Noname manuscript No.
(will be inserted by the editor)

SystemML

Declarative Large-Scale Machine Learning

Matthias Boehm

Received: date / Accepted: date

1 Definition

Apache SystemML [4,9] is a system for declarative, large-scale machine learning (ML) that aims to increase the productivity of data scientists. ML algorithms are expressed in a high-level language with R- or Python-like syntax and the system automatically generates efficient hybrid execution plans that are composed of single-node CPU or GPU operations, as well as distributed operations on data-parallel frameworks such as MapReduce [6] or Spark [14]. This high-level abstraction provides the necessary flexibility of specifying custom ML algorithms, while ensuring physical data independence, independence of the underlying runtime operations and technology stack, and scalability for large data. Separating the concerns of algorithm semantics and execution plan generation is essential because it allows for the automatic optimization with regard to different data and cluster characteristics, without the need for algorithm modifications in different deployments.

2 Overview

In SystemML [4,9], data scientists specify their ML algorithms in a language with R- or Python-like syntax, using abstract data types for scalars, matrices and frames, and operations such as linear algebra, element-wise operations, aggregations, indexing, and statistical operations but also control structures such as loops, branches, and functions. These scripts are then parsed into a hierarchy of statement blocks and statements, where control flow delineates the individual blocks. For each block of statements, the system then compiles DAGs (directed acyclic graphs) of high-level operators (HOPs), which is the core internal representation of SystemML's compiler. Size information such as matrix dimensions and sparsity are propagated via intra- and inter-procedural analysis from the inputs through the entire program. This size information is then used to compute memory estimates per operation and accordingly select physical operators, resulting in a DAG of low-level operations (LOPs). These LOP DAGs are finally compiled into executable instructions.

Example: As an example, consider the following script for linear regression via a closed-form method that computes and solves the normal equations:

```
 1:  X = read($X);
 2:  y = read($Y);
 3:  intercept = $icpt; lambda = 0.001;
 4:  if( intercept==1 )
 5:     X = cbind(X, matrix(1, nrow(X), 1));
 6:  I = matrix(1, ncol(X), 1);
 7:  A = t(X) %*% X + diag(I) * lambda;
 8:  b = t(X) %*% y;
 9:  beta = solve(A, b);
10:  write(beta, $B);
```

This script reads the feature matrix $\mathbf{X}$ and labels $\mathbf{y}$, optionally appends a column of 1s to $\mathbf{X}$ for computing the intercept, computes the normal equations, and finally solves the resulting linear system of equations. SystemML then compiles, for example for lines 6-10, a HOP DAG that contains logical operators such as matrix multiplications for $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Given input meta data (e.g., let $\mathbf{X}$ be a dense $10^7 \times 10^3$ matrix), the compiler also computes memory estimates for each operation (e.g., 80.08 GB for $\mathbf{X}^\top\mathbf{y}$). If the memory estimate of an opeation exceeds the local driver memory

Matthias Boehm
IBM Research – Almaden; San Jose, CA, USA
matthias.boehm1@ibm.com

budget, this operation is scheduled for distributed execution and appropriate physical operators are selected (e.g., `mapmm` as a broadcast-based operator for $\mathbf{X}^\top\mathbf{y}$).

Static and Dynamic Rewrites: SystemML's optimizer applies a broad range of optimizations throughout its compilation chain. An important class of optimizations with high performance impact are rewrites. SystemML applies static and dynamic rewrites [3]. Static rewrites are size-independent and include traditional programming language techniques—such as common subexpression elimination, constant folding, and branch removal—algebraic simplifications for linear algebra, as well as backend-specific transformations. For instance in the above example, after constant propagation, constant folding, and branch removal, the block of lines 4-5 is removed if `$icpt=0`, which further allows unconditional size propagation and the merge of the entire program into a single HOP DAG. Furthermore, the expression '$\text{diag}(\text{matrix}(1, \text{ncol}(\mathbf{X}), 1)) \odot \text{lambda}$' is simplified to '$\text{diag}(\text{matrix}(\text{lambda}, \text{ncol}(\mathbf{X}), 1))$', which avoids unnecessary operations and intermediates. SystemML's rewrite system contains hundreds of such rewrites some of which even change the asymptotic behavior (e.g., $\text{trace}(\mathbf{X}\,\mathbf{Y}) \rightarrow \text{sum}(\mathbf{X} \odot \mathbf{Y}^\top)$). Additional dynamic rewrites are size-dependent because they require sizes for cost estimation or validity constraints. Examples are matrix multiplication chain optimization as well as dynamic simplification rewrites such as $\text{sum}(\mathbf{X}^2) \rightarrow \mathbf{X}^\top\mathbf{X} \mid \text{ncol}(\mathbf{X}) = 1$. The former exploits the associativity of matrix multiplications and aims to find an optimal parenthesization for which SystemML applies a textbook dynamic programming algorithm [3].

