6 : f64
      linalg.yield %42 : f64
    }
    %33 = "daphne.syrk"(%18) : (...) -> ...
    %34 = "daphne.gemv"(%18, %arg1) : (...) -> ...
...
```

For instance, above, in the IR after physical operator selection, we can see a `daphne.ewMul` operation right before the `daphne.syrk` operation. `daphne.ewMul` is an elementwise multiplication of two values, and this particular occurrence multiplies all elements of a 12x1 matrix of `f64` values by an `f64` scalar. This `daphne.ewMul` operation is now lowered to a `linalg.generic` operation from the `linalg` MLIR dialect. `linalg.generic` can express a wide range of linear algebra operations using affine maps to specify how to iterate over the arguments and results, while the operation's body contains the operations performed on the individual elements. In this case, the operation performs an `arith.mulf` operation on an element `%in` of the input matrix and the given scalar value `%6`.

As both pre-compiled kernels and code generation have their own advantages, the DAPHNE compiler can combine both alternatives in one IR. This creates a unique **challenge of interoperability** between the C++-backed runtime data objects (i.e., matrices and frames) produced and consumed by DAPHNE's pre-compiled kernels and the data produced and consumed by MLIR operations from dialects like `linalg` (so-called `memrefs`). To solve these issues, we introduced dedicated DaphneIR operations for converting between DAPHNE data objects and MLIR memrefs at run-time, e.g., the operation `daphne.convertDenseMatrixToMemRef` applied to the argument matrix `%24` and the operation `daphne.convertMemRefToDenseMatrix` applied to the allocated output memref of the `linalg.generic` operation.

Lowering DaphneIR operations to MLIR's `linalg` dialect enables us to benefit from all existing MLIR lowering passes and optimization on and below the `linalg` dialect. Thus, to further lower the generated code in the second phase, we rely solely on existing MLIR passes. The IR after these additional lowering steps can be viewed by:

```
bin/daphne --mlir-codegen --explain mlir_codegen_mlir_specific \
      scripts/algorithms/lmDS.daph \
      XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

The complete IR can be found in `scenario1/ir_11_mlir_codegen_mlir_specific.txt`. At this level, we see the actual *for-loops* that iterate over the input matrix (`scf.for` operation) as well as the `memref.load` and `memref.store` operations that load/store the elements.

```
IR after MLIR codegen (MLIR-specific):
...
    %31 = "daphne.convertDenseMatrixToMemRef"(%24) : (!daphne.Matrix<12x1xf64>) -> memref<12x1xf64>
    %alloc = memref.alloc() : memref<12x1xf64>
    %intptr = memref.extract_aligned_pointer_as_index %alloc : memref<12x1xf64> -> index
    %32 = "daphne.convertMemRefToDenseMatrix"(%intptr, ...) : (index, ...) -> !daphne.Matrix<12x1xf64>
    scf.for %arg5 = %c0 to %c12 step %c1 {
      scf.for %arg6 = %c0 to %c1 step %c1 {
        %42 = memref.load %31[%arg5, %arg6] : memref<12x1xf64>
        %43 = arith.mulf %42, %6 : f64
        memref.store %43, %alloc[%arg5, %arg6] : memref<12x1xf64>
      }
    }
    %33 = "daphne.syrk"(%18) : (...) -> ...
    %34 = "daphne.gemv"(%18, %arg1) : (...) -> ...
```

We will see more examples of DAPHNE's MLIR-based code generation in Sections 4 and 5.

Lowering to LLVM and JIT-compilation. The last step of DAPHNE's compilation chain is the lowering to MLIR's `llvm` dialect. The IR after this step can be viewed by:

```
bin/daphne --vec --explain llvm \
  scripts/algorithms/lmDS.daph \
  XY=\"data/wine.csv\" reg=0.0000001 icpt=1 verbose=false
```

Note that we reinserted the `--vec` flag to show some interesting aspects. The complete IR can be found in `scenario1/ir_10_llvm.txt`. The lowering is largely done by existing MLIR conversion patterns. Nevertheless, some DaphneIR operations require special treatment. For instance, the body of a `daphne.vectorizedPipeline` is turned into an `llvm.func` and a pointer to this function is passed to the `vectorizedPipeline` kernel of the DAPHNE runtime. This can only be done at such a low level as the `llvm` dialect. As an example, consider the pipeline consisting of `colBind`, `syrk`, and `gemv` mentioned above.

```
IR after llvm lowering:
module {
  ...
  llvm.func @_vect2(%arg0: !llvm.ptr<ptr<ptr<i1>>>, %arg1: !llvm.ptr<ptr<i1>>, %arg2: !llvm.ptr<i1>) {
    ...
    llvm.call @_colBind__DenseMatrix_double__DenseMatrix_double__DenseMatrix_double(...) : (...) -> ()
    ...
    llvm.call @_gemv__DenseMatrix_double__DenseMatrix_double__DenseMatrix_double(...) : (...) -> ()
    ...
    llvm.call @_syrk__DenseMatrix_double__DenseMatrix_double(...) : (...) -> ()
    ...
  }
  ...
  llvm.func @"lmDS-1-1"(...) -> ... attributes {...} {
    ...
    %224 = llvm.mlir.addressof @_vect2 : ...
    ...
    %260 = llvm.bitcast %224 : ... to ...
    llvm.store %260, %50 : ...
    ...
    llvm.call
@_vectorizedPipeline__DenseMatrix_double_variadic__size_t__bool__Structure_variadic__size_t__int64_t__i
nt64_t__int64_t__int64_t__size_t__void_variadic(...) : (...) -> ()
    ...
  }
  ...
  llvm.func @main() attributes {...} {
    ...
  }
  ...
}
```

The body of this pipeline now became the `_vect2` function and a pointer to this function is obtained and passed to the `_vectorizedPipeline__`… kernel inside the function `lmDS-1-1`.

Summary. In this section we have taken a tour through some of the most important steps of DAPHNE's compilation chain from the initial DaphneIR after parsing a DaphneDSL script over various lowering and optimization steps down to the lowering to pre-compiled kernels or MLIR-based code generation and, ultimately, to the lowering to LLVM IR and JIT compilation. While the ideas behind these steps have been explained in detail in the DAPHNE compiler design and overview [D3.4], we have now seen concrete examples of the DaphneIR that reveal the effect of the individual compilation steps.

3.4 Experimental Results

To illustrate the impact of the compiler passes presented above on the compilation and execution time of DAPHNE, we conducted a series of micro benchmarks comparing the DS and CG method of linear regression model training on two double-precision input data sets with 1 million rows and either 100 (800 MB) or 1000 (8 GB) columns. These data sets are randomly generated at run-time, such that all experiments happen entirely in main memory and secondary storage is never accessed during the experiments. We experiment with all three options for the intercept.

The experiments were conducted on a server equipped with an Intel Xeon Gold 6338 CPU clocked at 2 GHz. This processor has two sockets with 32 physical cores each, resulting in 128 logical cores due to hyper-threading. The L1 data, L1 instruction, L2, and L3 caches have a total size of 3 MiB (32 KiB per core), 2 MiB (48 KiB per core), 80 MiB (1.25 MiB per core), and 96 MiB. The system is further equipped with 1 TiB of DDR4 memory, and during the experiments, all data resides in main memory. The operating system is Ubuntu 24.04.1 LTS GNU/Linux with kernel 6.8.0-48-generic. We compiled DAPHNE with g++ version 13.2.0. We repeated all time measurements 5 times and report the arithmetic means of all repetitions.

We use the default scheduling configuration for DAPHNE's vectorized engine, i.e., static task partitioning, one centralized queue, and a sequential victim selection (work stealing from the next adjacent worker). We refer the interested reader to deliverable D5.4 [D5.4] for further information on scheduling in DAPHNE. All experiments run purely locally, i.e., DAPHNE's distributed runtime was not used.

The experiments presented in the following can be reproduced using the scripts `scenario1/exp.sh` and `scenario1/dia.py`. The original results can be found in `scenario1/res.csv`. While we have already shown these experiments for the extended compiler prototype [D3.3], we re-ran them for the final compiler prototype, whereby the conclusions are largely the same.

Non-vectorized vs. vectorized execution. In the first experiment, we compare the execution time of the non-vectorized execution to the vectorized execution. We report only the execution times of the `lmDS` and `lmCG` functions here, i.e., the time for DaphneDSL parsing (negligible), compilation, and the random data generation are not included. The results are shown in Figure 2. With non-vectorized processing (blue bars), we can see that for 100 columns (upper row of diagrams), DS performs significantly better than CG, while for 1000 columns (lower row of diagrams) the advantage of DS is not as pronounced, which is expected. Vectorized processing is always faster than non-vectorized processing for the same method and data size. However, there is still room for improvement in terms of speed up, which can partly be attributed to the DAPHNE compiler and partly to the DAPHNE runtime. Indeed, DS benefits more from vectorization than CG at the moment.

Figure 2:

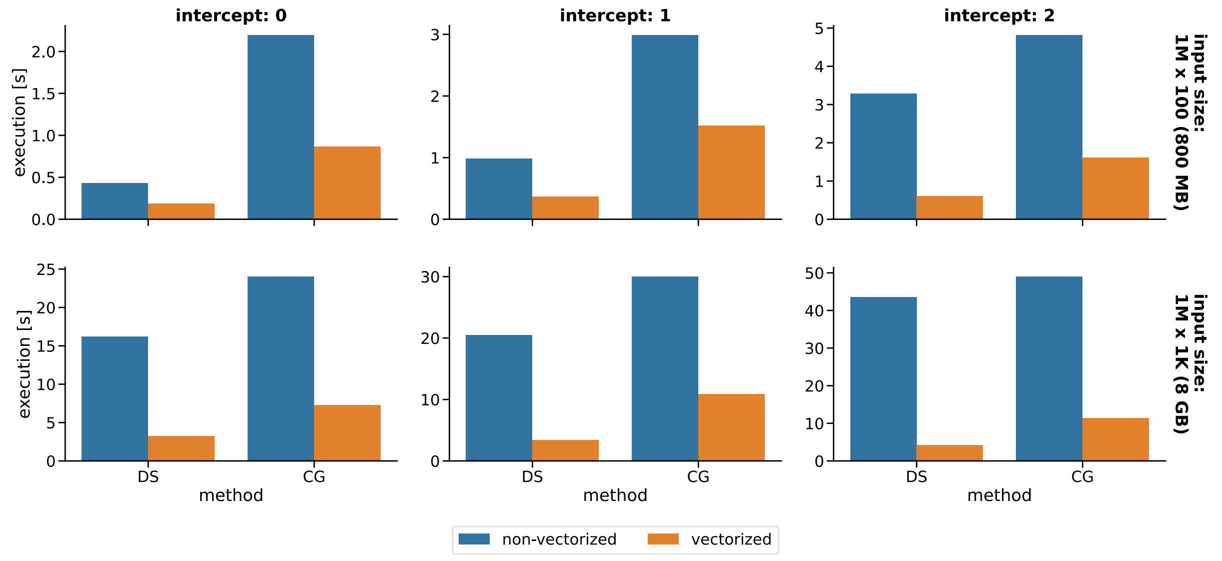

Figure 2: Linear regression model training: Execution time for non-vectorized vs. vectorized execution.

