AMLS 2022 Practical Exercises in the DAPHNE EU-Project

"The DAPHNE project aims to define and build an open and extensible system infrastructure for integrated data analysis pipelines, including data management and processing, high-performance computing (HPC), and machine learning (ML) training and scoring." (more information on https://daphne-eu.eu/)

In a consortium of 13 European partner institutions from academia and industry, we are developing the DAPHNE system with all its components including the DAPHNE compiler, and the DAPHNE runtime.

The initial open-source release of the code is planned for end of March 2022.

In summer 2022, we offer the following programming projects for interested AMLS students:

#197: Conversion Tool DML to DaphneDSL

DaphneDSL, DAPHNE's domain specific language, offers all necessary means for writing complex linear/relational algebra programs. However, implementing typical ML algorithms in such a DSL is a complex and time-intensive task. At the same time, the usefulness of the DAPHNE system depends on a rich library of algorithms. Interestingly, Apache SystemDS already has such a library, written in DML. Thus, our goal is to translate these DML scripts to DaphneDSL automatically.

The task is to implement a stand-alone tool (in Python, C/C++, or Java) which converts a given DML script into an equivalent DaphneDSL script. A documentation of both languages as well as ANTLR grammar definitions are already given.

#7: Sideways Entry into Compilation

Typically, a DAPHNE user would implement an integrated data analysis pipeline using the domain-specific language DaphneDSL. The DAPHNE system parses a DaphneDSL script into an internal representation, the so-called DaphneIR, which is based on MLIR/LLVM. To give researchers and practitioners a chance to fine-tune the internal representation manually (e.g., for what-if analyses on not-yet-existing compiler optimizations), we want to support reading DaphneIR files as an alternative. Then we could print the IR at any level (see #6), modify it externally (e.g., by hand to try out variants the compiler would not come up with), and load it again to continue the processing from that point on, which could be very valuable for experiments.

Reading a text file containing DaphneIR should not be too complicated since MLIR comes with IR parsers. See, e.g., the mlir-opt tool included in MLIR (thirdparty/llvm-project/mlir/tools/mlir-opt, https://github.com/llvm/llvm-project/tree/main/mlir/tools/mlir-opt).

The task is to extend the existing command-line API of the DAPHNE system to also accept DaphneIR files. The IR must be parsed and connected to the DAPHNE compilation chain, whereby the right entry point must be chosen carefully. Ideally, the compiler should handle it gracefully (or not change it at all) in the early passes, assuming that the compilation chain starts anew when invoking the system with a DaphneIR file. Implementation in C++.

#42: DaphneIR Operations on Scalars via MLIR std/math Operations

DaphneIR is the MLIR/LLVM-based intermediate representation used by the DAPHNE compiler. Currently, almost every operation is lowered to a call to a pre-compiled C++ kernel. In most cases the overhead of a function call is amortized by processing large chunks of data in each call. However, at the moment, even unary and binary operations on scalars (such as +, *, log(), ...) are lowered to kernel calls. When both operands and, thus, the result are scalars, the overhead of calling a kernel can be avoided by lowering these DaphneIR operations to the corresponding operations from MLIR's std and math dialects.

In fact, an earlier version of the prototype was capable of doing that for a few binary operations. However, for consistency this feature was removed in commit 2bc79f74. It would be great to re-introduce this feature in a systematic way for as many unary/binary functions as possible.

The task is to implement (C++) a compiler pass lowering DaphneIR's EwUnaryOp and EwBinaryOp on scalars to the appropriate operations from the MLIR std and math dialects. From that point, existing lowering passes can be applied to JIT-compile executable code. Special care should be taken to ensure that these MLIR operations yield exactly the same results as our pre-compiled kernels for scalars. The reason is that those scalar kernels are re-used in the elementwise kernels for matrices/frames. Thus, any inconsistent behavior would be counter-intuitive. Note also that this should hold for all supported value types. It shall be possible to turn off the new pass. Then, script-level tests could be used to compare the results with and without this feature.

#63: Detection of CSV Schema Based on Data

The DAPHNE system is able to read CSV files into frames, whereby each column may have its individual value type. However, at the moment we require the user to provide a metadata file, which specifies the value type of each column. Unfortunately, this complicates the ad-hoc use of CSV files. Thus, akin to libraries like pandas, we want to be able to automatically detect the value type of each column based on the data.

The task is to implement (C++) an I/O utility for that purpose. The value types to consider are signed/unsigned integers of different widths (8, 32, 64 bit), floating-point types of different widths (single, double), boolean, and string.

#202: Cumulative Aggregation on Dense and Sparse Matrices

The cumulative sum over a sequence of values outputs for each value the sum over all preceding values (including the value itself). Cumulative product, min, and max can be defined similarly. The cumulative aggregation over the columns of a matrix is an important ingredient of many ML algorithms.

The task is to implement (C++) kernels for columnar cumulative aggregation on DAPHNE's DenseMatrix and CSRMatrix. The kernels shall support nxm matrices, but a specialized case for one-column matrices (which can be handled more efficiently) is welcome.

#199: EXPLAIN: Difference of Two Intermediate Representations

The DAPHNE compiler applies numerous optimization and lowering passes on its MLIR/LLVM-based intermediate representation, the DaphneIR. Understanding how the IR changes during the optimization/compilation chain is crucial for identifying shortcomings of the compiler, for illustrating the effect of certain passes, and for debugging the compiler. As the IR of a non-trivial program may consist of thousands of operations, simply printing the entire IR is not always most useful. Instead, a compact comparison of the IR before and after a certain pass would be very useful.

