SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning

Tarek Elgamal[2*], Shangyu Luo[3*], Matthias Boehm[1], Alexandre V. Evfimievski[1], Shirish Tatikonda[4†], Berthold Reinwald[1], Prithviraj Sen[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] University of Illinois; Urbana-Champaign, IL, USA
[3] Rice University; Houston, TX, USA
[4] Target Corporation; Sunnyvale, CA, USA

ABSTRACT

Systems for declarative large-scale machine learning (ML) algorithms aim at high-level algorithm specification and automatic optimization of runtime execution plans. State-of-the-art compilers rely on algebraic rewrites and operator selection, including fused operators to avoid materialized intermediates, reduce memory bandwidth requirements, and exploit sparsity across chains of operations. However, the unlimited number of relevant patterns for rewrites and operators poses challenges in terms of development effort and high performance impact. Query compilation has been studied extensively in the database literature, but ML programs additionally require handling linear algebra and exploiting algebraic properties, DAG structures, and sparsity. In this paper, we introduce SPOOF, an architecture to automatically (1) identify algebraic simplification rewrites, and (2) generate fused operators in a holistic framework. We describe a snapshot of the overall system, including key techniques of sum-product optimization and code generation. Preliminary experiments show performance close to hand-coded fused operators, significant improvements over a baseline without fused operators, and moderate compilation overhead.

1. INTRODUCTION

Declarative machine learning (ML) aims at simplifying the usage and development of ML algorithms via a high-level specification of ML tasks or algorithms [8, 29]. Systems for declarative, large-scale ML algorithms allow data scientists to express algorithms in a high-level, analysis-centric language with physical data independence and automatically generated, optimized execution plans. Traditional statistical computing platforms like R [46, 53] and MATLAB [35, 43], or the deep-learning-centric TensorFlow [1] also provide high-level languages and APIs but interpret ML programs as specified. In contrast, state-of-the-art optimizing compilers such as SystemML [9], OptiML [47], and Cumulon [17] commonly rely on algebraic simplification rewrites [9, 47], operator selection [9] and fused operators [9, 17, 47].

Example Rewrites and Fused Operators: Opportunities for rewrites and fused operators are ubiquitous in ML programs. For example, consider the following rewrites: (1) $\mathbf{X}^\top\mathbf{y} \rightarrow (\mathbf{y}^\top\mathbf{X})^\top$, [7, 13] (2) $\text{sum}(\lambda \odot \mathbf{X}) \rightarrow \lambda \odot \text{sum}(\mathbf{X})$, and (3) $\text{trace}(\mathbf{XY}) \rightarrow \text{sum}(\mathbf{X} \odot \mathbf{Y}^\top)$, where $\odot$ denotes the element-wise multiplication. These rewrites avoid large unnecessary intermediates (e.g., $\mathbf{X}^\top$ and $\lambda \odot \mathbf{X}$) and change the asymptotic behavior (e.g., for $\text{trace}(\mathbf{XY})$, from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$) via techniques similar to aggregation and selection push-down. Furthermore, consider the example fused operator patterns: (1) $\text{sum}(\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z})$ [18], (2) $\mathbf{X}^\top(\mathbf{Xv})$ [2, 7], and (3) $\text{sum}(\mathbf{X} \odot \log(\mathbf{UV}^\top))$ [9]. These fused operators also avoid unnecessary intermediates, unnecessary scans of the input $\mathbf{X}$, and change the asymptotic behavior (e.g., for (3), from the number of cells to the number of non-zeros in $\mathbf{X}$, by selectively computing dot products $\mathbf{UV}^\top$ for non-zeros in $\mathbf{X}$). However, fused operators require runtime support.

A Case for Automatic Rewrites and Fusion: Optimization with pattern-matching rewrites and hand-coded fused operators faces two problems. First, the unlimited number of patterns requires a large development effort, especially for multi-backend systems like SystemML. Also, efficient support for dense and sparse matrices requires many operator implementations (e.g., 2^3 for ternary operators). Second, even slightly changed patterns like $\text{sum}(\mathbf{X} \odot \log(\mathbf{UV}^\top + \epsilon))$—to avoid $\log(0)$—can render existing fused operators inapplicable, with huge performance impact for sparsity-exploiting operators. In this paper, we make a case for automatic rewrites and operator fusion for unknown patterns, while still leveraging existing rewrites and operators.

Vision: Our vision is a holistic optimization framework for automatic rewrite identification and operator fusion. Holistic reasoning is important due to increased optimization opportunities and side effects such as common subexpressions and influences between rewrites and fusion potential. Rewrites and fusion of linear algebra operations have been studied for decades [4, 35], but only individually, often with pattern matching rewrites, and without exploiting sparsity across operations. Our core ideas are to break up linear algebra operations into their basic operators, apply elementary sum-product rewrites, and generate hybrid operators of hand-coded skeletons and custom body code.

*Work done during an internship at IBM Research – Almaden.
†Work done while at IBM Research – Almaden

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

Contributions: We introduce SPOOF (Sum-Product Optimization and Operator Fusion), an architecture for automatic rewrite identification and fused operator generation. In detail, we make the following technical contributions:

- *Architecture:* We describe the overall architecture of SPOOF and major design decisions in Section 2.
- *Sum-Product Optimization:* We provide an overview of our sum-product framework in Section 3. It is based on relational algebra, which opens up opportunities to leverage and extend existing optimizer technology.
- *Fused Operator Generation:* We describe the automatic code generation of efficient fused operators in Section 4. This includes the handling of general DAG structures, sparsity-exploiting operators, and effective plan caching across DAGs and during recompilation.
- *Experiments:* Finally, we present preliminary results, reporting on micro-benchmarks and end-to-end performance of various ML algorithms in Section 5.

2. SYSTEM ARCHITECTURE

We integrated the SPOOF compiler framework into Apache SystemML [7, 9]. In this section, we describe its integration into the SystemML compilation chain, as well as the architecture and components of the SPOOF framework.

2.1 SystemML Compiler Integration

To describe the SPOOF compiler integration, we introduce our running example, review SystemML's compilation chain, and discuss a non-invasive integration approach.

Example ML Script: Our running example is Poisson Nonnegative Matrix Factorization (PNMF) [32, 33], where we try to approximate a typically very sparse input matrix $\mathbf{X}$ with two factors $\mathbf{W}$ and $\mathbf{H}$ of low rank k, typically 10-500.

```
 1:  X = read("./input/X")
 2:  k = 100; eps = 1e-15; max_iter = 10; iter = 1;
 3:  W = rand(rows=nrow(X), cols=k, min=0, max=0.025)
 4:  H = rand(rows=k, cols=ncol(X), min=0, max=0.025)
 5:  while( iter < max_iter ) {
 6:    H = (H*(t(W)%*%(X/(W%*%H+eps)))) / t(colSums(W));
 7:    W = (W*((X/(W%*%H+eps))%*%t(H))) / t(rowSums(H));
 8:    obj = sum(W%*%H) - sum(X*log(W%*%H+eps));
 9:    print("iter=" + iter + " obj=" + obj);
10:    iter = iter + 1;
11:  }                              # %*%  .. matrix mult
12:  write(W, "./output/W");        # * .. cell-wise mult
13:  write(H, "./output/H");
```

SystemML uses fused operators `wdivmm left`/`wdivmm right` for the update rules (lines 6 and 7) and `wcemm` for the objective (line 8) as well as rewrites $\text{sum}(\mathbf{WH})$ to $\text{colSums}(\mathbf{W}) \cdot \text{rowSums}(\mathbf{H})$. Together these changes have significant performance impact—for a sparsity of 0.001, up to 1,000x—because they avoid computing the dense intermediate $\mathbf{WH}$.