Figure 3 shows the compilation times, including all optimizations and lowering performed by the DAPHNE compiler as well as the LLVM just-in-time compilation. The compilation times are far lower than the execution times. Even more importantly, the compilation time does not depend on the input data size. Finally, vectorization does not have a significant impact on the compilation time; indeed, any additional cost is outweighed by the improvements in execution time, given non-trivial data sizes.

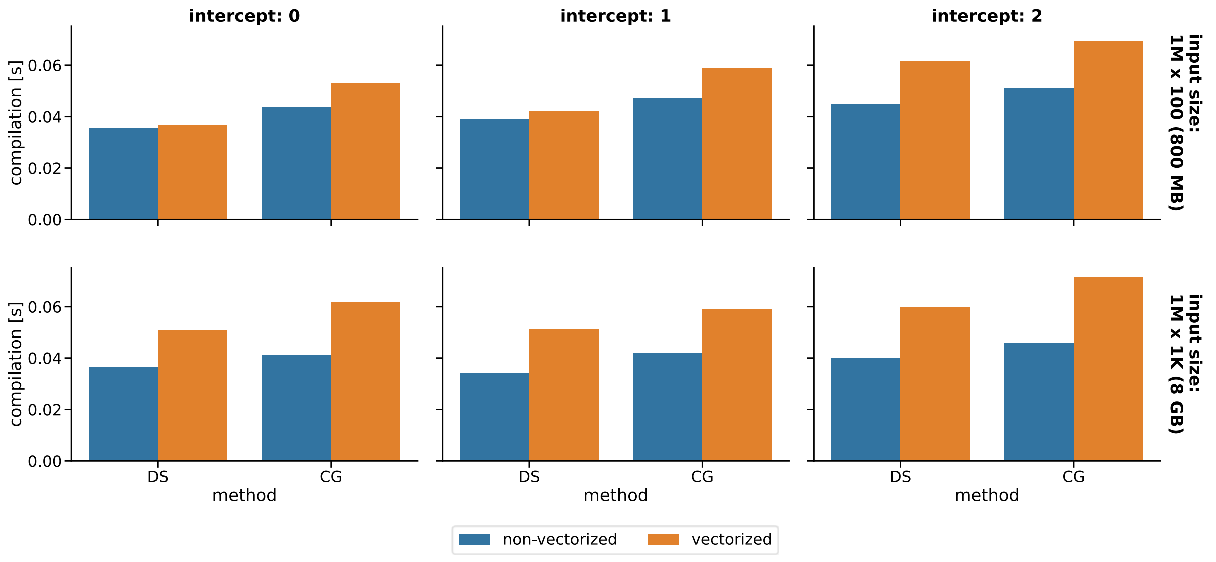

Figure 3: Linear regression model training: Compilation time for non-vectorized vs. vectorized execution.

Impact of compiler flags. In the second experiment, we revisit some remarks made in Section 3.3. For this purpose, we execute DAPHNE with different compiler flags. The results for the execution time of the `lmDS` and `lmCG` functions are shown in Figure 4. More precisely, we compare the default vectorized processing (blue bars) to two cases where we explicitly turned off decisive features of the DAPHNE compiler, namely inter-procedural constant propagation (orange bars, `--no-ipa-const-propa`, we mentioned before that this can result in less pipeline fusion opportunities) and physical operator selection (green bars, `--no-phy-op-`

`selection`, we mentioned before that this can result in suboptimal access patterns in vectorized processing). The figure below shows that turning off these compiler features can indeed cause a significantly worse performance, especially for the DS method.

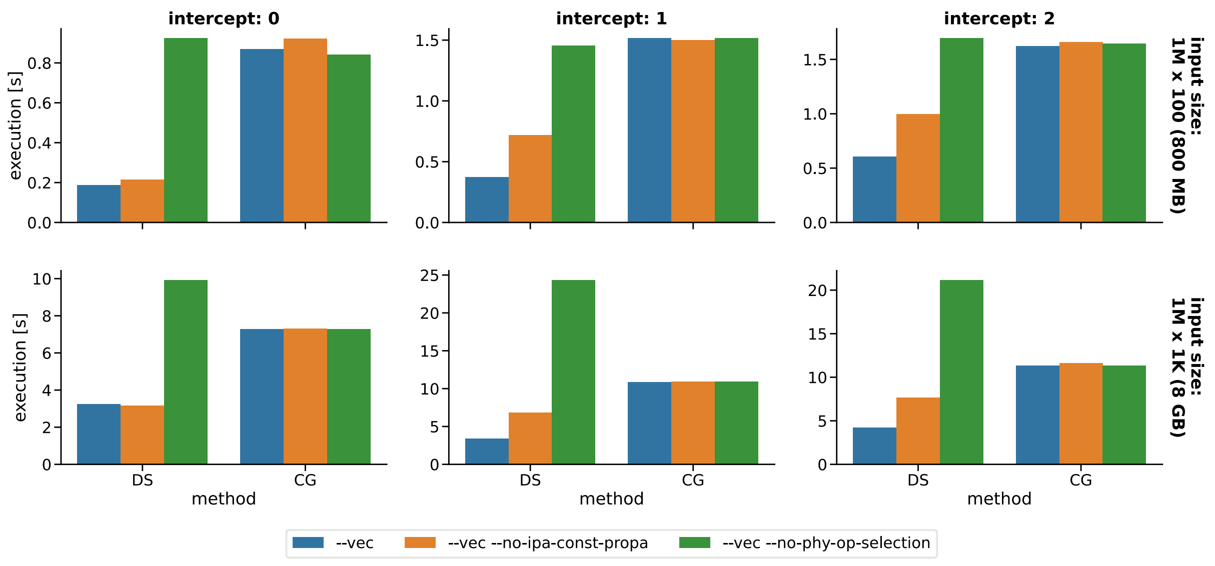

Figure 4: Linear regression model training: Execution time with different DAPHNE compiler flags.

Next, we also show the impact of these compiler flags on the compilation time. The results are shown in Figure 5. Omitting inter-procedural constant-propagation leads to an increased compilation time, since the IR stays unnecessarily verbose that way, thereby causing more effort for subsequent compiler passes. Omitting physical operator selection can slightly improve the compilation time; nevertheless, the extra effort is by far outweighed by the improvements in execution time.

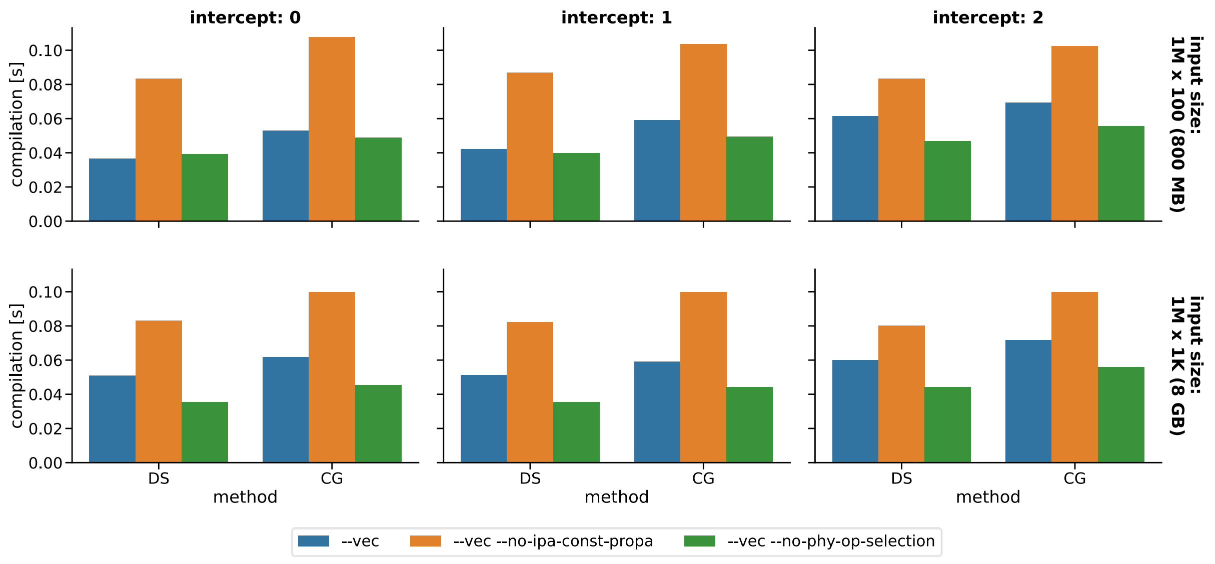

Figure 5: Linear regression model training: Compilation time with different DAPHNE compiler flags.

4 Scenario 2: Extensibility and Codegen / Data Preprocessing

4.1 Running Example: Shift-and-Scale (Data Preprocessing)

In the second demonstration scenario, we focus on a data preprocessing step frequently applied before training an ML model: shifting and scaling all values in a feature matrix to ensure a mean of zero and a standard deviation of one for each feature (column). This preprocessing step was also an optional part of the linear regression model training (Section 3.1), which was performed when the parameter `intercept` was set to 2. The DaphneDSL script looks as follows and can be found in `scenario2/shift-and-scale_nohints.daphne`. Note that we print the sum of the result matrix `Xshiftscale` in the end to prevent the DAPHNE compiler from optimizing away all computations (we do not print the entire matrix to avoid excessive output).

```
# Generate some random data.
X = rand($r, $c, 0.0, 100.0, 1, $seed);

# Pre-process the data: subtract the column-mean and divide by the column stddev.
colMeans = sum(X, 1) / nrow(X);
Xshift = X - colMeans;
colSDs = sqrt(sum(Xshift * Xshift, 1) / nrow(X));
Xshiftscale = Xshift / colSDs;

# For checking the result.
print("Done.");
print(sum(Xshiftscale));
```

We use this example to demonstrate two points: (1) expert users can use their own kernels in a DaphneDSL script through DAPHNE's extensibility mechanisms, and (2) DAPHNE's MLIR-based code generation backend can benefit the execution time of a DaphneDSL script.