This task is about implementing (C++) a utility that captures the IR before a pass and after a pass, compares them, and prints a meaningful summary of the changes. The IR is essentially an ordered DAG of operations, with a certain textual representation. Thus, the comparison shall be done by a graph difference algorithm, whereby the names of individual values shall be abstracted from to focus on the actual IR structure.

#200: Transposition-aware Matrix Multiplication

Matrix multiplications are at the heart of most ML algorithms and can be very expensive in terms of runtime. There are highly optimized routines (e.g., BLAS) for executing matrix multiplications. In many cases, one or both inputs are transposed matrices. Thus, linear algebra libraries like BLAS can efficiently process matrices without materializing the transposed representation. However, DAPHNE's MatMulOp is currently not aware of whether its inputs are transposed. Thus, an expression like `C = t(A) @ B;` would first transpose A before it calculates the matrix multiplication.

The task is to modify the existing MatMulOp such that it has two boolean flags indicating if the left/right-hand side input is transposed. The DaphneDSL parser shall initially create MatMulOps assuming non-transposed inputs, while a new compiler pass shall identify if an input to a MatMulOp is the result of a TransposeOp and rewrite the program accordingly. Finally, the runtime kernels of the MatMulOp must pass this information on transposition to the BLAS kernels they call internally. Implementation in C++

#201: MCSR Matrix Representation

Currently, the DAPHNE system supports two physical matrix representations: dense and compressed sparse row (CSR). In CSR, a matrix is represented by three dense arrays: values, column indexes, and row-offsets. While this format is suitable for storing and reading sparse matrices, it suffers from inefficient (random) writes/updates. Modified compressed sparse row (MCSR) is a more update-friendly sparse matrix representation. Here, individual values and column index arrays are allocated for each row.

The task is to implement (C++) an MCSRMatrix class, whereby inspiration can be takes from the existing CSR implementation. To see this new data structure in action, a few simple kernels (e.g., elementwise addition) shall be implemented for MCSRMatrix as well. Larger teams may also tackle the integration of MCSRMatrix into DAPHNE's vectorized execution engine.

#203: Elementwise Binary Operations with Matrix and Scalar

In linear algebra programs, elementwise binary operations where one input is a matrix and the other input is a scalar are commonplace. When mixing matrices and scalars, currently the DAPHNE system only supports a matrix on the left-hand side and a scalar on the right-hand side, whereby the user must take care to adhere to this order when writing algorithms in DaphneDSL, DAPHNE's domain-specific language. This requirement shall be relaxed.

The task is twofold: On the one hand, we need canonicalization rewrites for DAPHNE's EwBinaryOp which swap the left-hand side and right-hand side input to ensure matrix-scalar operations, for commutative operations. On the other hand, non-commutative operations shall be handled by a dedicated kernel for scalar-matrix operations (which shall be an adaptation of the existing matrix-scalar kernel). Optionally, non-commutative operations may also be handled by a compiler pass adapting the operands and the operation, e.g., `1 – mat` to `mat + -1`, but note that this is not possible for all operations.

#198: Second-order Functions in DaphneDSL: map()

While DaphneDSL provides a multitude of elementwise unary and binary operations on matrices and frames, users could still have very specific needs that are not fulfilled yet. Thus, we would like to offer a map() function (as typically known from functional languages), which takes a matrix and a user-defined function (UDF) as input and applies the UDF to each element of the matrix.

This task will touch multiple layers of the DAPHNE system, starting from the DaphneDSL parser, where it must be possible to pass a UDF as an argument, over the internal representation and compiler, where a new map-operation must be created along with appropriate lowering passes, down to the runtime, where a generic map-kernel must be provided for the actual execution. Implementation in C++. This task can be scaled depending on the team size, from only elementwise unary functions on dense matrices to unary/binary functions and row/column-wise aggregation on dense/sparse matrices and frames.

#204: Matrix and Frame Data Generators (Dense/Sparse, Properties)

Interesting properties of matrices and frames can be exploited to improve performance by means of special rewrites in the compiler and special algorithms at run-time. When micro-benchmarking such techniques, fine-grained control over the properties of a matrix/frame is invaluable. So far, the randMatrix-kernel supports only the specification of the size, sparsity, and lower/upper bound of a uniform distribution.

The task is to extend the randMatrix-kernel and/or to create multiple variants of it such that additional properties (such as symmetry, #distinct values, value distribution and its parameters, sort order, and column correlations) can be specified. There shall be individual data generators for dense matrices, sparse matrices, and frames. Furthermore, different value types shall be supported by means of template metaprogramming, such as floats (single and double) and (un)signed integers of different widths (8, 32, 64). Implementation in C++.

#x1: Cumulative aggregates CUDA operators in Daphne

Sum/Prod/Min/Max in dense, sparse and "densified" (requires a dens2sparse conversion op). Sparse can be a naive implementation but should be compared to the densified version.

#x2: DNN fallback CPU ops for Daphne

We have most of our neural network operators (convolutions et al) implemented as calls to cuDNN. An initial (pooling) operator is ported from SystemDS. The other operations need a C++ implementation (or porting), test cases and ideally a comparison to the Java version.

#x3: Statistics component in Daphne

Similar to the stats output of SystemDS we would benefit of stats collection in Daphne as well. The challenge would be an approach that has low impact on actual runtime.

#x4: DAG plotting

This one can be SysDS or Daphne (or both with a common core plus framework specifics). Similarly, to the -explain output, that has a certain format, we can save the relevant information in the format of some graph plotting library and call an external tool (or library function) to save a visual representation (preferably vector graphics).

#x5: Continuous (or (semi) automated) performance testing

A small/configurable set of scripts that can be run (ideally as a CI hook in Gitlab/Github) in various configurations of SysDS/Daphne (e.g. with or without GPU/Vectorization/Lineage/Etc) producing a bar plot and pushing that to a git repo to create a performance history.