Operator Selection and Fused Operators: Another important class of optimizations is the selection of execution types and physical operators [4]. SystemML's optimizer analyzes the memory budgets of the driver and executors, and selects—based on worst-case memory estimates [5]—local or distributed execution types. Besides data and cluster characteristics (e.g., data size/shape, memory, and parallelism), the compiler also considers matrix and operation properties (e.g., diagonal/symmetric matrices, sparse-safe operations) as well as data flow properties (e.g., co-partitioning, and data locality). Depending on the chosen execution types, different physical operators are considered with a selection preference from local or special-purpose to shuffle-based operators. For example, the multiplication $\mathbf{X}^\top\mathbf{X}$ from line 7, allows for a special transpose-self operation (`tsmm`), which is an easily parallelizable unary operator that exploits the output symmetry for less computation. For distributed operations, this operator has a blocksize constraint because it requires access to entire rows. If special-purpose or broadcast-based operators do not apply, the compiler falls back to the shuffle-based `cpmm` and `rmm` operators [9]. Additionally, SystemML replaces special patterns with hand-coded fused operators to avoid unnecessary intermediates [7,10] and unnecessary scans [2,3,7], as well as to exploit sparsity across chains of operations [4, 7]. For example, computing the weighted squared loss via $\text{sum}(\mathbf{W} \odot (\mathbf{X} - \mathbf{U}\mathbf{V}^\top)^2)$, would create huge dense intermediates. In contrast sparsity-exploiting operators leverage the sparse driver (i.e., the sparse matrix $\mathbf{W}$ and the sparse-safe multiply $\odot$) for selective computation that only considers non-zeros in $\mathbf{W}$.

Dynamic Recompilation: Dynamic rewrites and operator selection rely on size information for memory estimates, cost estimation, and validity constraints. Hence, unknown dimensions or sparsity lead to conservative fallback plans. Example scenarios are complex conditional control flow or function call graphs, user-defined functions, data-dependent operations, computed size expressions, and changing sizes and sparsity as shown in the following example:

```
while( continue ) {
   parfor( i in 1:n ) {
      if( fixed[1,i] == 0 ) {
          X = cbind(Xg, Xorig[,i]);
          AIC[1,i] = linregDS(X,y); }}
   #select & append best to Xg }
}
```

This example originates from a stepwise linear regression algorithm for feature selection that iteratively selects additional features and calls the previously introduced regression algorithm. SystemML addresses this challenge of unknown sizes via dynamic recompilation [3] that recompiles subplans—at the granularity of HOP DAGs—with exact size information of intermediates during runtime. During initial compilation, operations and DAGs with unknowns are marked for dynamic recompilation, which also includes splitting DAGs after data-dependent operators. During runtime, the recompiler then deep copies the HOP DAG, updates sizes, applies dynamic rewrites, recomputes memory estimates, and finally generates new runtime instructions. This approach yields good plans even with initial unknowns.

Runtime Integration: At runtime level, SystemML then interprets the generated instructions. Single-node and Spark operations directly map to instructions, whereas for the MapReduce backend, instructions are packed into a minimal number of MR jobs. For distributed operations, matrices (and similarly frames) are stored in a blocked representation of pairs of block indexes and squared blocks with fixed blocksize, where individual blocks can be dense, sparse, or ultra-sparse. In contrast, for single-node operations the entire matrix is represented as a single block, which allows reusing the block runtime across backends. Data transfers between the local and distributed backends as well as driver memory management are handled by a multi-level buffer-pool [4] that controls local evictions, parallelizes and collects RDDs, creates broadcasts, and reads/writes data from/to the distributed file system. For example, a single-node instruction first pins its inputs into memory—which triggers reads from HDFS or RDDs if necessary—performs the block operation, registers the output in the buffer pool, and finally unpins its inputs. Many block operations and the I/O system for different formats are multi-threaded to exploit parallelism in scale-up environments. For compute-intensive deep learning workloads, SystemML further calls native CPU and GPU libraries for BLAS and DNN operations. In contrast to deep learning frameworks, SystemML also supports sparse neural network operations. Memory management for the GPU device is integrated with the buffer pool allowing for lazy data transfer on demand. Finally, SystemML uses numerical-stable operations based on Kahan addition for descriptive statistics and certain aggregation functions [13].

3 Key Research Findings

In additional to the compiler and runtime techniques described so far, there are several advanced techniques with high performance impact.

Task-Parallel ParFor Loops: SystemML primary focus is on data parallelism. However, there are many use cases such as ensemble learning, cross validation, hyper-parameter tuning, and complex models with disjoint or overlapping data that are naturally expressed in a task-parallel manner. These scenarios are addressed by SystemML's `parfor` construct for parallel for loops [5]. In contrast to similar constructs in high-performance computing (HPC), `parfor` only asserts the independence of iterations and a dedicated `parfor` optimizer reasons about hybrid parallelization strategies that combine data- and task-parallelism. Reconsider the stepwise linear regression example. Alternative plan choices include (1) a local, i.e., multi-threaded, `parfor` with local operations, (2) a remote `parfor` that runs the entire loop as a distributed Spark job, or (3) a local `parfor` with concurrent data-parallel Spark operations if the data does not fit into the driver.