4.2 Adding a Custom Kernel

Expert users can add their own custom kernels to DAPHNE and run them in the context of a full integrated data analysis pipeline. To that end, DAPHNE offers a three-step extensibility process [D3.4, DB23]. In the following, we demonstrate this process using a concrete example. Modern server-grade and desktop processors are usually equipped with instruction set extensions for SIMD (single instruction multiple data) instructions that process multiple data elements at once by using so-called vector registers. Examples of SIMD extensions include Intel SSE, AVX, and AVX-512 as well as ARM Neon and SVE. For our example, we will try to speed up the costliest operations in the shift-and-scale preprocessing (elementwise binary subtraction/multiplication/division and columnar sum) by explicitly employing AVX2 instructions, which operate on 256-bit vector registers, i.e., four double-precision floating-point elements at a time.

Step 1: Implementing the Extension. In the first step, the expert user implements their kernel(s) in a stand-alone code base, but against a clearly defined interface. In the simplest case, a single C++ source file can be sufficient for that. The AVX2-based kernels for our example can be implemented as follows (the complete source code can be found in `scenario2/SIMDKernels/SIMDKernels.cpp`).

```
#include <runtime/local/datastructures/DenseMatrix.h>
class DaphneContext;

#include <iostream.h>
#include <immintrin.h>
#include <cstdlib>

extern "C" {
void SIMDSumCol(DenseMatrix<double> **res, const DenseMatrix<double> *arg, DaphneContext *ctx) {
    std::cout << "Hello from SIMDSumCol!" << std::endl;

    // Validation.
    const size_t numRows = arg->getNumRows();
    const size_t numCols = arg->getNumCols();
    if (numCols % 4)
        throw std::runtime_error("for simplicity, the number of columns must be a multiple of 4");

    // Create output matrix.
    if (*res == nullptr)
        *res = DataObjectFactory::create<DenseMatrix<double>>(1, numCols, true);

    // SIMD accumulation per column (4x f64).
    const double *valuesArg = arg->getValues();
    double *valuesRes = (*res)->getValues();
    for (size_t r = 0; r < numRows; r++) {
        for (size_t c = 0; c < numCols; c += 4) {
            _mm256_storeu_pd(valuesRes + c, _mm256_add_pd(_mm256_loadu_pd(valuesArg + c),
                                                          _mm256_loadu_pd(valuesRes + c)));
        }
        valuesArg += numCols;
    }
}

void SIMDSub(DenseMatrix<double> ** res, const DenseMatrix<double> * lhs,
             const DenseMatrix<double> * rhs, int kId, DaphneContext *ctx) {...}

void SIMDDiv(DenseMatrix<double> ** res, const DenseMatrix<double> * lhs,
             const DenseMatrix<double> * rhs, int kId, DaphneContext *ctx) {...}

void SIMDMul(DenseMatrix<double> ** res, const DenseMatrix<double> * lhs,
             const DenseMatrix<double> * rhs, int kId, DaphneContext *ctx) {...}
}
```

At the top of the file, a few necessary headers from DAPHNE, such as the `DenseMatrix`, are included. Each of the four kernels we want to accelerate through SIMD instructions needs to be implemented as an individual C++ function. The functions are surrounded by an `extern "C"` block to ensure the right linkage such that the kernels can be called from the just-in-time-compiled DaphneDSL code later. The function names are arbitrary, and we call them `SIMDSumCol`, `SIMDSub`, `SIMDDiv`, and `SIMDMul` here. These functions consume and produce DAPHNE data objects (`DenseMatrix`, in this case), just like DAPHNE's built-in kernels. For a simple example, we show the code of the `SIMDSumCol` kernel here, which calculates the sum along each column of a given matrix `arg` and returns a row-vector `res`. In the first line, the kernel prints a hello-message to standard output, just so that we can easily see that the kernel actually runs in this demonstration. More importantly, the kernel makes use of SIMD intrinsics from the `immintrin.h` header: _mm256_loadu_pd (which loads a vector of four double-precision values from memory into a vector register), _mm256_add_pd (which adds the corresponding elements of two vector registers), and `_mm256_storeu_pd` (which stores a vector register to memory)[2].

The custom kernels must be compiled as a shared library. To this end, we provide the following Makefile. The full file can be found in `scenario2/SIMDKernels/Makefile`.

```
libSIMDKernels.so: SIMDKernels.o
	g++ -shared SIMDKernels.o -o libSIMDKernels.so

SIMDKernels.o: SIMDKernels.cpp
	g++ -c -fPIC SIMDKernels.cpp -I../../src -std=c++17 -O3 -mavx2 -o SIMDKernels.o
```

To build the extension library, we simply execute `make` in the directory `scenario3/SIMDKernels/`, which gives the following output and produces the file `libSIMDKernels.so`.

```
g++ -c -fPIC SIMDKernels.cpp -I../../src -std=c++17 -O3 -mavx2 -o SIMDKernels.o
g++ -shared SIMDKernels.o -o libSIMDKernels.so
```

Step 2: Registering the Extension with DAPHNE. To inform DAPHNE of the existence of the SIMDKernels extension and to provide essential information on the new kernels to the DAPHNE compiler, we require a simple JSON catalog file containing this information. An excerpt of this file can be seen below, while the full file can be found as part of the artifact in `scenario2/SIMDKernels/SIMDKernels.json`.

```
[
  {
    "opMnemonic": "sumCol",
    "kernelFuncName": "SIMDSumCol",
    "resTypes": ["DenseMatrix<double>"],
    "argTypes": ["DenseMatrix<double>"],
    "backend": "CPP",
    "libPath": "libSIMDKernels.so"
  },
  ...
]
```

This file has the same format as the kernel catalog file we have seen in Section 3.3. However, in this case, the catalog entry contains the C++ function name of our custom kernel from `SIMDKernels.cpp` as well as the name of our extension's shared library `libSIMDKernels.so`.

A kernel extension can be registered with the DAPHNE system at start-up, i.e., DAPHNE does not need to be built anew. To that end, the `--kernel-ext` command line argument is passed to the daphne executable, as we will see in the next step.

[2]For a full reference of Intel's SIMD intrinsics, please see the Intel Instrinsics Guide at https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Step 3: Using the Extension. To make use of a kernel extension, users can employ kernel hints in DaphneDSL. While hints circumvent the idea of automatic decisions made by the DAPHNE compiler and runtime, they are an invaluable tool for experimentation. For instance, hints can be used to find out how a DaphneDSL script behaves if a certain kernel is employed, even if the DAPHNE compiler does not choose the kernel on its own. In the future, we also plan to device automatic strategies for selecting among multiple registered kernels for the same DaphneIR operation and argument/result types, e.g., based on cost models. The shift-and-scale script from above can be augmented with kernel hints by adding the kernel name after a DaphneDSL built-in function or operator symbol, separated by double colon. The full script can be found in `scenario2/shift-and-scale_hints.daphne`.

```
# Generate some random data.
X = rand($r, $c, 0.0, 100.0, 1, -1);

# Pre-process the data: subtract the column-mean and divide by the column stddev.
colMeans = sum::SIMDSumCol(X, 1) / nrow(X);
Xshift = X -::SIMDSub colMeans;
colSDs = sqrt(sum::SIMDSumCol(Xshift *::SIMDMul Xshift, 1) /::SIMDDiv nrow(X));
Xshiftscale = Xshift / colSDs;

# For checking the result.
print("Done.")
print(sum(Xshiftscale));
```

Kernel hints are respected by the DAPHNE compiler, i.e., they remain untouched in early compiler passes. During the lowering of DaphneIR operations to calls to pre-compiled kernels (Section 3.3), these kernel hints override the look-up of a suitable kernel in the DAPHNE compiler's kernel catalog. Instead, the DAPHNE compiler merely checks if the given kernel hint is the name of a valid kernel function registered for the DaphneIR operation at hand and its argument/result types. We can double-check the effect of DaphneDSL kernel hints on the DAPHNE compilation chain through DAPHNE's explanation feature (note how we use the `--kernel-ext` argument to inform DAPHNE of the extension). The full IR and output can be found in `scenario2/ir_hints.txt`.

```
bin/daphne --kernel-ext scenario2/SIMDKernels/SIMDKernels.json \
  --explain parsing_simplified,kernels \
  scenario2/shift-and-scale_hints.daphne \
  r=1000 c=100 seed=12345
```

```
IR after parsing and some simplifications:
module {
  func.func @main() {
    ...
    %9 = "daphne.randMatrix"(...) : (...) -> ...
    %10 = "daphne.sumCol"(%9) {kernel_hint = "SIMDSumCol"} : (...) -> ...
    %11 = "daphne.numRows"(%9) : (...) -> index
    %12 = "daphne.ewDiv"(%10, %11) : (...) -> ...
    %13 = "daphne.ewSub"(%9, %12) {kernel_hint = "SIMDSub"} : (...) -> ...
    ...
  }
}IR after kernel lowering:
module {
  func.func @main() {
    ...
    %15 = "daphne.call_kernel"(...) {callee =
"_randMatrix__DenseMatrix_double__size_t__size_t__double__double__double__int64_t"} : (...) -> ...
    %c3_i32 = arith.constant 3 : i32
    %16 = "daphne.call_kernel"(%15, ...) {callee = "SIMDSumCol"} : (...) -> ...
    %c4_i32 = arith.constant 4 : i32
    %17 = "daphne.call_kernel"(...) {callee = "_ewDiv__DenseMatrix_double__DenseMatrix_double__double"}
: (...) -> ...
    %c5_i32 = arith.constant 5 : i32
    "daphne.call_kernel"(%16, %c5_i32, %14) {callee = "_decRef__Structure"} : (...) -> ()
    %c6_i32 = arith.constant 6 : i32
    %18 = "daphne.call_kernel"(%15, %17, %c6_i32, %14) {callee = "SIMDSub"} : (...) -> ...
    ...
  }
}
Hello from SIMDSumCol!
Hello from SIMDSub!
Hello from SIMDMul!
Hello from SIMDSumCol!
Hello from SIMDDiv!
Done.
-8.35376e-12
```

In the very early IR after parsing and some initial simplifications, we see that the expected DaphneIR operations have the kernel hint in the form of a custom MLIR attributed `kernel_hint` attached to them. After lowering to kernel calls, these are exactly the kernel functions that will be called, while the operations we did not provide with a hint, such as `daphne.rand` and the `daphne.ewDiv` by a scalar, are still backed by built-in kernels. Furthermore, in the output of the DaphneDSL script, we see the messages printed by our custom SIMD-based kernels.

