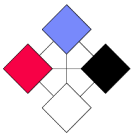
D3.3 Extended Compiler Prototype



DAPHNE

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

Version 1.1

PUBLIC

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

Document Description

Previous deliverables already shared the overall and refined system architecture [D2.1, D2.2] as well as the initial language and compiler designs [D3.1, D+22] and the initial compiler prototype [D3.2]. This document presents the extended DAPHNE compiler prototype, which is still based on MLIR (multi-level intermediate representation) [LA+21] as a library of compiler infrastructure to facilitate a cost-effective development of our domain-specific language, reuse of compiler infrastructure, and good extensibility. This document shares a snapshot of this prototype, describes an example scenario of running linear regression model training on a real dataset, which walks the reader through some of the most decisive steps of DAPHNE's optimizing compilation pipeline, and presents the results of some micro benchmarks. The DAPHNE prototype, including the DAPHNE compiler prototype, has been available as open source on GitHub since March 2022.

D3.3 Extended Compiler Prototype			
WP3 – DSL Abstractions and Compilation			
Type of document	D	Version	1.1
Dissemination level	PU	M30 (May 2023)	
Lead partner	TUB		
Author(s)	Patrick Damme (TUB), Matthias Boehm (TUB), DAPHNE Development Team		
Reviewer(s)	Philippe Bonnet (ITU), Mark Dokter (KNOW)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial structure, write-up, and experiments	Patrick Damme (TUB)
V1.1	Incorporated feedback by Philippe Bonnet (ITU), Mark Dokter (KNOW), and Matthias Boehm (TUB); additional experiments; various minor improvements	Patrick Damme (TUB)

1 Artifact Access

The extended compiler prototype is publicly accessible as a password-protected snapshot of the DAPHNE development repository under the following link:

- **Link:** https://daphne-eu.know-center.at/index.php/s/zzRjX4kJkwXkPoM (~7 MB)
- **Password:** GZNoXliPEruY

This snapshot is a copy of the DAPHNE open-source repository at https://github.com/daphne-eu/daphne (commit 63fdf44d24c5ceb72b8800494e02587b72612ae1, May 29, 2023).

Furthermore, the artifact contains a directory `daphne/_d3.3/` with some files specific to this deliverable, such as scripts for executing DAPHNE in a Docker container, the DaphneIR outputs produced by the demonstration scenario in Section 3, and the scripts for the micro benchmarks in Section 4, all of which are referenced later in this document.

2 Environment Setup

After you have downloaded the file `daphne-d3.3-v1.1.zip` from the link above, please open a terminal in the same directory and execute the following commands to set up the environment for going along with the demonstration scenario in the next section. You can copy each command separately into the terminal. Commands that span multiple lines use `\` at the line endings. Alternatively, the DAPHNE documentation in `daphne/doc/GettingStarted.md` 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 output of the commands in the directory `daphne/_d3.3/ir/`, such that it can also be understood without running the system.

```
# Unpack the deliverable artifact and cd into it.
unzip daphne-d3.3-v1.1.zip
cd daphne-d3.3-v1.1

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

# Run bash in the container for an interactive environment.
./_d3.3/run-docker.sh

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

# 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 Demonstration Scenario

3.1 Running Example: Linear Regression Model Training

To demonstrate some of the most important (but not all) features of the extended DAPHNE compiler prototype, 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 in deliverable D3.2 [D3.2] on the initial compiler prototype. Meanwhile, we have significantly extended the DAPHNE compiler support for DaphneDSL scripts like this, as will become clear in the following. Furthermore, we support two methods for the task of linear regression model training: the direct solve (DS) method and the conjugate gradient (CG) method (deliverable D3.2 [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 4.

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
```

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

To illustrate what the DAPHNE compiler does to make this script work, we have a look at selected steps of the compilation chain. To this end, we utilize DAPHNE's explanation feature, which prints the DaphneIR at chosen stages.

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. 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 `daphne/_d3.3/ir/ir_01_parsing.txt`.