SystemML Compilation Chain: SystemML compiles such scripts into a hierarchy of statement blocks and statements, where blocks are delineated by control flow (e.g., lines 1-4, 5-11, 6-10). Each last-level block is then compiled into a DAG of high-level operators (HOPs), where we also propagate size information such as dimensions and sparsity from the inputs through the entire program and perform optimizations like constant folding, branch removal, common-subexpression elimination, matrix multiplication chain optimization, and algebraic rewrites. Each HOP DAG is then compiled into a DAG of low-level operators (LOPs) and executable runtime instructions by selecting physical operators including existing fused operators, and additional rewrites.

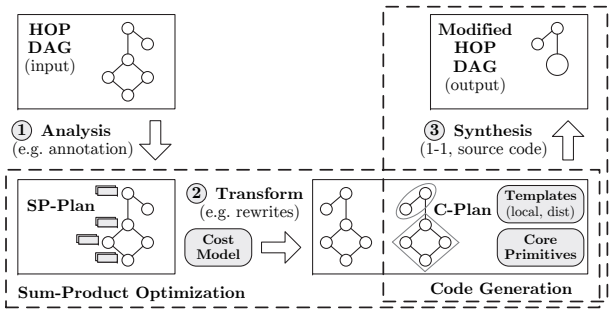


Figure 1: SPOOF Compiler Framework.

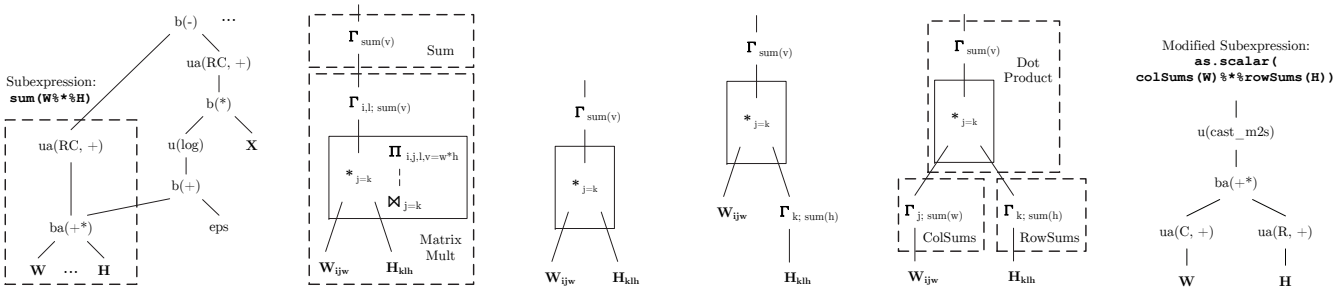
SPOOF Compiler Integration: To leverage the existing compiler and runtime infrastructure, we made the design decision to complement it by SPOOF in a non-invasive manner. We invoke the SPOOF compiler after HOP DAG rewrites and operator selection, separately for each HOP DAG. The SPOOF compiler creates a—potentially modified—HOP DAG. Fused operators are represented via generic HOP and LOP nodes that are parameterized with the generated class name as well as generic instructions that load the generated class into the JVM and call a common operator interface. This HOP DAG integration also ensures a seamless integration with advanced techniques like inter-procedural analysis, `parfor` optimization, and dynamic re-compilation [7].

2.2 SPOOF Compiler Framework

We now describe the SPOOF compiler framework to optimize individual HOP DAGs, including its major components, cost model, and constraints, as well as overall design.

Framework Overview: Figure 1 shows the SPOOF compiler framework. The input HOP DAG also carries propagated size information per operator. In a first *analysis* step, we create Sum-Product Plans (SP-Plans), i.e., restricted relational algebra plans, for relevant partial HOP DAGs. SP-Plans represent operations like matrix multiply as basic operators, i.e., multiply and sum. We also annotate matrix properties and sparse-safeness, i.e., if zero inputs can be ignored. In a second *transform* step, the sum-product optimizer then enumerates alternative SP-Plans via elementary rewrites like the distributive law rewrite. After pruning, we generate Codegen Plans (C-Plans) that are overlay plans of the HOP DAG and represent operator templates and core code generation primitives like `dotProduct`. Costs are evaluated on C-Plans to account for side effects between sum-product optimization and operator fusion. Once the optimal plan is found, we perform a third *synthesis* step, where we (1) map HOPs without C-Plan overlay directly to the output HOP DAG, and (2) generate source code for each C-Plan. Note that C-Plans can overlap, which avoids materialization of intermediates at the cost of redundant computation. Sections 3 and 4 provide additional details of compilation techniques. Finally, we invoke the programmatic Java compiler API to compile generated classes of fused operators.

Cost Model and Constraints: To cost individual C-Plans, we extended our cost model of estimated plan execution time [18] to (1) work directly over C-Plans, and (2) reflect computation (floating point operations) and I/O (HDFS and memory bandwidth). Costing C-Plans instead



(a) Original HOP DAG (b) SP-Plan 1 (c) SP-Plan 2 (d) SP-Plan 3 (e) SP-Plan 4 (f) New HOP DAG

Figure 2: Sum-Product Optimization of $\text{sum}(\mathbf{WH}) \rightarrow \text{colSums}(\mathbf{W}) \cdot \text{rowSums}(\mathbf{H})$.

of runtime plans captures the semantics of generated fused operators, avoids unnecessary compilation overhead, and allows memoization. Furthermore, we impose two constraints on fused operators. First, C-Plans are created for neither single operators nor scalar-scalar operations. This restriction prevents code generation for existing fused operators and more understandable `EXPLAIN` output. Second, fused operators of in-memory operations must satisfy the given memory budget, which prevents out-of-memory errors.

Design Discussion: Finally, we review major design decisions for the overall compiler framework. First, covering both sum-product optimization and code generation in a holistic framework accounts for side effects and opens up more optimization opportunities because sum-product plans do not necessarily need to map to existing operators but can be compiled to custom fused operators. Second, complementing the existing compiler and runtime infrastructure by SPOOF—for automatic rewrites and operator fusion—allowed us to support the general case of all operations and features but compile efficient fused operators where possible. Finally, note that the connection between linear and relational algebra has been explored before. While we use restricted relational algebra as our central plan representation for sum-product optimization and operator fusion, existing work separately studied common language and runtime support [30], ML algorithms over joins [28, 41, 44], algebraic laws [13], a view-based SQL backend for R [53], and instantiating low-level sparse operators [34].

3. SUM-PRODUCT OPTIMIZATION

Sum-product optimization aims to automatically perform algebraic rewrites without coarse-grained pattern matching. To accomplish that, we represent partial DAGs in restricted relational algebra and apply elementary sum-product and relational rewrites. Side effects between rewrites and operator fusion are evaluated by costing alternative plans.

3.1 Plan Representation

Sum-product plans (SP-Plans) are restricted relational algebra plans of partial HOP DAGs. This conceptual framework explicitly represents linear algebra operations like matrix multiply as compositions of basic operators to allow elementary rewrites over sums and products. We construct SP plans for sub-plans with applicable operations.

Data and Operations: Input matrices are relations of (i,j,v)-tuples, i.e., row-/column-indexes, and values. Intermediate relations are tensors of arbitrary dimension. Our framework supports the following operations: selection σ, extended projection Π (for expression computation), aggregation Γ (min, max, sum), and join $\bowtie$. To simplify reasoning and HOP DAG reconstruction, we further introduce

- *Composite Operations* such as multiply $\mathbf{A}_{ij} *_{j=k} \mathbf{B}_{kl} := \Pi_{i,j,l,a \cdot b}(A_{ija} \bowtie_{j=k} B_{klb})$, and
- *Two Restrictions:* (1) a single value attribute per relation, and (2) unique composite indexes per relation; these restrictions ensure a single value per tensor cell.