4.3 MLIR-based Code Generation

We can also use DAPHNE's MLIR-based code generation backend for the shift-and-scale script. In fact, DAPHNE can generate low-level MLIR code for all operations required for the calculation. We can double-check this through DAPHNE's explain feature.

```
bin/daphne --mlir-codegen --explain mlir_codegen_mlir_specific \
  scenario2/shift-and-scale_nohints.daphne \
  r=1000 c=100 seed=12345
```

```
IR after MLIR codegen (MLIR-specific):
module {
  func.func @main() {
    ...
    %10 = "daphne.randMatrix"(...) : (...) -> !daphne.Matrix<1000x100xf64:sp[1.000000e+00]>
    %11 = "daphne.convertDenseMatrixToMemRef"(%10) : (...) -> memref<1000x100xf64>
    %alloc = memref.alloc() : memref<1x100xf64>
    ...
    scf.for %arg0 = %c0 to %c999 step %c1 {
      scf.for %arg1 = %c0 to %c100 step %c1 {
        %14 = arith.addi %arg0, %c1 : index
        %15 = memref.load %11[%14, %arg1] : memref<1000x100xf64>
        %16 = memref.load %alloc[%c0, %arg1] : memref<1x100xf64>
        %17 = arith.addf %16, %15 : f64
        memref.store %17, %alloc[%c0, %arg1] : memref<1x100xf64>
      }
    }
    %alloc_0 = memref.alloc() : memref<1x100xf64>
    scf.for %arg0 = %c0 to %c1 step %c1 {
      scf.for %arg1 = %c0 to %c100 step %c1 {
        %14 = memref.load %alloc[%arg0, %arg1] : memref<1x100xf64>
        %15 = arith.divf %14, %0 : f64
        memref.store %15, %alloc_0[%arg0, %arg1] : memref<1x100xf64>
      }
    }
    %alloc_1 = memref.alloc() : memref<1000x100xf64>
    scf.for %arg0 = %c0 to %c1000 step %c1 {
      scf.for %arg1 = %c0 to %c100 step %c1 {
        %14 = memref.load %11[%arg0, %arg1] : memref<1000x100xf64>
        %15 = memref.load %alloc_0[%c0, %arg1] : memref<1x100xf64>
        %16 = arith.subf %14, %15 : f64
        memref.store %16, %alloc_1[%arg0, %arg1] : memref<1000x100xf64>
      }
    }
    ...
  }
}
```

For brevity, we show only the first three operations from the DaphneDSL script, i.e., the columnar sum and the elementwise division involved in calculating the DaphneDSL variable `colMeans`, and the elementwise subtraction calculating `Xshift`. The full IR can be found in `scenario2/ir_codegen.txt`. The loops and scalar operations involved in them, like `arith.addf`, `arith.divf`, and `arith.subf`, can clearly be seen in the IR.

4.4 Experimental Results

After we have discussed different variants for compiling the shift-and-scale data preprocessing script in DAPHNE, we evaluate which impact the two discussed variants, i.e., (1) using a custom kernel extension of SIMD-based kernels and (2) using DAPHNE's MLIR-based code generation backend, have on the execution time of the algorithm. To this end, we randomly generated a dense matrix of double-precision floating-point values with 1 million rows and 1000 columns for the input data X, which implies a physical size of 8 GB. The data generation happens at run-time and fully in-memory, such that the secondary storage is never accessed.

The experiments were conducted on a server equipped with an AMD EPYC 7443P CPU with a base frequency of 2.85 GHz. This processor supports the AVX2 SIMD extension and has a single socket with 24 physical cores each, resulting in 48 logical cores due to hyper-threading. The L1 data, L1 instruction, L2, and L3 caches have a total size of 768 KiB (32 KiB per core), 768 KiB (32 KiB per core),12 MiB (512 KiB per core), and 128 MiB. The system is further equipped with 256

GiB of DDR4 memory, and during the experiments, all data resides in main memory. The operating system is Ubuntu 24.04.1 LTS GNU/Linux with kernel 6.8.0-48-generic. We compiled DAPHNE with g++ version 13.2.0. We measure only the time required for the shift-and-scale calculation itself, not the time required for the random number generation. We repeated all measurements 10 times and report only the mean.

The experiment ran purely in DAPHNE's local runtime and can be repeated using the scripts `scenario2/experiment.sh` and `scenario2/dia.py`. The original results can be found in `scenario2/res.csv` and `scenario2/dia.pdf`.

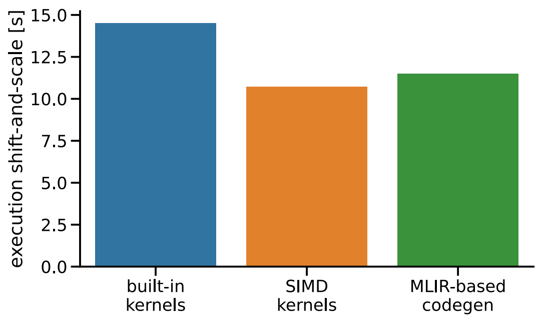

Figure 6: Shift-and-scale execution time with built-in kernels, extension kernels, and MLIR-based code generation.

The results are presented in Figure 6. When using DAPHNE's built-in pre-compiled kernels, the execution takes approximately 15 seconds. By using the custom SIMD-based kernels presented in Section 4.2, the execution can be accelerated to approximately 12 seconds. Likewise, compiling the DaphneDSL script with DAPHNE's MLIR-based code generation backend presented in Section 4.3 achieves a runtime of approximately 12 seconds. These results show that (a) employing custom kernels for the concrete hardware architecture can improve performance, and (b) DAPHNE's MLIR-based code generation backend is effective. However, our intension is not to make any claims about the suitability of SIMD-based kernels versus code generation; and in fact, all three variants could be further optimized. Instead, the main takeaway is that DAPHNE, as an open and extensible infrastructure for integrated data analysis pipelines, (1) does support different routes through the compilation chain (e.g., lowering to pre-compiled kernels or low-level code generation) and (2) enables expert users (e.g., researchers) to very easily employ their custom hardware-specific kernels in the context of an entire integrated data analysis pipeline, while benefitting from the surrounding compiler and runtime infrastructure. This overall setup makes DAPHNE a brilliant host system for further research on the efficient compilation and execution of integrated data analysis pipelines.

5 Scenario 3: Sparsity-exploiting Operator-Fusion / PNMF

5.1 Running Example: Poisson Nonnegative Matrix Factorization (PNMF)

In the third demonstration scenario, we consider Poisson Nonnegative Matrix Factorization (PNMF). This algorithm tries to represent a typically very sparse input matrix `X` by two factors `W` and `H` of a lower rank [BRH+18]. These factors are initialized with random numbers. Then, the algorithm works iteratively. In each iteration, the factors `W` and `H` are updated to better approximate `X`. The algorithm runs until either a maximum number of iterations has been performed or until the approximation is close enough. In the latter case, the quality of the approximation is typically evaluated by the cross entropy loss `obj = sum(X * log(W @ t(H)) + eps)`. In the following, we focus specifically on this sub-expression, as it offers optimization potential for sparsity-exploiting operator fusion, a feature we prototypically integrated into the DAPHNE compiler. The overall DaphneDSL script we consider in this section looks as follows and can also be found in `scenario3/cross-entropy-loss.daphne`.

```
# Generate random data with the specified dimensions and sparsity.
X = rand($n, $n, 0.0, 1.0, $sp, $seed);
U = rand($n, $k, 0.0, 1.0, 1, $seed);
V = rand($n, $k, 0.0, 1.0, 1, $seed);

# Calculate cross-entropy loss.
res = sum(X * ln(U @ t(V)));

# For checking the result.
print(res);
```

5.2 MLIR Code Generation for Sparsity-exploiting Operator Fusion

By default, the DAPHNE compiler lowers DaphneIR operations to calls to pre-compiled kernels for dense or sparse data. Alternatively, the low-level instructions for the DaphneIR operations can be generated on-the-fly, as we have seen in Section 4. Calculating the cross entropy loss this way results in the materialization of large dense intermediate results `U @ t(V)`, `U @ t(V) + eps`, and `log(U @ t(V) + eps)` in memory. However, most of the involved calculations are redundant, as the elementwise multiplication with `X`, which is performed last, effectively uses only a small subset of the elements of `log(U @ t(V) + eps)`. This redundancy can be avoided through a sparsity-exploiting operator fusion: The DAPHNE compiler automatically detects this pattern of operations and generates custom low-level MLIR operations for the entire pattern, rather than the individual operations. In particular, the code iterates only over the non-zero elements in `X`, for each such element, it calculates the dot product of the respective row in `U` and column in `t(V)` and applies the logarithm. That way, only the actually required elements are calculated and the large dense intermediates are completely avoided, as illustrated by Figure 7.

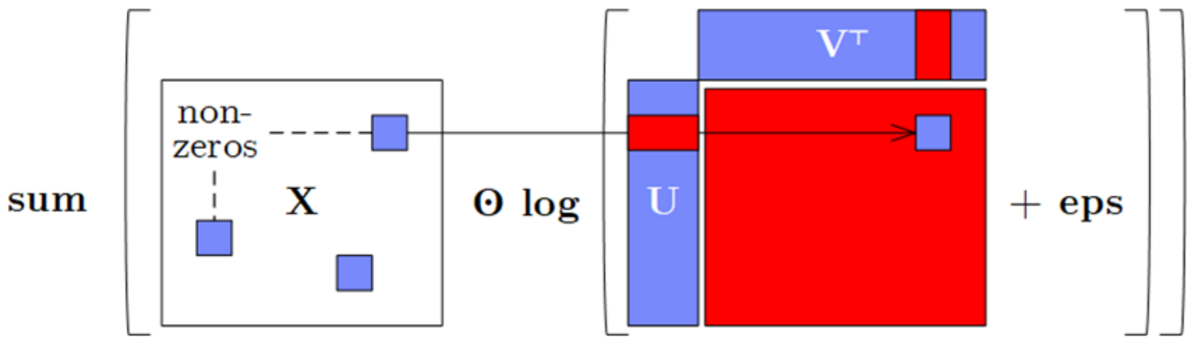

Figure 7: Illustration of the cross entropy loss calculation with a sparse input matrix X. Extracted from https://mboehm7.github.io/teaching/ss24_amls/04_AdvancedCompilation.pdf.