Initial simplification. The DAPHNE compiler chain starts by applying a first round of straightforward simplifications, including 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 `daphne/_d3.3/ir/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"(%67) : (!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?xf64>
    ...
    %12 = "daphne.sliceCol"(%8, ...) : (!daphne.Matrix<?x?xf64>, ...) -> !daphne.Matrix<?x?xf64>
    ...
    %17 = "daphne.sliceCol"(%8, ...) : (!daphne.Matrix<?x?xf64>, ...) -> !daphne.Matrix<?x?xf64>
    %18 = "daphne.generic_call"(%12, %17, ...) {callee = "lmDS-1"} : (!daphne.Matrix<?x?xf64>,
!daphne.Matrix<?x?xf64>, ...) -> !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 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?xf64>`.

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, we will come back to shortly. The complete IR still has a length of ~310 lines and can be found in `daphne/_d3.3/ir/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>) {
      %72 = "daphne.colBind"(%arg0, %39) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
      %73 = "daphne.cast"(%72) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<4898x?xf64>
      scf.yield %73 : !daphne.Matrix<4898x?xf64>
    } else {
      %72 = "daphne.cast"(%arg0) : (!daphne.Matrix<4898x11xf64>) -> !daphne.Matrix<4898x?xf64>
      scf.yield %72 : !daphne.Matrix<4898x?xf64>
    }
    ...
    "daphne.return"(%71) : (!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, %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, ...) {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 shapes inside the main function have now become known by propagating the known 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 `lsDS-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 `daphne/_d3.3/ir/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"(%32) : (!daphne.Matrix<12x1xf64>) -> ()
  }
  func.func @main() {
    ...
    %10 = "daphne.read"(%9) : (...) -> !daphne.Matrix<4898x12xf64:...>
    %11 = "daphne.sliceCol"(%10, %2, %0) : (!daphne.Matrix<4898x12xf64:...>, index, index) ->
!daphne.Matrix<4898x11xf64>
    %12 = "daphne.sliceCol"(%10, %0, %1) : (!daphne.Matrix<4898x12xf64:...>, index, index) ->
!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 more 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 `daphne/_d3.3/ir/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 `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:
...
%16 = "daphne.colBind"(%arg0, %14) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
...
%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>
```

```
IR after selecting physical operators:
...
%15 = "daphne.colBind"(%arg0, %13) : (!daphne.Matrix<4898x11xf64>, !daphne.Matrix<4898x1xf64>) ->
!daphne.Matrix<4898x12xf64>
...
%26 = "daphne.syrk"(%15) : (!daphne.Matrix<4898x12xf64>) -> !daphne.Matrix<12x12xf64>
%27 = "daphne.gemv"(%15, %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.

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 `daphne/_d3.3/ir/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"(...) -> ... {
  ...
  %25:2 = "daphne.vectorizedPipeline"(%arg0, %13, %arg1, ...) ({
  ^bb0(%arg5: !daphne.Matrix<?x11xf64>, %arg6: !daphne.Matrix<?x1xf64>, %arg7:
!daphne.Matrix<?x1xf64>):
    %30 = "daphne.colBind"(%arg5, %arg6) : (!daphne.Matrix<?x11xf64>, !daphne.Matrix<?x1xf64>) ->
!daphne.Matrix<?x?xf64>
    %31 = "daphne.gemv"(%30, %arg7) : (!daphne.Matrix<?x?xf64>, !daphne.Matrix<?x1xf64>) ->
!daphne.Matrix<?x?xf64>
    %32 = "daphne.syrk"(%30) : (!daphne.Matrix<?x?xf64>) -> !daphne.Matrix<?x?xf64>
    "daphne.return"(%31, %32) : (!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, and the general matrix-vector multiplication 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 resembles 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 `daphne/_d3.3/ir/ir_07_vectorized_alternative.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 are ultimately freed. DAPHNE's memory management concerns two levels: One 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 not needed anymore 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. 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 `daphne/_d3.3/ir/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() {
  ...
  %12 = "daphne.read"(...) : (...) -> ...
  %13 = "daphne.sliceCol"(%12, ...) : (...) -> ...
  %14 = "daphne.sliceCol"(%12, ...) : (...) -> ...
  "daphne.decRef"(%12) : (...>) -> ()
  "daphne.incRef"(%13) : (...) -> ()
  "daphne.incRef"(%14) : (...) -> ()
  %15 = "daphne.generic_call"(%13, %14, ...) {callee = "lmDS-1-1"} : (...) -> ...
  "daphne.decRef"(%14) : (...) -> ()
  "daphne.decRef"(%13) : (...) -> ()
  ...
  "daphne.print"(%15, ...) : (...) -> ()
  "daphne.decRef"(%15) : (...) -> ()
  ...
}
```

Lowering DaphneIR operations to kernel calls. 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 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 `daphne/_d3.3/ir/ir_09_kernels.txt`. As an example, the `syrk` operation has been lowered to:

```
IR after kernel lowering:
...
%28 = "daphne.call_kernel"(%17, %14) {callee = "_syrk__DenseMatrix_double__DenseMatrix_double"} :
(!daphne.Matrix<4898x12xf64>, !daphne.DaphneContext) -> !daphne.Matrix<12x12xf64>
```

Here, `_syrk__DenseMatrix_double__DenseMatrix_double` is a specific kernel function that internally calls a BLAS routine. The lowering to kernel calls is one of the most decisive connections to the DAPHNE runtime and WP4.

Lowering DaphneIR operations by low-level code generation. As an alternative to lowering to C++ kernels, we are currently exploring the on-the-fly generation of low-level MLIR code for DaphneIR domain-specific operations, by relying on MLIR dialects such as linalg, affine, arith, and memref. Details on code generation will be provided in deliverable D3.4, which follows in six months.

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 `daphne/_d3.3/ir/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 functions 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 pipelines 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 {...} {
    ...
    %118 = llvm.mlir.addressof @_vect2 : ...
    ...
    %168 = llvm.bitcast %118 : ...> to ...
    llvm.store %168, %166 : ...
    ...
    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`.

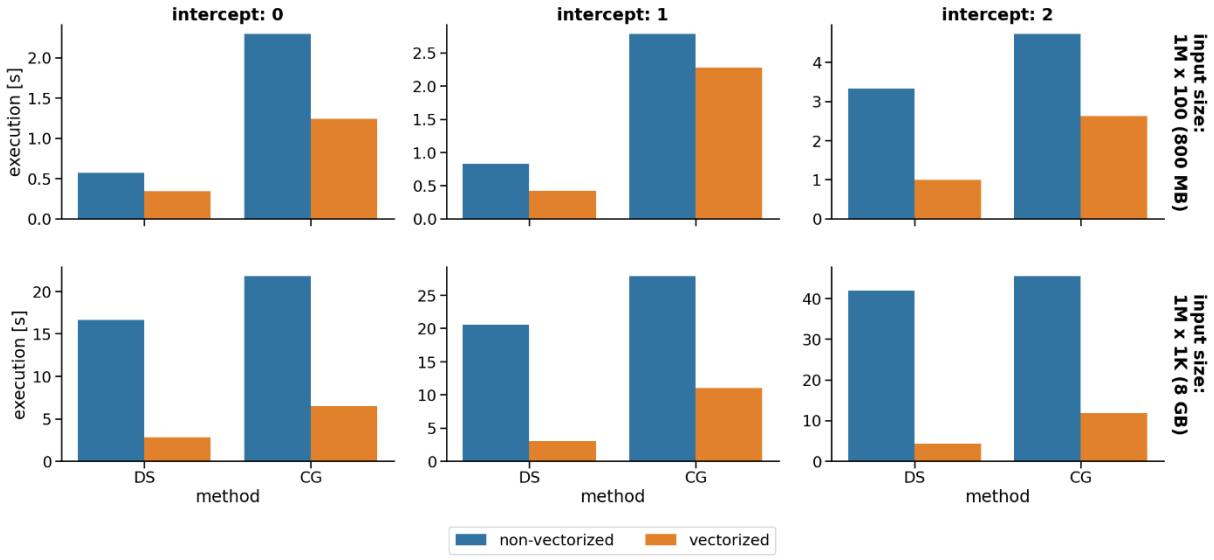
4 Micro Benchmarks