Sparse-safeness properties are specified via join types: we define element-wise multiplication as $\mathbf{A}_{ij} *_{i=k \wedge j=l} \mathbf{B}_{kl} := \Pi_{i,j,a \cdot b}(A_{ija} \bowtie_{i=k \wedge j=l} B_{klb})$ but element-wise addition—and similarly element-wise subtraction—as full outer join $\mathbf{A}_{ij} +_{i=k \wedge j=l} \mathbf{B}_{kl} := \Pi_{i,j,a+b}(A_{ija} \bowtie\!\!\!\!\!\!\circ_{i=k \wedge j=l} B_{klb})$, where `NULL` values are systematically treated as zeros.

Example SP-Plan $\text{sum}(\mathbf{WH})$**:** Consider the partial HOP DAG for line 8 of our running example, shown in Figure 2(a). The subexpression $\text{sum}(\mathbf{WH})$ is represented via binary aggregate (matrix multiply) and unary aggregate (full sum). Figure 2(b) then shows the related SP-Plan. The matrix multiplication is mapped to (1) a $*_{j=k}$ (i.e., $\bowtie_{j=k}$ over the common dimension and $\Pi_{i,j,l,w \cdot h}$ to compute element-wise multiplications), and (2) $\Gamma_{i,l,\text{sum}}$ to aggregate values per output cell. Similarly, the sum is mapped to a final Γ_{sum}.

3.2 Rewrites and Interesting Properties

Initial SP-Plans are transformed into modified SP-Plans via *elementary sum-product rewrites* leveraging algebraic properties. Examples are distributive and associative laws of addition and multiplication (e.g., $x_1 y_1 + x_1 y_2 \rightarrow x_1(y_1 + y_2)$). The distributive law rewrite applies to $\Gamma_{A,\text{sum}}(\mathbf{X} *_J \mathbf{Y})$ with respect to index $i \in \mathbf{Y}$, iff $i \notin A \wedge i \notin J$. Additionally, we use traditional *relational rewrites* like selection, projection, and aggregation push-down/pull-up [11, 51].

Example SP-Plan Rewrite $\text{sum}(\mathbf{WH})$**:** Given the SP-Plan shown in Figure 2(b), we iteratively apply elementary transformation rewrites. First, in Figure 2(c), we collapse the two subsequent aggregations into a single aggregation over indexes i, j, l, corresponding to $\Sigma_{ijl}(w_{ij} \cdot h_{jl})$. Second, in Figure 2(d), we apply the distributive law rewrite of sum-product and push the aggregation over index l under the join of $*_{j=k}$, corresponding to $\Sigma_{ij}(w_{ij} \cdot \Sigma_l h_{jl})$. Third, in Figure 2(e), we similarly apply the distributive law rewrite to the left matrix and push down the aggregation over index i, corresponding to $\Sigma_j((\Sigma_i w_{ij}) \cdot (\Sigma_l h_{jl}))$. Finally, we map the SP-Plan back to the modified HOP DAG shown in Figure 2(f), where for example $\Gamma_{j,\text{sum}}(\mathbf{W}_{ijw})$ is identified as $\text{colSums}(\mathbf{W})$ due to aggregation over the row index i.

Additional Examples: Sum-product rewrites are very simple yet highly flexible, allowing the composition of a va-

riety of complex rewrites. Consider the following examples, which we classify by relational rewrite categories along with the major sources of performance improvements:

- **Aggregation Push-Down** (reduces number of matrix intermediates and floating point operations):

$$\begin{aligned}&\text{sum}(\lambda \odot \mathbf{X}) \rightarrow \Gamma_{\text{sum}}(\Pi_{i,j,\lambda \cdot x}(\mathbf{X}_{ij}))\\&\quad \rightarrow \Pi_{\lambda \cdot x}(\Gamma_{\text{sum}}(\mathbf{X}_{ij})) \rightarrow \lambda \odot \text{sum}(\mathbf{X})\\&\text{sum}(\mathbf{X}+\mathbf{Y}) \rightarrow \Gamma_{\text{sum}}(\mathbf{X}_{ij} +_{i=k \wedge j=l} \mathbf{Y}_{kl})\\&\quad \rightarrow \Gamma_{\text{sum}}(\mathbf{X}_{ij}) + \Gamma_{\text{sum}}(\mathbf{Y}_{kl})) \rightarrow \text{sum}(\mathbf{X}) + \text{sum}(\mathbf{Y})\end{aligned} \tag{1}$$

- **Selection Push-Down** (improves the asymptotic behavior, e.g., $\mathcal{O}(n^3) \rightarrow \mathcal{O}(n^2)$ and $\mathcal{O}(n^3) \rightarrow \mathcal{O}(n)$ in below examples, assuming squared $n \times n$ input matrices):

$$\begin{aligned}&\text{trace}(\mathbf{XY}) \rightarrow \Gamma_{\text{sum}}(\sigma_{i=l}(\Gamma_{i,l;\text{sum}}(\mathbf{X}_{ij} *_{j=k} \mathbf{Y}_{kl})))\\&\quad \rightarrow \Gamma_{\text{sum}}(\mathbf{X}_{ij} *_{i=l \wedge j=k} \mathbf{Y}_{kl})) \rightarrow \text{sum}(\mathbf{X} \odot \mathbf{Y}^\top)\\&(\mathbf{XY})[a,b] \rightarrow \sigma_{i=a \wedge l=b}(\Gamma_{i,l;\text{sum}}(\mathbf{X}_{ij} *_{j=k} \mathbf{Y}_{kl}))\\&\quad \rightarrow \Gamma_{\text{sum}}(\sigma_{i=a}(\mathbf{X}_{ij}) *_{j=k} \sigma_{l=b}(\mathbf{Y}_{kl})) \rightarrow \mathbf{X}[a,]\mathbf{Y}[,b]\end{aligned} \tag{2}$$

- **Join Elimination** (reduces number of matrix intermediates and floating point operations; changes binary to unary operations, which increases fusion potential):

$$\begin{aligned}&\mathbf{X} \odot \mathbf{X} \rightarrow \mathbf{X}_{ij} *_{i=i \wedge j=j} \mathbf{X}_{ij} \rightarrow \Pi_{i,j,x^2}(\mathbf{X}_{ij}) \rightarrow \mathbf{X}^2\\&\mathbf{X} - \mathbf{X} \odot \mathbf{Y} \rightarrow \mathbf{X}_{ij} -_{i=i \wedge j=j} (\mathbf{X}_{ij} *_{i=k \wedge j=l} \mathbf{Y}_{kl})\\&\quad \rightarrow \mathbf{X}_{ij} *_{i=k \wedge j=l} \Pi_{k,l,1-y}(\mathbf{Y}_{kl}) \rightarrow \mathbf{X} \odot (1-\mathbf{Y})\\&\mathbf{X} \odot (\mathbf{X} > 0) \rightarrow \mathbf{X}_{ij} *_{i=i \wedge j=j} \Pi_{i,j,x>0}(\mathbf{X}_{ij})\\&\quad \rightarrow \sigma_{x>0}(\mathbf{X}_{ij}) \rightarrow \text{codegen required, or } \max(\mathbf{X},0)\\&\mathbf{X} - \mu \odot (\mathbf{X} \neq 0) \rightarrow \mathbf{X}_{ij} -_{i=i \wedge j=j} \Pi_{i,j,\mu \cdot (x \neq 0)}(\mathbf{X}_{ij})\\&\quad \rightarrow \Pi_{i,j,x-\mu}(\sigma_{x \neq 0}(\mathbf{X}_{ij})) \rightarrow \text{codegen required}\end{aligned} \tag{3}$$