Sparsity-exploiting operator fusion is implemented as a separated pass in the DAPHNE compiler that runs by default as a part of DAPHNE's MLIR-based code generation backend. The effect of this pass on the IR can be examined using the DAPHNE compiler's explanation features.

To get familiar with the IR before sparsity-exploiting operator fusion, a decisive step to look at is the selection of physical data representations. We can view the IR at this point by the following command. The full IR can be found in `scenario3/ir_select_matrix_repr.txt`.

```
bin/daphne --select-matrix-repr --explain select_matrix_repr \
  scenario3/cross-entropy-loss.daphne n=20000 k=100 sp=0.0001 seed=12345
```

```
IR after selecting matrix representations:
module {
  func.func @main() {
    ...
    %13 = "daphne.matMul"(...) : (!daphne.Matrix<20000x100xf64:...>, !daphne.Matrix<20000x100xf64:...>,
...) -> !daphne.Matrix<20000x20000xf64:...>
    %14 = "daphne.ewLn"(%13) : (!daphne.Matrix<20000x20000xf64:...>) -> !daphne.Matrix<20000x20000xf64>
    %15 = "daphne.ewMul"(%9, %14) : (!daphne.Matrix<20000x20000xf64:sp[1.000000e-04]:rep[sparse]>,
!daphne.Matrix<20000x20000xf64>) -> !daphne.Matrix<20000x20000xf64:sp[1.000000e-04]:rep[sparse]>
    %16 = "daphne.sumAll"(%15) : (!daphne.Matrix<20000x20000xf64:sp[1.000000e-04]:rep[sparse]>) -> f64
    ...
  }
}
```

The four decisive operations `matMul`, `ewLn`, `ewMul`, and `sumAll` are clearly visible (we omit the addition by eps here for brevity). Moreover, the first operand of `ewMul` (X in the DaphneDSL script) is represented in the sparse format Compressed Sparse Row (CSR). The same holds for the result of `ewMul`, which is also the input to `sumAll`. The large dense intermediate results are `%13` and `%14`.

If we further lower these operations to kernel calls, the overall pattern of operations cannot be considered, and the large dense intermediates will still exist. We can verify this situation by printing the IR after lowering to kernel calls. The IR after this pass can be viewed by the following command. The full IR can be found in `scenario3/ir_kernels.txt`.

```
bin/daphne --select-matrix-repr --explain kernels \
  scenario3/cross-entropy-loss.daphne n=20000 k=100 sp=0.0001 seed=12345
```

```
IR after kernel lowering:
module {
  func.func @main() {
    ...
    %18 = "daphne.call_kernel"(...) {callee =
"_matMul__DenseMatrix_double__DenseMatrix_double__DenseMatrix_double__bool__bool"} : (...) -> ...
    ...
    %19 = "daphne.call_kernel"(...) {callee = "_ewLn__DenseMatrix_double__DenseMatrix_double"} : (...)
-> ...
    ...
    %20 = "daphne.call_kernel"(...) {callee =
"_ewMul__CSRMatrix_double__CSRMatrix_double__DenseMatrix_double"} : (...) -> ...
    ...
    %21 = "daphne.call_kernel"(...) {callee = "_sumAll__double__CSRMatrix_double"} : (...) -> f64
    ...
  }
}
```

We can see that the four operations result in individual kernel calls, whereby `ewMul` and `sumAll` were lowered to calls to kernels for sparse (CSR) matrices. The large dense intermediates, now called `%18` and `%19`, still exist.

Alternatively, we can view the IR after sparsity exploiting operator fusion by the following command. The full IR is also in `scenario3/ir_sparsity_exploiting_op_fusion.txt`.

```
bin/daphne --select-matrix-repr --mlir-codegen \
  --explain mlir_codegen_sparsity_exploiting_op_fusion \
  scenario3/cross-entropy-loss.daphne n=20000 k=100 sp=0.0001 seed=12345
```

```
IR after MLIR codegen (sparsity-exploiting operator fusion):
module {
  func.func @main() {
    ...
    %13 = "daphne.convertDenseMatrixToMemRef"(%10) : (!daphne.Matrix<20000x100xf64:sp[1.000000e+00]>) -
> memref<20000x100xf64>
    %14 = "daphne.convertDenseMatrixToMemRef"(%11) : (!daphne.Matrix<20000x100xf64:sp[1.000000e+00]>) -
> memref<20000x100xf64>
    %15 = "daphne.convertCSRMatrixToValuesMemRef"(%9) : (!daphne.Matrix<20000x20000xf64:sp[1.000000e-
04]:rep[sparse]>) -> memref<?xf64>
    %16 = "daphne.convertCSRMatrixToColIdxsMemRef"(%9) : (!daphne.Matrix<20000x20000xf64:sp[1.000000e-
04]:rep[sparse]>) -> memref<?xindex>
    %17 = "daphne.convertCSRMatrixToRowOffsetsMemRef"(%9) :
(!daphne.Matrix<20000x20000xf64:sp[1.000000e-04]:rep[sparse]>) -> memref<20001xindex>
    %18 = affine.parallel (%arg0) = (0) to (20000) reduce ("addf") -> (f64) {
      %25 = affine.load %17[%arg0] : memref<20001xindex>
      %26 = affine.load %17[%arg0 + 1] : memref<20001xindex>
      %27 = scf.for %arg1 = %25 to %26 step %c1 iter_args(%arg2 = %cst) -> (f64) {
        %28 = memref.load %16[%arg1] : memref<?xindex>
        %29 = scf.for %arg3 = %c0 to %c100 step %c1 iter_args(%arg4 = %cst) -> (f64) {
          %33 = memref.load %13[%arg0, %arg3] : memref<20000x100xf64>
          %34 = memref.load %14[%28, %arg3] : memref<20000x100xf64>
          %35 = math.fma %33, %34, %arg4 : f64
          scf.yield %35 : f64
        }
        %30 = math.log %29 : f64
        %31 = memref.load %15[%arg1] : memref<?xf64>
        %32 = math.fma %31, %30, %arg2 : f64
        scf.yield %32 : f64
      }
      affine.yield %27 : f64
    }
    ...
  }
}
```

Now, the entire DAG of operations was replaced by a construct of nested loops that efficiently calculates the overall result. The outermost loop (represented by an `affine.parallel` operation to exploit parallelism in the future) iterates over the rows of the CSR matrix `X`. The outer `scf.for` loop iterates over the non-zero elements in the current row. For each of those, the innermost `scf.for` loop calculates the dot product of the respective vectors in `U` and `t(V)`, whereby fused-multiply-add (`math.fma`) operations are employed for efficiency (`%35`). Then, the logarithm of each such dot product is calculated by a `math.log` operation (`%30`). The logarithms are multiplied by the current element in `X` and added to the global accumulator of what was originally the `sumAll` operation by another fused-multiply-add operation (`%32`). Note that the large dense intermediates are never materialized.

The interoperability of the generated MLIR code with DAPHNE runtime data objects is again achieved through zero-copy conversion operations. We have already seen `convertDenseMatrixToMemRef` in Sections 3.3 and 4.3. The CSR matrix representation is integrated in an analogous way. As CSR matrices consist of three arrays (row offsets, column indexes, values), we introduce three auxiliary conversion operations, namely `convertCSRMatrixToRowOffsetsMemRef`, `convertCSRMatrixToColIdxsMemRef`, and `convertCSRMatrixToValuesMemRef`.

5.3 Experimental Results

After we have seen the effect of sparsity-exploiting operator fusion on the IR, we evaluate its impact on runtime performance of the cross entropy loss calculation. We compare three variants: (1) using dense data representations and kernels only, (2) using sparse representations (CSR) and kernels where beneficial, and (3) employing sparsity-exploiting operator fusion. We randomly generate the input data. We set `X` to a shape of 20k rows by 20k columns with a sparsity ranging from 1.0 (fully dense) to 10^{-7} (1 non-zero in 10 million values). We set `U` and `V` to a shape of 20k rows by 200 columns each.

The experiments were conducted on a server equipped with an AMD EPYC 7443P CPU with a base frequency of 2.85 GHz. This processor has a single socket with 24 physical cores each, resulting in 48 logical cores due to hyper-threading. The L1 data, L1 instruction, L2, and L3 caches have a total size of 768 KiB (32 KiB per core), 768 KiB (32 KiB per core),12 MiB (512 KiB per core), and 128 MiB. The system is further equipped with 256 GiB of DDR4 memory, and during the experiments, all data resides in main memory. The operating system is Ubuntu 24.04.1 LTS GNU/Linux with kernel 6.8.0-48-generic. We compiled DAPHNE with g++ version 13.2.0. We measure only the time required for the cross-entropy loss calculation itself, not the time required for the random number generation. We repeated all measurements 10 times and report only the mean. The experiment ran purely in DAPHNE's local runtime and can be repeated using the scripts `scenario3/experiment.sh` and `scenario3/dia.py`. The original results can be found in `scenario3/res.csv` and `scenario3/dia.pdf`.

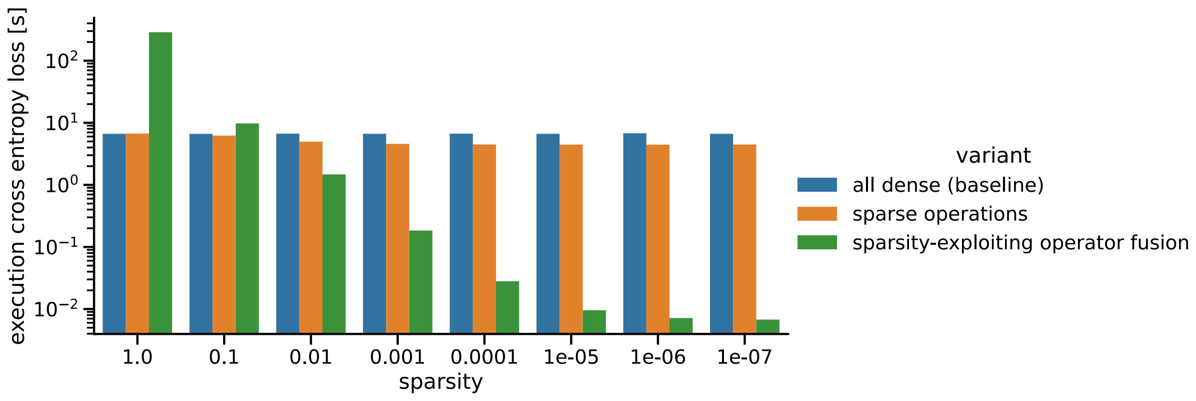

Figure 8: Cross entropy loss execution time for all dense operations, sparse operations, and sparsity-exploiting code generation.