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 randomly generated double-precision input data sets with 1 million rows and either 100 (800 MB) or 1000 (8 GB) columns as well as all three options for the intercept.

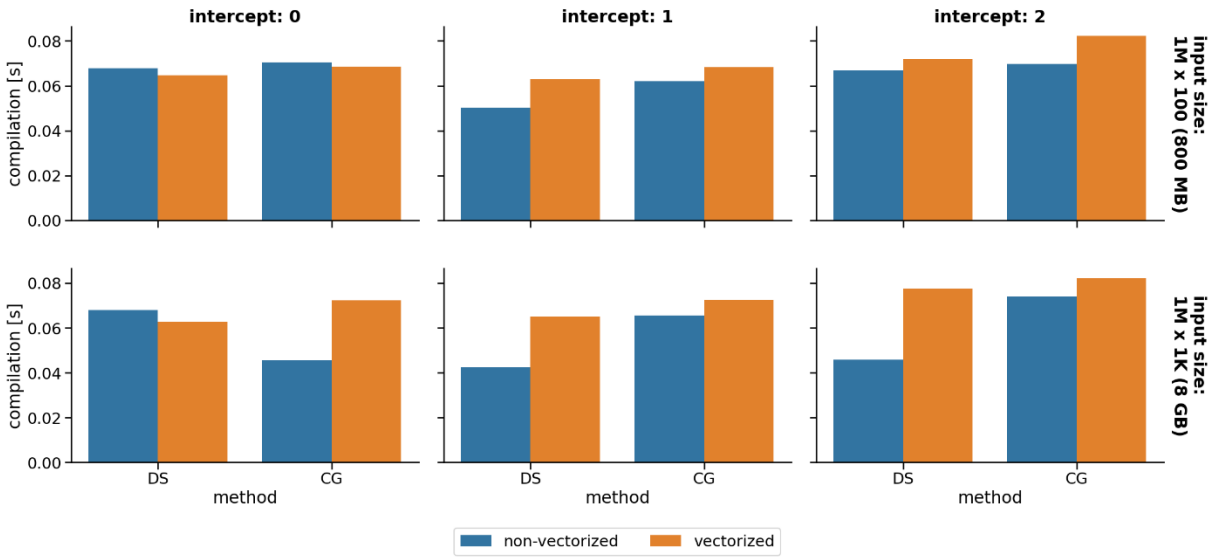
The experiments were conducted on a server equipped with an Intel Xeon Gold 6338 CPU clocked at 2 GHz. This processor has two sockets with 32 physical cores each, resulting in 128 logical cores due to hyper-threading. The L1 data, L1 instruction, L2, and L3 caches have a total size of 3 MiB (32 KiB per 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 20.04.5 LTS GNU/Linux with kernel 5.4.0-144-generic. We compiled DAPHNE with g++ version 9.4. We repeated all time measurements 10 times and report the means of all repetitions.

The experiments presented in the following can be reproduced using the scripts `_d3.3/exp.sh` (within the container) and `_d3.3/dia.py`. The original results can be found in `_d3.3/res.csv`.

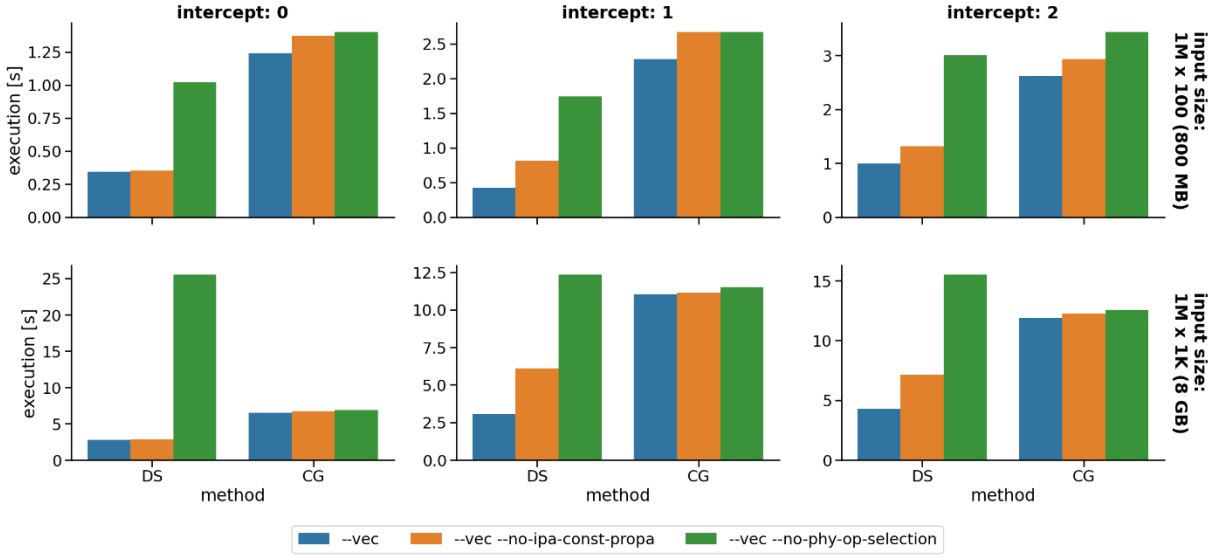
Non-vectorized vs. vectorized execution. In the first experiment, we compare the execution time of the non-vectorized execution to the vectorized execution. We report only the execution times of the `lmDS` and `lmCG` functions here, i.e., the time for DaphneDSL parsing (negligible), compilation, and the random data generation are not included. The results are shown in the figure below. 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.