- **Join Ordering** (improves the asymptotic behavior, e.g., $\mathcal{O}(n^3) \rightarrow \mathcal{O}(n^2)$ in below example, assuming $n \times n$ input matrices $\mathbf{X}$ and $\mathbf{Y}$ and an $n \times 1$ input vector $\mathbf{v}$):

$$\begin{aligned}&(\mathbf{XY})\mathbf{v} \rightarrow \Gamma_{i;\text{sum}}(\Gamma_{i,l;\text{sum}}(\mathbf{X}_{ij} *_{j=k} \mathbf{Y}_{kl}) *_{l=m} \mathbf{v}_m)\\&\quad \rightarrow \Gamma_{i;\text{sum}}(\mathbf{X}_{ij} *_{j=k} \Gamma_{k;\text{sum}}(\mathbf{Y}_{kl} *_{l=m} \mathbf{v}_m))\\&\quad \rightarrow \mathbf{X}(\mathbf{Yv}) \rightarrow \text{enables fusion in case } \mathbf{X}^\top(\mathbf{Xv})\end{aligned} \tag{4}$$

- **Aggregation Elimination** (reduces number of matrix intermediates and floating point operations; can enable merge of surrounding operations):

$$\begin{aligned}&\text{rowSums}(\mathbf{X}) \rightarrow \Gamma_{i;\text{sum}}(\mathbf{X}_i) \rightarrow \mathbf{X}_i \rightarrow \mathbf{X}\\&\mathbf{X} \odot (\mathbf{v}\ [1,1,\ldots]) \rightarrow \mathbf{X}_{ij} *_{i=k \wedge j=l} \Gamma_{k,l;\text{sum}}(\mathbf{v}_k * 1_l)\\&\quad \rightarrow \mathbf{X}_{ij} *_{i=k} \mathbf{v}_k \rightarrow \mathbf{X} \odot \mathbf{v}\\&\text{diag}(\mathbf{v})\mathbf{X} \rightarrow \Gamma_{i,l;\text{sum}}(\Pi_{i,j=i,v}(\mathbf{v}_i) *_{j=k} \mathbf{X}_{k,l})\\&\quad \rightarrow \mathbf{v}_i *_{i=k} \mathbf{X}_{k,l} \rightarrow \mathbf{X} \odot \mathbf{v}\end{aligned} \tag{5}$$

Note that some of these rewrite examples require a tight integration with operator fusion, i.e., code generation, because their output plans cannot be directly mapped back to HOP DAGs. Furthermore, many rewrites also exploit size information and various structural matrix properties. For example, in Equation (5), we exploit vectors of ones and diagonal matrices to reduce matrix multiplications to simple element-wise multiplications. Therefore, we are interested in systematically leveraging interesting properties.

Interesting Properties: Sum-product optimization aims to annotate enumerated plans with *interesting properties* [19] of two categories: (1) physical properties of intermediate results, and (2) operation properties. First, as physical properties of intermediates, we track row/column/block partitioning, ordering, co-partitioning, and special matrix properties (constant values, sequences, diagonal matrices). Second, we determine types of sparse-safeness for operations and DAGs of operations. For example, the b(*)-rooted operations in Figure 2(a) are fully sparse-safe even though individual operations like b(+)-eps are not sparse-safe.

We leave the transformation rules, the search space analysis, the enumeration algorithms, and further pruning strategies as interesting directions for future work.

3.3 Optimization

Alternative plans are passed to operator fusion for costing without code generation. This allows reasoning about side effects. For example, in Figure 2(a), the sum-product rewrite of sum($\mathbf{WH}$) alone is counterproductive, as it does not leverage the common subexpression $\mathbf{WH}$. However, since we also create a sparsity-exploiting fused operator for $\text{sum}(\mathbf{X} \odot \log(\mathbf{WH}+\epsilon))$, it is indeed crucial for performance because only if applied together, the asymptotic behavior is changed. Furthermore, sum-product rewrites can increase or decrease fusion potential. Similarly, operator fusion fixes the execution order of chains of operations, which in turn decreases rewrite potential. Hence, it is important to reason about sum-product rewrites and fusion in a holistic manner.

Limitations: Our current sum-product framework exchanges plans as HOP DAGs, applies only transformation-based rewrites, and only considers trees. It is interesting future work to integrate it into a dynamic programming plan generator for DAGs [38] with direct mapping to fusion plans, potentially based on a more fine-grained plan representation.

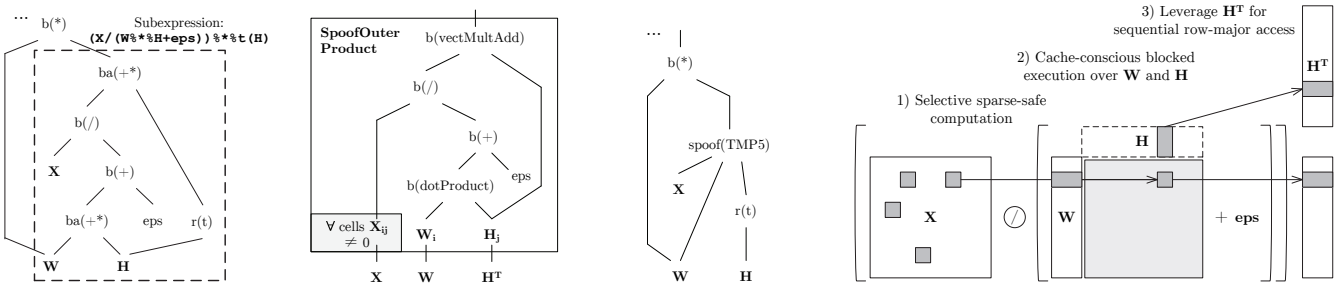
4. FUSED OPERATOR GENERATION

Given a rewritten operator DAG, we then compile fused physical operators in order to (1) exploit sparsity over chains of operators, and (2) reduce the number of materialized intermediates. Our hybrid approach relies on a small set of *meta templates* and *core operation primitives* to compile Java source code for the body of generic fused operators.

4.1 Plan Representation

Code-generation plans (C-Plans) are overlay plans of partial HOP DAGs, where multiple—potentially overlapping—C-Plans can be associated with a single HOP DAG. C-Plans consist of C-Nodes that are either *template* or *primitive* nodes. Template nodes are generic fused operator skeletons that contain a DAG of C-Nodes and exhibit a specific data binding and constraints. Primitive nodes are operations, and edges are scalar or matrix data dependencies.

Templates and Core Primitives: Template skeletons with a generated operator body and core primitives allow efficient handling of sparse and dense input matrices, cache-conscious implementations, single- and multi-threaded skeletons, and the use of vector primitives tuned for instruction-level parallelism. At the same time, custom body code allows a wide variety of operation patterns. Example templates are `SpoofOuterProduct` (sparse-safe operations over outer-product-like matrix multiplications), `SpoofRowAggregate` (row-wise operations with aggregation), and `SpoofCellwise` (cell-wise operations with and without aggregation). All these templates can leverage scalar primitives (unary or binary) or vector primitives (e.g., `dotProduct`, `vectMult`, `vectAdd`, `vectMultAdd`), which also allow for a seamless integration of tuned BLAS level 1 routines.



(a) Original HOP DAG (b) C-Plan (c) New HOP DAG (d) Access Pattern of Fused Operator `TMP5`

Figure 3: Fused Operator Generation for $(\mathbf{X}/(\mathbf{WH}+\epsilon))\mathbf{H}^\top$.