The results are shown in Figure 8. When using only dense data representations and kernels, the runtime performance is constant, i.e., does not depend on the sparsity of X. When DAPHNE selects sparse data representations and kernels where beneficial, but still uses individual kernels, we can observe a significant performance improvement compared to the all-dense approach for sparsities smaller than 0.01 (note the logarithmic vertical axis). However, the greatest improvements are achieved by sparsity-exploiting operator fusion. While this approach is not beneficial for rather dense X (sparsity of 1.0 and 0.1), it clearly outpaces even the approach of using individual sparse kernels. In fact, we can see that sparsity-exploiting operator fusion leads to improved asymptotic behavior of the cross-entropy calculation.

Besides these concrete results for the cross-entropy calculation, this experiment shows that compiler optimizations like sparsity-exploiting operator fusion are possible in the DAPHNE compiler. In the future, we plan to support sparsity-exploiting operator fusion for additional frequently occurring patterns of operations over sparse data, following the ideas in [BRH+18].

6 Other Completed and Ongoing Work in the Compiler

The three demonstration scenarios presented in Sections 3-5 focused on our work regarding extensibility and MLIR-based code generation, as these are our major contributions since the extended compiler prototype [D3.3]. However, besides these contributions, we have worked on various individual aspects of the DAPHNE compiler, some of which have already been merged into the main branch of the DAPHNE prototype and some of which are still ongoing work.

Completed work:

- Improved error handling and more consistent and actionable error messages. This generally increases the ease of use of the DAPHNE prototype as it facilitates the implementation and debugging of complex DaphneDSL scripts. Errors could be caused by invalid user inputs (DaphneDSL scripts or input data) and could happen at all stages of DAPHNE but are presented in a consistent way. Below, we present two examples of error messages that can be provoked by introducing subtle bugs into the shift-and-scale script from Section 4 (these modified scripts can be found in `scenario2/shift-and-scale_error1.daphne` and `scenario2/shift-and-scale_error2.daphne`):

```
[error]: While parsing: DSLVisitor failed with the following message [ variable `Y` referenced
before assignment ]
     | Source file -> "/daphne/./scenario2/shift-and-scale_nohints.daphne":6:9
     |
   6 | Xshift = Y - colMeans;
     |          ^~~
```

```
[error]: Execution error: The kernel-function _ewSqrt__DenseMatrix_double__DenseMatrix_double
failed during runtime with the following message [ invalid argument '-831.744' passed to unary
func SQRT with domain [-0, inf] ]
     | Source file -> "/daphne/./scenario2/shift-and-scale_nohints.daphne":7:9
     |
   7 | colSDs = sqrt(-sum(Xshift * Xshift, 1) / nrow(X));
     |          ^~~
```

- Compiler support for a DaphneDSL `str` (string) value type to be used with matrices and frames. This string value type can be backed by an `std::string` or our own `FixedStr16` type at run-time (we plan to investigate more string representations in the future). This enables DAPHNE to be used in more realistic settings where data sets often contain string columns that are transformed to numeric representations early on.
- Various little improvements and bug fixes, especially related to memory management.

Ongoing work:

- Sparsity estimation for virtually all DaphneIR operations based on naïve meta data estimators. This allows for a more effective use of sparse data representations and sparse kernels throughout complex data analysis pipelines and will serve as a baseline for more advanced sparsity estimators in the future.
- Recompilation of individual blocks of a DaphneIR program at run-time to effectively deal with compile-time unknown data characteristics. This will make DAPHNE more efficient on algorithms that dynamically change the characteristics of the processed data.

- Improvements and extensions to the vectorized pipeline fusion pass by exploiting multiple dimensions for splitting/combining the data as well as aligning data layout decisions with split/combine decisions. This will lead to larger and more efficient vectorized pipelines.
- Automatic loop vectorization that rewrites DaphneDSL loops operating on elements of matrix or frame to the respective coarse-grained matrix/frame operations in DaphneIR; mainly to compensate inefficient DaphneDSL code by users.
- Lowering of DaphneIR frame operations from relational algebra to columnar operations suitable for processing analytical database queries more efficiently.
- Using MLIR's `tensor` dialect with the code generated for DaphneIR operations. This will enable us to benefit from an even larger part of the MLIR framework for low-level optimizations such as lowering to GPU operations.
- Improved inter-procedural analyses, e.g., removal of near-similar specialized user-defined functions to keep the IR simple and reduce the time spent in subsequent compiler passes.

7 Overview of the Final Compiler Prototype Source Code

A general overview of the DAPHNE code base has already been provided in deliverable D3.2 [D3.2]. Here, we focus on the source code relevant to the DAPHNE compiler, which can be found in the following directories of the DAPHNE repository:

- **`src/ir/daphneir/`:** This directory contains the source code of the DaphneIR. Most interestingly, all DaphneIR operations are defined in `DaphneOps.td` in LLVM TableGen notation. Furthermore, specific parts of the type and property inference can be found in the files `DaphneInfer*.td/h/cpp`, such as `DaphneInferShapeOpInterface.cpp`. DaphneIR is the basis for both, the DaphneDSL parser and the DAPHNE compiler.
- **`src/compiler/`:** This directory contains all compiler passes of the DAPHNE compiler (except for the standard MLIR passes we reuse). Most interestingly, `execution/DaphneIrExecutor.cpp` defines the overall DAPHNE compilation chain, `lowering/` contains all passes for lowering and optimizations, `inference/` contains passes related to type and property inference, and `catalog/` contains the kernel extension catalog data structure.
- **`src/parser/catalog/`:** This directory contains the parser for the kernel catalog JSON files and builds up the kernel extension catalog at DAPHNE start-up.
- **`src/parser/daphnedsl/`:** This directory contains the source code of the DaphneDSL parser, which creates the initial, unoptimized DaphneIR representation of the given DaphneDSL script. This initial IR is the starting point for the DAPHNE compiler.
- **`test/api/cli/`:** This directory contains numerous script-level test cases for all kinds of features of DaphneDSL, many of which are designed to trigger and test specific cases in the DAPHNE compiler.
- **`test/codegen/`:** This directory contains unit test cases related to the MLIR-based code generation backend.

References

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

[BRH+18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, Niketan Pansare: On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. Proc. VLDB Endow. 11(12): 1755-1768 (2018)

[D3.1] Matthias Boehm, Ce Zhang, Patrick Damme: DAPHNE D3.1 Language Design Specification, EU Project Deliverable, 11/2021

[D3.2] Matthias Boehm, Ce Zhang, Patrick Damme, DAPHNE Development Team: DAPHNE D3.2 Compiler Prototype, EU Project Deliverable, 02/2022

[D3.3] Patrick Damme, Matthias Boehm, DAPHNE Development Team: DAPHNE D3.3 Extended Compiler Prototype, EU Project Deliverable, 05/2023

[D3.4] Patrick Damme, Philipp Ortner: DAPHNE D3.4 Compiler Design and Overview, EU Project Deliverable, 11/2023

[D4.4] Aristotelis Vontzalidis, Dimitrios Tsoumakos, Stratos Psomadakis, Konstantinos Bitsakos, Vassiliki Kostoula, Jonas H. Müller Korndörfer, Quentin Guilloteau, Florina M. Ciorba: DAPHNE D4.4 Final DSL Runtime Prototype, EU Project Deliverable, 11/2024

[D5.4] Jonas H. Müller Korndörfer, Quentin Guilloteau, Florina M. Ciorba, Matthias Boehm, Patrick Damme: DAPHNE D5.4 Final Design and Prototype of Scheduling Components, EU Project Deliverable, 11/2024

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

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

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

Appendix 1: DaphneDSL Scripts

In the following, we present the complete DaphneDSL scripts used for linear regression model training in the demonstration scenario. These files can also be found in the DAPHNE repository in the directory `scripts/algorithms/`. We omit the license headers to save some space.