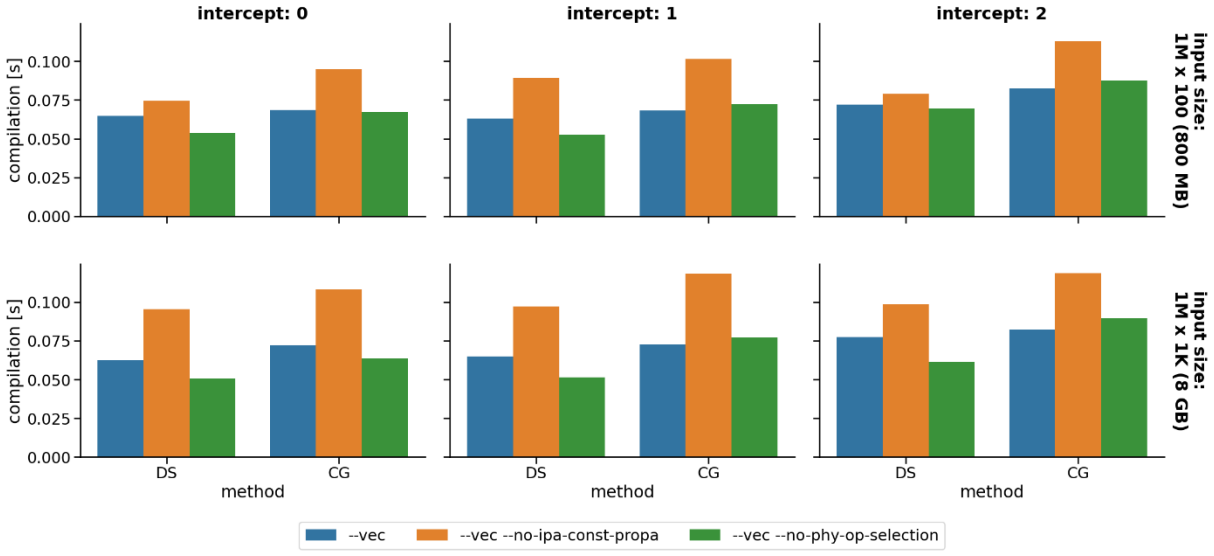
The next figure shows the compilation times, including all optimizations and lowering performed by the DAPHNE compiler as well as the LLVM just-in-time compilation. The compilation times are far lower than the execution times. Even more importantly, the compilation time does not depend on the input data 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.



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 the figure below. 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.



Next, we also show the impact of these compiler flags on the compilation time. The results are shown in the figure below. 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.



5 Overview of the Extended 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, and `inference/` contains passes related to type and property inference.
- **`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.

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

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

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

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

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

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

[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 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"() : () -> ()
  }
}
```