Example C-Plan $(\mathbf{X}/(\mathbf{WH}+\epsilon))\mathbf{H}^\top$: Figure 3(a) shows the partial HOP DAG for line 7 of our running example. The associated C-Plan—shown in Figure 3(b)—leverages the `SpoofOuterProduct` template that binds non-zero cells $\mathbf{X}_{ij}$ of a matrix $\mathbf{X}$ and corresponding rows $\mathbf{W}_i$ and $\mathbf{H}_j$ of an outer-product-like matrix multiply $\mathbf{WH}^\top$. The template body encodes the computation per $\mathbf{X}_{ij}$ cell, using a `dotProduct` of $\mathbf{W}_i$ and $\mathbf{H}_j$, scalar addition and division, as well as `vectMultAdd` (element-wise scalar-vector multiply and add to an output vector). As shown in Figure 3(d), we work over $\mathbf{H}^\top$ for sequential access of our row-major matrix representation. Note that the final b(*) is not fused because this would incur redundant computation before aggregation.

4.2 Plan Alternatives and Cost Model

On C-Plan construction, there are again plan alternatives. For example, overlapping C-Plans require decisions on materialization points, i.e., to create overlapping fused operators with redundant computation, or to materialize results of the shared sub-plan. Another example is the decision on different templates applying to overlapping sub-DAGs. We explore these alternatives with a `MEMO` structure and cost comparisons. The same cost model—extended for entire operator DAGs—is also used for sum-product alternatives.

Cost Model: We use a simple analytical cost model of $C(o_i) = \max(C_{\text{MEM}}(o_i), C_{\text{CPU}}(o_i))$—to explore trade-offs of redundant computation—where $C_{\text{MEM}}(o_i)$ indicates the time for off-chip memory transfers, and $C_{\text{CPU}}(o_i))$ indicates compute time based on the number of floating point operations. Similar to a roofline analysis [50], this model allows determining the influencing cost factor without profiling. In detail, we recursively compute the input data size and the number of weighted floating point operations of the C-Plan, and finally derive C_{MEM} and C_{CPU} times based on peak memory bandwidth and peak floating point performance.

Example C-Plan Costs $(\mathbf{X}/(\mathbf{WH}+\epsilon))\mathbf{H}^\top$: Recall the C-Plan from Figure 3(b) and assume a 100K × 100K input $\mathbf{X}$ with sparsity 0.01 (1.2 GB), two dense 100K × 100 factors $\mathbf{W}$ and $\mathbf{H}$, and peak memory bandwidth of 64 GB/s and peak floating point performance of 230 GFLOP/s. For this scenario, we compute the costs C as follows:

$$\begin{aligned} C_{\text{MEM}} &= (1.2\,\text{GB} + 80\,\text{MB} + 80\,\text{MB}) \;/\; 64\,\text{GB/s} = 21.25\,\text{ms} \\ C_{\text{CPU}} &= 100{,}000^2 \cdot 0.01 \cdot (100 \cdot 2 + 1 + 22 + 100 \cdot 2)\,\text{FLOP} \\ &\quad \;/\; 230\,\text{GFLOP/s} = 183.91\,\text{ms} \\ C &= \max(C_{\text{MEM}}, C_{\text{CPU}}) = 183.91\,\text{ms}, \end{aligned} \tag{6}$$

where C_{CPU} scales the number of floating point operations of the inner C-Plan by the number of non-zero cells in $\mathbf{X}$, and we use a weight of 22 for divisions [20, p. C-14].

4.3 Code Generation

We generate code from each C-Plan by recursively calling `codegen` on its root node. Template nodes instantiate a class template, and call `codegen` on its DAG outputs to produce the body code. Row- and column-wise templates create sparse and dense code separately. Each node has a unique ID from which we derive temporary variable names that are used by all parent nodes. Depth-first code generation ensures the ordering of code fragments by data dependencies. We further adapted register allocation heuristics to handle temporary vector memory requirements. Note that we generate Java source code instead of native code via LLVM to avoid Java-native boundary crossing (via JNI) and because compile time is usually negligible on large data.

Example $(\mathbf{X}/(\mathbf{WH}+\epsilon))\mathbf{H}^\top$: The generated code for our C-Plan from Figure 3(b) is shown below. The generated operator `TMP5` inherits its skeleton from `SpoofOuterProduct` but implements the cellwise computation with custom code. This hybrid of implemented skeletons and generated body allows carefully tuned data access, as shown in Figure 3(d), and generated code similar to hand-coded UDFs [15, p. 6].

```
1: public final class TMP5 extends SpoofOuterProduct {
2:  public TMP5() {
3:    _type = OuterProductType.RIGHT;
4:  }
5:  protected void exec(double a, double[] b, int bi,
6:  double[] c, int ci,...,double[] d, int di, int k) {
7:    double TMP1 = dotProduct(b, c, bi, ci, k); // WH
8:    double TMP2 = TMP1 + 1.0E-15;              // +eps
9:    double TMP3 = a / TMP2;                    // X/
10:   vectMultiplyAdd(TMP3, c, d, ci, di, k);   // t(H)
11: }}
```

HOP DAG/Runtime Integration: After code generation, we replace the partial HOP DAGs associated with C-Plans—i.e., with generated fused operators—by generic `spoof` HOPs as shown in Figure 3(c). These HOPs simply refer to the generated classes by name. Similarly, we provide generic `spoof` LOPs and instructions for a seamless compiler and runtime integration. For distributed Spark operations, the generated classes are shipped via task closures from the driver to the executors and deserialized at the executors.

Limitations: Our code generator does so far neither allow nested templates (e.g., cell-wise within row-wise) nor operations over compressed matrices [14], and supports only a basic generation of distributed Spark operators.

4.4 Plan Caching

For various ML algorithms, the size of intermediates—i.e., their dimensions and sparsity—are unknown during initial compilation. Examples are complex function call patterns,

UDFs, data-dependent operators, size expressions, or changing sizes. Therefore, SystemML uses dynamic recompilation at the granularity of HOP DAGs to adaptively recompile plans during runtime [7]. Size information is also important for SPOOF as many rewrites and templates have size constraints, regarding memory requirements and applicability. Hence, we integrated SPOOF into SystemML's recompiler.

Dynamic Recompilation: Similar to the basic compiler integration described in Section 2.1, we invoke the SPOOF compiler after the update of size information, memory estimates, and HOP DAG rewrites. Dynamic recompilation works on deep copies of original HOP DAGs to enable non-reversible rewrites, which also allows a seamless integration of operator fusion. However, applying operator fusion naïvely during recompilation creates huge overhead because code generation and compilation—with ≈100 ms per fused operator—dominates the basic HOP DAG recompilation, which only takes ≈1 ms for average DAGs.

Plan Cache Probing: As source code compilation is the major bottleneck, we establish a simple yet very effective plan cache to reuse generated operators across DAGs and during dynamic recompilation. This plan cache is a hash map from CPlans to compiled classes, where we recursively compute the hash of a CPlan over its relevant, canonicalized features. For any CPlan constructed during initial compilation or recompilation, we then probe this plan cache, and reuse the generated class in case of an exact match; otherwise we invoke code generation and compilation as described before. Comparing CPlans allows adaptation of optimization decisions but avoids redundant class compilation.