File lmDS_.daph

```
# This script has been manually translated from Apache SystemDS.

# The lmDC function solves linear regression using the direct solve method
#
# INPUT:
# -------------------------------------------------------------------------------------
# X        Matrix of feature vectors.
# y        1-column matrix of response values.
# icpt     Intercept presence, shifting and rescaling the columns of X
# reg      Regularization constant (lambda) for L2-regularization. set to nonzero
#          for highly dependant/sparse/numerous features
# verbose  If TRUE print messages are activated
# -------------------------------------------------------------------------------------
#
# OUTPUT:
# --------------------------------------------------------------------------------------------
# B     The model fit
# --------------------------------------------------------------------------------------------

def lmDS(X:matrix<f64>, y:matrix<f64>, icpt:si64, reg:f64,
 verbose:bool) -> matrix<f64> {
  intercept_status = icpt;
  regularization = reg;

  n = nrow (X);
  m = ncol (X);
  ones_n = fill (1.0, n, 1);

  zero_cell = [0.0];

  # Introduce the intercept, shift and rescale the columns of X if needed

  m_ext = m;
  if (intercept_status == 1 || intercept_status == 2)  # add the intercept column
  {
    X = cbind (X, ones_n);
    m_ext = ncol (X);
  }

  scale_lambda = fill (1.0, m_ext, 1);
  if (intercept_status == 1 || intercept_status == 2)
  {
    scale_lambda [m_ext - 1, 0] = [0.0];
  }

  scale_X = [0.0]; # TODO this should not be necessary
  shift_X = [0.0]; # TODO this should not be necessary
  if (intercept_status == 2)  # scale-&-shift X columns to mean 0, variance 1
  {                           # Important assumption: X [, m_ext - 1] = ones_n
    avg_X_cols = t(sum(X, 1)) / n;
    var_X_cols = (t(sum (X ^ 2.0, 1)) - n * (avg_X_cols ^ 2.0)) / (n - 1);
    is_unsafe = (var_X_cols <= 0);
    scale_X = 1.0 / sqrt (var_X_cols * (1 - is_unsafe) + is_unsafe);
    scale_X [m_ext - 1, 0] = [1.0];
    # TODO unary minus
    shift_X = (0 - avg_X_cols) * scale_X;
    shift_X [m_ext - 1, 0] = [0.0];
  } else {
    scale_X = fill (1.0, m_ext, 1);
    shift_X = fill (0.0, m_ext, 1);
  }

  # Henceforth, if intercept_status == 2, we use "X @ (SHIFT/SCALE TRANSFORM)"
  # instead of "X".  However, in order to preserve the sparsity of X,
  # we apply the transform associatively to some other part of the expression
  # in which it occurs.  To avoid materializing a large matrix, we rewrite it:
  #
  # ssX_A  = (SHIFT/SCALE TRANSFORM) @ A    --- is rewritten as:
  # ssX_A  = diagMatrix (scale_X) @ A;
  # ssX_A [m_ext - 1, ] = ssX_A [m_ext - 1, ] + t(shift_X) @ A;
  #
  # tssX_A = t(SHIFT/SCALE TRANSFORM) @ A   --- is rewritten as:
  # tssX_A = diagMatrix (scale_X) @ A + shift_X @ A [m_ext - 1, ];

  lambda = scale_lambda * regularization;
  # BEGIN THE DIRECT SOLVE ALGORITHM (EXTERNAL CALL)
  A = t(X) @ X;
  b = t(X) @ y;
  if (intercept_status == 2) {
    A = t(diagMatrix (scale_X) @ A + shift_X @ A [m_ext - 1, ]);
    A =   diagMatrix (scale_X) @ A + shift_X @ A [m_ext - 1, ];
    b =   diagMatrix (scale_X) @ b + shift_X @ b [m_ext - 1, ];
  }
  A = A + diagMatrix (lambda);
  if (verbose)
  print ("Calling the Direct Solver...");

  beta_unscaled = solve (A, b);

  # END THE DIRECT SOLVE ALGORITHM
  beta = [0.0]; # TODO this should not be necessary
  if (intercept_status == 2) {
      beta = scale_X * beta_unscaled;
      beta [m_ext - 1, ] = beta [m_ext - 1, ] + t(shift_X) @ beta_unscaled;
  } else {
      beta = beta_unscaled;
  }

  if (verbose) {
    print ("Computing the statistics...");
    avg_tot = sum (y) / n;
    ss_tot = sum (y ^ 2);
    ss_avg_tot = ss_tot - n * avg_tot ^ 2;
    var_tot = ss_avg_tot / (n - 1);
    y_residual = y - X @ beta;
    avg_res = sum (y_residual) / n;
    ss_res = sum (y_residual ^ 2);
    ss_avg_res = ss_res - n * avg_res ^ 2;

    R2 = 1 - ss_res / ss_avg_tot;
    dispersion = (n > m_ext) ? (ss_res / (n - m_ext)) : nan;
    adjusted_R2 = (n > m_ext) ? (1 - dispersion / (ss_avg_tot / (n - 1))) : nan;

    R2_nobias = 1 - ss_avg_res / ss_avg_tot;
    deg_freedom = n - m - 1;
    var_res = 0.0; # TODO this should not be necessary
    adjusted_R2_nobias = 0.0; # TODO this should not be necessary
    if (deg_freedom > 0) {
      var_res = ss_avg_res / deg_freedom;
      adjusted_R2_nobias = 1 - var_res / (ss_avg_tot / (n - 1));
    } else {
      var_res = nan;
      adjusted_R2_nobias = nan;
      print ("Warning: zero or negative number of degrees of freedom.");
    }

    R2_vs_0 = 1 - ss_res / ss_tot;
    adjusted_R2_vs_0 = (n > m) ? (1 - (ss_res / (n - m)) / (ss_tot / n)) : nan;

    print ("AVG_TOT_Y, " + avg_tot +                 # Average of the response value Y
      "\nSTDEV_TOT_Y, " + sqrt (var_tot) +           # Standard Deviation of the response value Y
      "\nAVG_RES_Y, " + avg_res +                    # Average of the residual Y - pred(Y|X), i.e. residual bias
      "\nSTDEV_RES_Y, " + sqrt (var_res) +           # Standard Deviation of the residual Y - pred(Y|X)
      "\nDISPERSION, " + dispersion +                # GLM-style dispersion, i.e. residual sum of squares / # d.f.
      "\nR2, " + R2 +                                # R^2 of residual with bias included vs. total average
      "\nADJUSTED_R2, " + adjusted_R2 +              # Adjusted R^2 of residual with bias included vs. total average
      "\nR2_NOBIAS, " + R2_nobias +                  # R^2 of residual with bias subtracted vs. total average<Paste>
      "\nADJUSTED_R2_NOBIAS, " + adjusted_R2_nobias);  # Adjusted R^2 of residual with bias subtracted vs. total average
    if (intercept_status == 0) {
      print ("R2_VS_0, " + R2_vs_0 +               #  R^2 of residual with bias included vs. zero constant
        "\nADJUSTED_R2_VS_0, " + adjusted_R2_vs_0);  #  Adjusted R^2 of residual with bias included vs. zero constant
    }
  }

  B = beta;
  return B;
}
```

File lmDS.daph

```
import "lmDS_.daph";

# Command-line arguments:
# XY      ... file name of the input file
# icpt    ... intercept, must be in [0, 1, 2]
# reg     ... regularization, recommended: 0.0000001
# verbose ... whether to print verbose output, must be in [false, true]

XY = readMatrix($XY);
X = XY[, :(ncol(XY) - 1)];
y = XY[, ncol(XY) - 1];

b = lmDS_.lmDS(X, y, $icpt, $reg, $verbose);

print("");
print("RESULT");
print(b);
```

File lmCG_.daph

```
# This script has been manually translated from Apache SystemDS.

# The lmCG function solves linear regression using the conjugate gradient algorithm
#
# INPUT:
# ---------------------------------------------------------------------------------------
# X        Matrix of feature vectors.
# y        1-column matrix of response values.
# icpt     Intercept presence, shifting and rescaling the columns of X
# reg      Regularization constant (lambda) for L2-regularization. set to nonzero
#          for highly dependant/sparse/numerous features
# tol      Tolerance (epsilon); conjugate gradient procedure terminates early if L2
#          norm of the beta-residual is less than tolerance * its initial norm
# maxi     Maximum number of conjugate gradient iterations. 0 = no maximum
# verbose  If TRUE print messages are activated
# ---------------------------------------------------------------------------------------
#
# OUTPUT:
# ----------------------------------------------------------------------------------------------
# B     The model fit
# ----------------------------------------------------------------------------------------------

def lmCG(X:matrix<f64>, y:matrix<f64>, icpt:si64, reg:f64, tol:f64,
 maxi:si64, verbose:bool) -> matrix<f64> {
  intercept_status = icpt;
  regularization = reg;
  tolerance = tol;
  max_iteration = maxi;

  n = nrow (X);
  m = ncol (X);
  ones_n = fill (1.0, n, 1);
  zero_cell = [0.0];

  # Introduce the intercept, shift and rescale the columns of X if needed

  m_ext = m;
  if (intercept_status == 1 || intercept_status == 2)  # add the intercept column
  {
    X = cbind (X, ones_n);
    m_ext = ncol (X);
  }

  scale_lambda = fill (1.0, m_ext, 1);
  if (intercept_status == 1 || intercept_status == 2)
  {
      scale_lambda [m_ext - 1, 0] = [0.0];
  }

  scale_X = [0.0]; # TODO this should not be necessary
  shift_X = [0.0]; # TODO this should not be necessary
  if (intercept_status == 2)  # scale-&-shift X columns to mean 0, variance 1
  {                           # Important assumption: X [, m_ext - 1] = ones_n
    avg_X_cols = t(sum(X, 1)) / n;
    var_X_cols = (t(sum (X ^ 2.0, 1)) - n * (avg_X_cols ^ 2.0)) / (n - 1);
    is_unsafe = (var_X_cols <= 0.0);
    scale_X = 1.0 / sqrt (var_X_cols * (1.0 - is_unsafe) + is_unsafe);
    scale_X [m_ext - 1, 0] = [1.0];
    shift_X = (0 - avg_X_cols) * scale_X;
    shift_X [m_ext - 1, 0] = [0.0];
  } else {
    scale_X = fill (1.0, m_ext, 1);
    shift_X = fill (0.0, m_ext, 1);
  }

  # Henceforth, if intercept_status == 2, we use "X @ (SHIFT/SCALE TRANSFORM)"
  # instead of "X".  However, in order to preserve the sparsity of X,
  # we apply the transform associatively to some other part of the expression
  # in which it occurs.  To avoid materializing a large matrix, we rewrite it:
  #
  # ssX_A  = (SHIFT/SCALE TRANSFORM) @ A    --- is rewritten as:
  # ssX_A  = diagMatrix (scale_X) @ A;
  # ssX_A [m_ext - 1, ] = ssX_A [m_ext - 1, ] + t(shift_X) @ A;
  #
  # tssX_A = t(SHIFT/SCALE TRANSFORM) @ A   --- is rewritten as:
  # tssX_A = diag (scale_X) @ A + shift_X @ A [m_ext - 1, ];

  lambda = scale_lambda * regularization;
  beta_unscaled = fill (0.0, m_ext, 1);

  if (max_iteration == 0) {
    max_iteration = as.si64(m_ext);
  }
  i = 0;

  # BEGIN THE CONJUGATE GRADIENT ALGORITHM
  if (verbose) print ("Running the CG algorithm...");

  r = (0.0 - t(X)) @ y;

  if (intercept_status == 2) {
    r = scale_X * r + shift_X @ r [m_ext - 1, ];
  }

  p = 0.0 - r;
  norm_r2 = sum (r ^ 2.0);
  norm_r2_initial = norm_r2;
  norm_r2_target = norm_r2_initial * tolerance ^ 2.0;
  if (verbose) print ("||r|| initial value = " + sqrt (norm_r2_initial) + ",  target value = " + sqrt (norm_r2_target));

  while (i < max_iteration && norm_r2 > norm_r2_target)
  {
    ssX_p = [0.0]; # TODO this should not be necessary
    if (intercept_status == 2) {
      ssX_p = scale_X * p;
      ssX_p [m_ext - 1, ] = ssX_p [m_ext - 1, ] + t(shift_X) @ p;
    } else {
      ssX_p = p;
    }

    q = t(X) @ (X @ ssX_p);

    if (intercept_status == 2) {
      q = scale_X * q + shift_X @ q [m_ext - 1, ];
    }

    q = q + lambda * p;
    a = norm_r2 / sum (p * q);
    beta_unscaled = beta_unscaled + a * p;
    r = r + a * q;
    old_norm_r2 = norm_r2;
    norm_r2 = sum (r ^ 2);
    p = (0.0 - r) + (norm_r2 / old_norm_r2) * p;
    i = i + 1;
    if (verbose) print ("Iteration " + i + ":  ||r|| / ||r init|| = " + sqrt (norm_r2 / norm_r2_initial));
  }

  if (i >= max_iteration) {
    if (verbose) print ("Warning: the maximum number of iterations has been reached.");
  }

  # END THE CONJUGATE GRADIENT ALGORITHM
  beta = [0.0]; # TODO this should not be necessary
  if (intercept_status == 2) {
    beta = scale_X * beta_unscaled;
```

```
    beta [m_ext - 1, ] = beta [m_ext - 1, ] + t(shift_X) @ beta_unscaled;
  } else {
    beta = beta_unscaled;
  }

  if (verbose) {
    print ("Computing the statistics...");

    avg_tot = sum (y) / n;
    ss_tot = sum (y ^ 2);
    ss_avg_tot = ss_tot - n * avg_tot ^ 2;
    var_tot = ss_avg_tot / (n - 1);
    y_residual = y - X @ beta;
    avg_res = sum (y_residual) / n;
    ss_res = sum (y_residual ^ 2);
    ss_avg_res = ss_res - n * avg_res ^ 2;

    R2 = 1 - ss_res / ss_avg_tot;
    dispersion = (n > m_ext) ? (ss_res / (n - m_ext)) : nan;
    adjusted_R2 = (n > m_ext) ? (1 - dispersion / (ss_avg_tot / (n - 1))) : nan;

    R2_nobias = 1 - ss_avg_res / ss_avg_tot;
    deg_freedom = n - m - 1;
      var_res = 0.0; # TODO this should not be necessary
      adjusted_R2_nobias = 0.0; # TODO this should not be necessary
    if (deg_freedom > 0) {
      var_res = ss_avg_res / deg_freedom;
      adjusted_R2_nobias = 1 - var_res / (ss_avg_tot / (n - 1));
    } else {
      var_res = nan;
      adjusted_R2_nobias = nan;
      print ("Warning: zero or negative number of degrees of freedom.");
    }

    R2_vs_0 = 1 - ss_res / ss_tot;

    adjusted_R2_vs_0 = (n > m) ? (1 - (ss_res / (n - m)) / (ss_tot / n)) : nan;

    print ("AVG_TOT_Y, " + avg_tot +                 # Average of the response value Y
      "\nSTDEV_TOT_Y, " + sqrt (var_tot) +           # Standard Deviation of the response value Y
      "\nAVG_RES_Y, " + avg_res +                    # Average of the residual Y - pred(Y|X), i.e. residual bias
      "\nSTDEV_RES_Y, " + sqrt (var_res) +           # Standard Deviation of the residual Y - pred(Y|X)
      "\nDISPERSION, " + dispersion +                # GLM-style dispersion, i.e. residual sum of squares / # d.f.
      "\nR2, " + R2 +                                # R^2 of residual with bias included vs. total average
      "\nADJUSTED_R2, " + adjusted_R2 +              # Adjusted R^2 of residual with bias included vs. total average
      "\nR2_NOBIAS, " + R2_nobias +                  # R^2 of residual with bias subtracted vs. total average<Paste>
      "\nADJUSTED_R2_NOBIAS, " + adjusted_R2_nobias);  # Adjusted R^2 of residual with bias subtracted vs. total average
    if (intercept_status == 0) {
      print ("R2_VS_0, " + R2_vs_0 +               #  R^2 of residual with bias included vs. zero constant
        "\nADJUSTED_R2_VS_0, " + adjusted_R2_vs_0);  #  Adjusted R^2 of residual with bias included vs. zero constant
    }
  }

  B = beta;
  return B;
}
```