Resource Optimization: The selection of execution types and operators is strongly influenced by memory budgets of the driver and executor processes. Unfortunately, finding a good static cluster configuration that works well for a broad range of ML algorithms is a hard problem. SystemML addresses this challenge with a dedicated resource optimizer for automatic resource provisioning [10] on resource negotiation frameworks such as YARN or Mesos. The key idea is to optimize resource configurations via an online what-if analysis with regard to the given ML program as well as data and cluster characteristics. This framework optimizes performance without unnecessary over-provisioning, which can increase throughout in shared on-premise clusters and save money in cloud environments.

Compressed Linear Algebra: Furthermore, there is a broad class of iterative ML algorithms that use repeated read-only data access and I/O-bound matrix-vector multiplications to converge to an optimal model. For these algorithms it is crucial for performance to fit the data into available single-node or distributed memory. However, general-purpose, lightweight and heavyweight compression techniques struggle to achieve both good compression ratios and fast decompression to enable block-wise uncompressed operations. Compressed linear algebra (CLA) [8] tackles this challenge by applying lightweight database compression techniques—for column-wise compression with heterogeneous encoding formats and co-coding—to matrices, and executing linear algebra operations such as matrix-vector multiplications directly on the compressed representation. CLA yields compression ratios similar to heavyweight compression and thus allows fitting large datasets into memory, while achieving operation performance close to the uncompressed case.

Automatic Operator Fusion: Similar to query compilation and loop fusion in databases and HPC, the opportunities for fused operators—in terms of fused chains of operations—are ubiquitous. Example benefits are a reduced number of intermediates, reduced number of scans, and sparsity exploitation across operations. Despite their high performance impact, hand-coded fused operators are usually limited to few operators and incur a large development effort. Automatic operator fusion via code generation [7] overcomes this challenge by automatically determining valid fusion plans, and generating access-pattern-aware operators in the form of hand-coded skeletons with custom body code. In contrast to existing work on operator fusion, SystemML introduced a cost-based optimizer framework to find optimal fusion plans in DAGs of linear algebra programs for dense, sparse, and compressed data as well as local and distributed operations.

4 Examples of Application

SystemML has been applied in a variety of ML applications ranging from traditional statistics, classification, regression, clustering problems, over matrix factorization and survival analysis, to deep learning. In contrast to specialized systems for graph analytics like GraphLab [12] or deep learning like TensorFlow [1], SystemML provides a unified system for small to large-scale problems with support for dense, sparse, and ultra-sparse data. Accordingly, SystemML's primary application area is an environment with diverse ML algorithms, data characteristics, and deployments.

Example deployments include large-scale computation on top of MapReduce and Spark, and programmatic APIs for notebook environments or embedded scoring. Thanks to deployment-specific compilation, ML algorithms can be reused without script changes. This flexibility enabled the use in a range of systems with different architectures. For example, SystemML has been shipped as part of the open source project R4ML and the IBM products BigInsights, Data Science Experience (DSX), and multiple Watson services.

5 Future Directions for Research

Given the goal of a unified system for ML applications and recent algorithm and hardware trends, there are many direction for future research throughout the stack of SystemML and similar systems [11]:

Specification Languages: SystemML focuses on optimizing fixed algorithm specifications. However, end-to-end applications would benefit from extensions regarding feature engineering, model selection and life cycle management in general. A promising direction is a stack of declarative languages that allows for reuse.

Optimization Techniques: Regarding the automatic optimization of ML programs, further work is required regarding size and sparsity estimates, adaptive query processing and storage (as an extension of dynamic recompilation), and principled approaches to automatic rewrites and automatic operator fusion.

Runtime Techniques: A better support for deep learning and scientific applications requires the extension from matrices to dense/sparse tensors of different data types and their operations. Additionally, further research is required regarding the automatic exploitation of accelerators and heterogenous hardware.

Benchmarks: Finally, SystemML—but also the community at large—would benefit from dedicated benchmarks for the different classes of ML workloads and different levels of specification languages.

References

1. M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.
2. A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, 2015.
3. M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.
4. M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.
5. M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7), 2014.
6. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
7. T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*, 2017.
8. A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12), 2016.
9. A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.
10. B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.
11. A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*, 2017.
12. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 5(8), 2012.
13. Y. Tian, S. Tatikonda, and B. Reinwald. Scalable and Numerically Stable Descriptive Statistics in SystemML. In *ICDE*, 2012.
14. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

Cross References

TODO