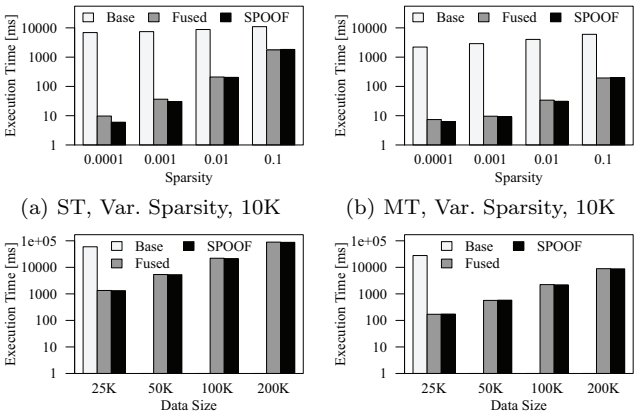
Literal Handling: Dynamic recompilation also includes constant folding and literal replacement. By default, our SPOOF compiler generates constants instead of input variables for literals. However, these literals limit reuse potential because variables like step sizes change in each iteration. Hence, we use a context-sensitive heuristic for literal handling (CSLH): during initial compilation, all literals are generated as constants, whereas during recompilation, any non-integer literals are generated as variables. This heuristic has shown to provide a good compromise between efficiency of generated code and reuse potential of operators.

5. EXPERIMENTS

Our experiments study the impact of sum-product optimization and operator fusion on individual operations and end-to-end ML algorithms, including our running example.

5.1 Experimental Setting

Our setup was a 1+6 node cluster, i.e., one head node of 2x4 Intel E5530 @ 2.40 GHz-2.66 GHz with hyper-threading and 64 GB RAM @800 MHz, as well as 6 nodes of 2x6 Intel E5-2440 @ 2.40 GHz-2.90 GHz with hyper-threading, 96 GB RAM @1.33 GHz (registered ECC), 12x2 TB disks, 10Gb Ethernet, and Red Hat Enterprise Linux Server 6.5. We used OpenJDK 1.8.0, Apache Hadoop 2.2.0, and Apache Spark 1.5.2. Spark was configured with 6 executors, 24 cores/executor, 30 GB driver memory, and 60 GB executor memory. Our baselines are Apache SystemML 0.10 (May 2016), Julia 0.5 (Sep 2016) that uses an LLVM-based just-in-time compiler [5], and peak memory and compute bandwidth. All datasets have been synthetically generated to evaluate a range of scenarios, where we used algorithm-specific data generators for the end-to-end experiments.



(a) ST, Var. Sparsity, 10K (b) MT, Var. Sparsity, 10K

(c) ST, Var. Data Size, 0.01 (d) MT, Var. Data Size, 0.01

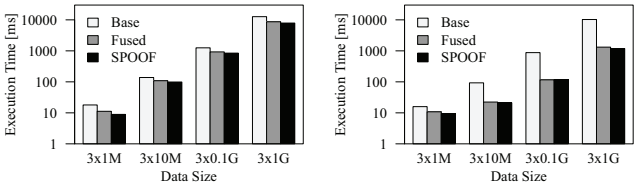
Figure 4: Single-/Multi-Threaded $\mathbf{X}/(\mathbf{WH}+\epsilon)\mathbf{H}^\top$.

5.2 Operations Performance

To understand the runtime characteristics of generated fused operators, we first present single-threaded (ST) and multi-threaded (MT) micro-benchmarks. We use one worker node with 80 GB heap size and report the mean of 20 runs of a baseline without fused operators (Base), hand-coded fused operators (Fused) and generated fused operators (SPOOF).

Fused $\mathbf{X}/(\mathbf{WH}+\epsilon)\mathbf{H}^\top$: The fused `wdivmm` operator exploits sparsity and aggregation. Figures 4(a) and 4(b) show the execution time (in log-scale) over a 10K × 10K input $\mathbf{X}$ with varying sparsity (ratio of non-zeros) and factor rank 100. For a sparsity of 10^{-4}, we see an improvement of three orders of magnitude over the baseline without fused operators. Further, our SPOOF code generation approach matches the performance of the existing `wdivmm` operator; sometimes we even slightly outperform it due to branchless generated code. Figures 4(c) and 4(d) show consistent results with increasing data size at sparsity 0.01. The baseline without fused operators cannot execute the larger scenarios from 50K onwards as the intermediate exceeds the dense block limitation of 16 GB. At MT 200K, we observe a runtime that is at 1/12 of peak floating point performance (230GFLOP/s), which is good considering the sparse input, large factors of 160 MB, the complex operator pattern, and multi-threading.

Fused $\text{sum}(\mathbf{X}\odot\mathbf{Y}\odot\mathbf{Z})$: Complementary to the above operator, the fused `tak+` operator avoids unnecessary intermediates for I/O-bound operations. Figures 5(a) and 5(b) show the execution time (in log-scale) over three dense vectors with varying data size. For single-threaded execution, we see only moderate improvements as the memory bandwidth is not fully utilized yet. However, for multi-threaded execution of large data, we see a substantial improvement of one order of magnitude, achieving peak single-socket local or remote memory bandwidth of ≈ 25 GB/s. The small 3x1M



(a) ST, Var. Data Size, 1.0 (b) MT, Var. Data Size, 1.0

Figure 5: Single-/Multi-Threaded $\text{sum}(\mathbf{X}\odot\mathbf{Y}\odot\mathbf{Z})$.

Table 1: Julia Comparison $\mathbf{X}/(\mathbf{WH}+\epsilon)\mathbf{H}^\top$, MT.

Dataset	Base	Julia	SPOOF
10K × 10K, 0.0001	2,211 ms	1,611 ms	**6 ms**
10K × 10K, 0.001	2,892 ms	1,621 ms	**9 ms**
10K × 10K, 0.01	4,034 ms	1,709 ms	**31 ms**
10K × 10K, 0.1	6,012 ms	1,871 ms	**201 ms**

Table 2: Julia Comparison $\text{sum}(\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z})$, MT.

Dataset	Base	Julia	SPOOF
3 × 1M, 1.0	16 ms	8 ms	**9 ms**
3 × 10M, 1.0	93 ms	76 ms	**21 ms**
3 × 100M, 1.0	877 ms	733 ms	**119 ms**
3 × 1G, 1.0	10,249 ms	11,271 ms	**1,189 ms**

case is an exception because the intermediates of 1M rows (8 MB) still fit into L3 cache (15 MB). These results were obtained with the `-server` configuration; we observed an additional 30% improvement using the `-XX:+UseNUMA` flag.

Comparison to Julia: So far, we compared SPOOF exclusively with SystemML baselines. To further understand the results, we now also compare with Julia that uses an LLVM-based just-in-time compiler. Specifically, we re-evaluate the representative scenarios of Figures 4(b) and 5(b). We ran Julia with `-compile=all` and report the average of 100 runs to amortize compilation overheads. Table 1 shows the results for $\mathbf{X}/(\mathbf{WH}+\epsilon)\mathbf{H}^\top$, comparing SystemML Base, Julia, and SPOOF. Julia uses by default OpenBLAS for matrix multiplications which shows moderately better performance than SystemML Base for this compute-intensive scenario. As the number of non-zeros increases, SystemML Base also suffers from an expensive sparse-dense divide due to repeated output allocations. However, we observe that Julia does not generate sparsity-exploiting code leading to huge performance differences compared to SPOOF. Furthermore, Table 2 shows the results for $\text{sum}(\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z})$. For this scenario, SystemML Base and Julia show almost identical performance as both become memory bandwidth bound and create two large intermediates. In contrast, SPOOF generates a fused operator without intermediates, leading to performance improvements of up to an order of magnitude.

5.3 End-to-End Performance

We now investigate the impact of fused operators on end-to-end performance, i.e., total elapsed time, including compilation and I/O, where we report the mean of 5 runs. As complementary algorithm use cases, we use our running example Poisson Nonnegative Matrix Factorization (PNMF), L2-regularized Support Vector Machine (L2SVM), and Multinomial Logistic Regression (Mlogreg).