File lmCG.daph

```
import "lmCG_.daph";
# Command-line arguments:
# XY      ... file name of the input file
# icpt    ... intercept, must be in [0, 1, 2]
# reg     ... regularization, recommended: 0.0000001
# tol     ... tolerance, recommended: 0.0000001
# maxi    ... maximim number of iterations, recommended: 0 (no maximum)
# verbose ... whether to print verbose output, must be in [false, true]

XY = readMatrix($XY);
X = XY[, :(ncol(XY) - 1)];
y = XY[, ncol(XY) - 1];

b = lmCG_.lmCG(X, y, $icpt, $reg, $tol, $maxi, $verbose);

print("");
print("RESULT");
print(b);
```

Appendix 2: Example of a Complete DaphneIR

Here, we provide the complete DaphneIR after type/property inference in Section 3.3 as an example.

```
IR after inference:
module {
  func.func @"lmDS-1-1"(%arg0: !daphne.Matrix<4898x11xf64>, %arg1: !daphne.Matrix<4898x1xf64>, %arg2: si64, %arg3: f64, %arg4: i1) -> !daphne.Matrix<12x1xf64> {
    %0 = "daphne.constant"() {value = false} : () -> i1
    %1 = "daphne.constant"() {value = 11 : index} : () -> index
    %2 = "daphne.constant"() {value = 12 : index} : () -> index
    %3 = "daphne.constant"() {value = 4898 : index} : () -> index
    %4 = "daphne.constant"() {value = 9.9999999999999995E-8 : f64} : () -> f64
    %5 = "daphne.constant"() {value = 0 : index} : () -> index
    %6 = "daphne.constant"() {value = 1 : index} : () -> index
    %7 = "daphne.constant"() {value = 93916474695824 : ui64} : () -> ui64
    %8 = "daphne.constant"() {value = 93916474637328 : ui64} : () -> ui64
    %9 = "daphne.constant"() {value = 93916474631232 : ui64} : () -> ui64
    %10 = "daphne.constant"() {value = 93916474611760 : ui64} : () -> ui64
    %11 = "daphne.constant"() {value = 93916474428544 : ui64} : () -> ui64
    %12 = "daphne.constant"() {value = 0.000000e+00 : f64} : () -> f64
    %13 = "daphne.constant"() {value = 1.000000e+00 : f64} : () -> f64
    %14 = "daphne.fill"(%13, %3, %6) : (f64, index, index) -> !daphne.Matrix<4898x1xf64>
    %15 = "daphne.matrixConstant"(%11) : (ui64) -> !daphne.Matrix<1x1xf64>
    %16 = "daphne.colBind"(%arg0, %14) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) -> !daphne.Matrix<4898x12xf64>
    %17 = "daphne.fill"(%13, %2, %6) : (f64, index, index) -> !daphne.Matrix<12x1xf64>
    %18 = "daphne.matrixConstant"(%10) : (ui64) -> !daphne.Matrix<1x1xf64>
    %19 = "daphne.sliceRow"(%17, %1, %2) : (!daphne.Matrix<12x1xf64>, index, index) -> !daphne.Matrix<1x1xf64>
    %20 = "daphne.insertCol"(%19, %18, %5, %6) : (!daphne.Matrix<1x1xf64>, !daphne.Matrix<1x1xf64>, index, index) -> !daphne.Matrix<1x1xf64>
    %21 = "daphne.insertRow"(%17, %20, %1, %2) : (!daphne.Matrix<12x1xf64>, !daphne.Matrix<1x1xf64>, index, index) -> !daphne.Matrix<12x1xf64>
    %22 = "daphne.matrixConstant"(%9) : (ui64) -> !daphne.Matrix<1x1xf64>
    %23 = "daphne.matrixConstant"(%8) : (ui64) -> !daphne.Matrix<1x1xf64>
    %24 = "daphne.fill"(%13, %2, %6) : (f64, index, index) -> !daphne.Matrix<12x1xf64>
    %25 = "daphne.fill"(%12, %2, %6) : (f64, index, index) -> !daphne.Matrix<12x1xf64>
    %26 = "daphne.ewMul"(%21, %4) : (!daphne.Matrix<12x1xf64>, f64) -> !daphne.Matrix<12x1xf64>
    %27 = "daphne.transpose"(%16) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<12x4898xf64>
    %28 = "daphne.matMul"(%27, %16, %0, %0) : (!daphne.Matrix<12x4898xf64>, !daphne.Matrix<4898x12xf64>, i1, i1) -> !daphne.Matrix<12x12xf64>
    %29 = "daphne.matMul"(%27, %arg1, %0, %0) : (!daphne.Matrix<12x4898xf64>, !daphne.Matrix<4898x1xf64>, i1, i1) -> !daphne.Matrix<12x1xf64>
    %30 = "daphne.diagMatrix"(%26) : (!daphne.Matrix<12x1xf64>) -> !daphne.Matrix<12x12xf64:sp[0.083333333333333329]>
    %31 = "daphne.ewAdd"(%28, %30) : (!daphne.Matrix<12x12xf64>, !daphne.Matrix<12x12xf64:sp[0.083333333333333329]>) -> !daphne.Matrix<12x12xf64>
    %32 = "daphne.solve"(%31, %29) : (!daphne.Matrix<12x12xf64>, !daphne.Matrix<12x1xf64>) -> !daphne.Matrix<12x1xf64>
    %33 = "daphne.matrixConstant"(%7) : (ui64) -> !daphne.Matrix<1x1xf64>
    "daphne.return"(%32) : (!daphne.Matrix<12x1xf64>) -> ()
  }
  func.func @main() {
    %0 = "daphne.constant"() {value = 11 : index} : () -> index
    %1 = "daphne.constant"() {value = 12 : index} : () -> index
    %2 = "daphne.constant"() {value = 0 : index} : () -> index
    %3 = "daphne.constant"() {value = "RESULT"} : () -> !daphne.String
    %4 = "daphne.constant"() {value = true} : () -> i1
    %5 = "daphne.constant"() {value = ""} : () -> !daphne.String
    %6 = "daphne.constant"() {value = false} : () -> i1
    %7 = "daphne.constant"() {value = 9.9999999999999995E-8 : f64} : () -> f64
    %8 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %9 = "daphne.constant"() {value = "data/wine.csv"} : () -> !daphne.String
    %10 = "daphne.read"(%9) : (!daphne.String) -> !daphne.Matrix<4898x12xf64:sp[1.000000e+00]>
    %11 = "daphne.sliceCol"(%10, %2, %0) : (!daphne.Matrix<4898x12xf64:sp[1.000000e+00]>, index, index) -> !daphne.Matrix<4898x11xf64>
    %12 = "daphne.sliceCol"(%10, %0, %1) : (!daphne.Matrix<4898x12xf64:sp[1.000000e+00]>, index, index) -> !daphne.Matrix<4898x1xf64>
    %13 = "daphne.generic_call"(%11, %12, %8, %7, %6) {callee = "lmDS-1-1"} : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>, si64, f64, i1) ->
!daphne.Matrix<12x1xf64>
    "daphne.print"(%5, %4, %6) : (!daphne.String, i1, i1) -> ()
    "daphne.print"(%3, %4, %6) : (!daphne.String, i1, i1) -> ()
    "daphne.print"(%13, %4, %6) : (!daphne.Matrix<12x1xf64>, i1, i1) -> ()
    "daphne.return"() : () -> ()
  }
}
```