PNMF: For PNMF, we generated sparse nonnegative input data of varying dimensions and sparsity 0.001. Table 3 reports the runtime of 20 iterations with rank 100. Similar to our micro-benchmarks, we see huge improvements of sparsity-exploiting fused operators. SPOOF compiled four operators and matches the performance of Fused except a 2 s overhead for compilation. The baseline on scenario 200K resulted in distributed operations, as the dense intermediates were 320 GB, exceeding the driver memory. With the Spark runtime, it failed due to a 2 GB limitation of shuffle files, whereas with the MR runtime it took more than 39 h.

L2SVM: The L2SVM algorithm is a representative iterative ML algorithm, using two nested while loops as well as matrix-vector multiplications and vector operations. We generated dense feature and label inputs $\mathbf{X}$ and $\mathbf{y}$ of a varying number of rows and only 10 features. In this scenario,

Table 3: PNMF End-to-End Execution Time.

Dataset	Base	Fused	SPOOF
10K × 10K, 0.001	251 s	**6 s**	9 s
25K × 25K, 0.001	4,748 s	**9 s**	11 s
200K × 200K, 0.001	>24h	**121 s**	125 s

Table 4: L2SVM End-to-End Execution Time.

Dataset	Base	Fused	SPOOF
100K × 10, 1.0 (8 MB)	3 s	**3 s**	5 s
1M × 10, 1.0 (80 MB)	9 s	7 s	**8 s**
10M × 10, 1.0 (800 MB)	50 s	34 s	**17 s**
100M × 10, 1.0 (8 GB)	525 s	320 s	**114 s**

Table 5: Mlogreg End-to-End Execution Time.

Dataset	Base	Fused	SPOOF
5K × 200, 1.0 (8 MB)	**4 s**	5 s	6 s
50K × 200, 1.0 (80 MB)	6 s	**6 s**	8 s
500K × 200, 1.0 (800 MB)	18 s	**14 s**	**14 s**
5M × 200, 1.0 (8 GB)	122 s	68 s	**57 s**
50M × 200, 1.0 (80 GB)	1,044 s	749 s	**627 s**
50M × 1000, 1.0 (400 GB)	20,152 s	**13,380 s**	14,156 s

the vector operations become the bottleneck. Table 4 reports the runtime of 20 outer iterations with $\lambda = 10^{-3}$ and $\epsilon = 10^{-14}$. The larger the data size, the more we benefit from fused operators such as $\text{sum}(\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})$ (twice in the inner loop), due to fewer intermediates, which determine the amount of data read and written from and to memory as well as potential evictions. SPOOF compiled the entire inner loop into two overlapping fused operators for substantial improvements compared to existing fused operators.

Mlogreg: The Mlogreg algorithm also comprises two nested while loops but with two matrix multiplications for $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{Xv}))$ in the inner loop. We generated dense inputs with 200 features and binomial labels. Table 5 shows the runtime of 20 outer iterations, with up to 5 inner iterations, and $\lambda = 10^{-3}$ and $\epsilon = 10^{-14}$. The larger the data size, the more we benefit from a fused operator for $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{Xv}))$ because it avoids an unnecessary scan of $\mathbf{X}$ from memory. This fused `mmchain` operator computes the chain of operations row-wise, reusing rows of $\mathbf{X}$ from L1 cache. SPOOF additionally compiles various cell-wise operators for a total improvement of more than 2x compared to Base. The same applies to distributed in-memory and out-of-core datasets.

5.4 Example Fused Operators

In order to better understand the end-to-end performance results in Section 5.3, we provide additional details of generated fused operators. Since we already discussed PNMF, we specifically focus on the algorithms L2SVM and Mlogreg.

5.4.1 L2SVM Fused Operators

Below script shows the general algorithm structure and the inner loop operations of L2SVM [18]:

```
1:  while(continueOuter & iter < maxi) {
2:    while(continueInner) {
3:      out = 1 - Y * (Xw + step_sz*Xd);
4:      sv = (out > 0);
5:      out = out * sv;
6:      g = wd + step_sz*dd - sum(out * Y * Xd);
7:      h = dd + sum(Xd * sv * Xd);
8:      step_sz = step_sz - g/h;
9:  } } ...
```

Fusion Potential: Figure 6 shows the HOP DAG of the inner loop (lines 3-7) without fused operators. Vector inputs and operations are highlighted in bold on gray background; the remaining scalar operations are negligible. This HOP

DAG creates ten vector intermediates and computes two vector sums, where the computation paths of both sums share a large common subexpression. SystemML Fused compiles the fused operators `tak+` (twice for $\text{sum}(\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z})$), `selp` (for $\mathbf{X} \odot (\mathbf{X} > 0)$), and `-1*` (for $1 - \mathbf{XY}$), reducing the number of intermediates to five. SystemML 0.11 would additionally compile a `+*` (for $\mathbf{X} + s \odot \mathbf{Y}$). However, pattern-specific fused operators usually comprise only few operations, which fails to exploit the full potential. In contrast, SPOOF eliminates all intermediates by generating two overlapping fused operators of type `SpoofCellwise` with full aggregation.

Example Cell-wise Fused Operator: Understanding the generated source code requires some details of the hand-coded `SpoofCellwise` template. It supports cell-wise operations over one input matrix and arbitrary sideways input vectors with (1) dense and sparse inputs, (2) sparse-safe and -unsafe operations, (3) computation with and without aggregation, as well as (4) single- and multi-threaded execution. This template defines and uses an abstract `exec` method to process one value at-a-time. Refraining from generating all these aspects greatly simplified the code generator and ensures efficiency close to hand-coded operators. Generated cell-wise operators then extend the `SpoofCellwise` operator by implementing a custom `exec` method. Below source code was generated for `SPOOF TMP2` in Figure 6 using $\mathbf{Xd}$ as the main input matrix, and $\mathbf{Y}$ and $\mathbf{Xw}$ as sideways vectors:

```
 1: public final class TMP2 extends SpoofCellwise {
 2:  public TMP2() {
 3:    _type = CellType.FULL_AGG;
 4:  }
 5:  protected double exec(double a, double[][] vectors,
 6:  double[] scalars, ..., int rowIndex) {
 7:    double TMP3 = vectors[1][rowIndex];
 8:    double TMP4 = vectors[0][rowIndex];
 9:    double TMP5 = a * scalars[0];
10:    double TMP6 = TMP4 + TMP5;
11:    double TMP7 = TMP3 * TMP6;
12:    double TMP8 = 1 - TMP7;
13:    double TMP9 = (TMP8 > 0) ? 1 : 0;
14:    double TMP10 = TMP8 * TMP9;
15:    double TMP11 = TMP10 * TMP3;
16:    double TMP12 = TMP11 * a;
17:    return TMP12;
18: }}
```

Under certain conditions (e.g., single `SpoofCellwise` operator), the JVM JIT compiler is able to inline these calls per value; otherwise, the invocation overhead is still often negligible compared to reading from memory.

5.4.2 Mlogreg Fused Operators

Similar to L2SVM, Mlogreg also exhibits two nested while loops but with matrix multiplications in the inner loop. Below script shows the major operations of this inner loop:

```
1: Q = P [, 1:K] * (X %*% ssX_V);
2: HV = t(X)%*%(Q-P[,1:K]*(rowSums(Q)%*%matrix(1,1,K)));
```

For binomial labels, where `P[,1:K]` and `Q` are column vectors, this reduces after rewrites such as $\mathbf{X} \odot (\mathbf{v}\ [1, 1, \ldots]) \rightarrow \mathbf{X} \odot \mathbf{v}$, $\text{rowSums}(\mathbf{X}) \rightarrow \mathbf{X}$, or $\mathbf{X} - \mathbf{X} \odot \mathbf{Y} \rightarrow \mathbf{X} \odot (1 - \mathbf{Y})$—as shown in Equations (3) and (5)—to the following expression:

```
1: HV = t(X)%*%((P[,1:K]*(1-P[,1:K]))*(X %*% ssX_V));
```

Fusion Potential: Without fused operators, this expression creates five column vector intermediates and requires two scans over $\mathbf{X}$. SystemML Fused compiles the operators

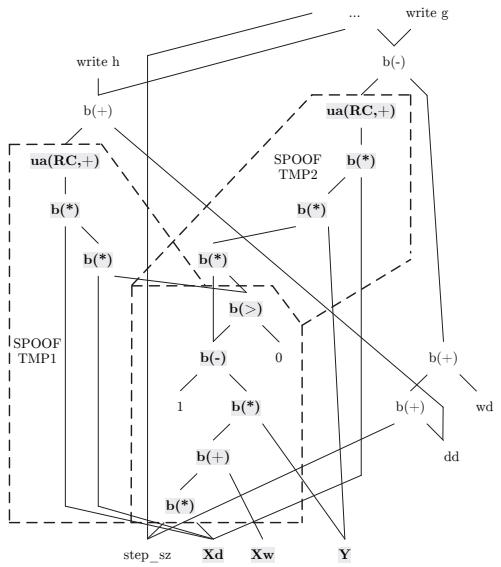


Figure 6: L2SVM HOP DAG of Inner Loop.

`sprop` (sample proportion) for $P \odot (1 - P)$ and `mmchain` for $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{Xv}))$, which only creates two vector intermediates and requires only a single scan over $\mathbf{X}$. In contrast, SPOOF generates a single fused operator for the entire expression, except for the right indexing operation `P[,1:K]`, which eliminates one additional vector intermediate.

Example Row-Aggregate Fused Operator: The `SpoofRowAggregate` template supports row-wise processing over one input matrix and arbitrary sideways vectors with final aggregation over columns or rows. Similar to `SpoofCellwise`, it supports dense and sparse inputs, as well as single- and multi-threaded execution. Since entire rows are exposed to the generated body, this templates comprises two abstract methods `execDense` and `execSparse` for efficient handling of dense and sparse inputs. The generated fused operator for the inner loop of Mlogreg then extends `SpoofRowAggregate` by implementing these two methods as follows:

```
 1: public final class TMP29 extends SpoofRowAggregate {
 2:  public TMP29() {
 3:    _colVector = true;
 4:  }
 5:  protected void execDense(double[] a, int ai,
 6:  double[][] vectors, double[] scalars, double[] c,
 7:  int len, int rowIndex) {
 8:    double TMP17 = vectors[1][rowIndex];
 9:    double TMP18 = 1 - TMP17;
10:    double TMP19 = TMP18 * TMP17;
11:    double TMP20 = dotProduct(a,vectors[0],ai,0,len);
12:    double TMP21 = TMP19 * TMP20;
13:    vectMultiplyAdd(TMP21, a, c, ai, 0, len);
14:  }
15:  protected void execSparse(double[] avals, int[] aix,
16:  int ai, double[][] vectors, double[] scalars,
17:  double[] c, int len, int rowIndex ) {
18:    double TMP23 = vectors[1][rowIndex];
19:    double TMP24 = 1 - TMP23;
20:    double TMP25 = TMP24 * TMP23;
21:    double TMP26 = dotProduct(avals,
22:                     vectors[0], aix, ai, 0, len);
23:    double TMP27 = TMP25 * TMP26;
24:    vectMultiplyAdd(TMP27, avals, c, aix, ai, 0, len);
25: }}
```

Note that for `execSparse`, we directly pass `avals` and `aix`—the value and column index arrays of non-zeros—along with

Table 6: Mlogreg 500K × 200 Plan Cache Statistics.

Statistic	SPOOF no PC	SPOOF PC constant	SPOOF PC CSLH
Total runtime	49.29 s	19.87 s	14.48 s
PC hit rates	0/462	388/462	449/462
Javac compile time	34.45 s	6.88 s	1.97 s
JIT compile time	25.36 s	18.84 s	10.50 s

the current position `ai` in these arrays. This applies to the sparse formats CSR (compressed sparse row) and MCSR (modified CSR), which are both used by SystemML [9].

5.5 Plan Caching Effects

The Mlogreg algorithm is an example that heavily relies on dynamic recompilation because the size of many intermediates depends on the number of classes in the label vector $\mathbf{y}$, which is unknown during initial compilation. Therefore, we use Mlogreg with moderate data size of 500K × 200 (800 MB) to evaluate the impact of our plan cache (PC) described in Section 4.4. We compare SPOOF without PC, SPOOF with PC and constant literals, and SPOOF with PC and our literal heuristic CSLH (our default), as the mean of 5 runs.

Plan Cache Summary Statistics: Table 6 shows the plan cache statistics representing an end-to-end run of Mlogreg. The SPOOF compiler, optimized 473 HOP DAGs, compiled 939 C-Plans (not counting any subsumed plans), and created 462 Java classes. We see that the plan cache has large impact on the total runtime, improving performance by more than 3x on this scenario, with even larger improvements for smaller inputs. Furthermore, we observe that our literal heuristic improves the hit rate from 84% to 97%, which directly affects total Javac compilation time. The remaining SPOOF compilation overheads, including C-Plan construction, are negligible as they did not exceed 300 ms in all runs. Finally, note that the plan cache hit rates also significantly affect just-in-time (JIT) compilation overheads, which in turn indirectly—due to asynchronous JIT compilation—affect the total runtime as well.

Discussion: The simple yet very effective plan cache significantly contributed to the practicability of SPOOF. Most importantly, the good hit rates enable the application of the SPOOF compiler during dynamic recompilation to exploit the full rewrite and fusion potential without worrying about unjustified compilation overhead or code inefficiency.

6. RELATED WORK

We review related work of query compilation as well as sum-product optimization and operator fusion for ML.

Query Compilation: Compiled queries have been studied as far back as in System R [10]. Krikellas et al. reconsidered query compilation and introduced holistic query evaluation [26] to generate query- and hardware-specific code via operator templates. DBToaster further introduced query compilation techniques to generate incremental view maintenance programs [23]. Later Neumann also showed the practicability of LLVM-based query compilation for modern in-memory database systems [36]. Recent systems with query compilation include Hyper [37], Impala [49], Hekaton [16], and Tupleware [12], most of which follow similar template expansion mechanisms. LegoBase [25] and DBLAB/LB [45] further focused on generative programming and a modular compilation chain. Additional work explored query compilation tradeoffs for scans over compressed blocks with heterogeneous compression schemes in Hyper [31], and query engine specialization for heterogeneous data formats like CSV and JSON in Proteus [22]. Orthogonal to query compilation, sideways information passing [21] aims to exploit predicates across chains of operators and sub-queries. None of these query compilers, however, supports linear algebra, sum-product optimization, or sparsity-exploiting operations.

Sum-Product Optimization: Pure sum-product problems have also been studied in various areas [27, 39]. Recent work on regression and factorization over relational block structures [41], generalized linear models over joins [28], and linear regression over factorized joins [44] also implicitly exploit sum-product optimizations over known join dependencies. Similarly, relational rewrites like eager group-by [11, 51] and its generalization for factorized databases [3] can be considered special cases of sum-product optimization. Khamis et al. further generalized sum-product optimization to so called functional aggregate queries [24], where they also showed the relation to matrix multiplication chains. However, none of these works considered the general case of linear algebra programs and its integration into an optimizing compiler. Systems like OptiML [47] and SystemML [7] apply algebraic simplification rewrites, but rely on coarse-grained pattern matching, which does not exploit the full potential. Desharnais et al. further already presented selected algebraic laws of linear algebra, derived from laws of relational algebra [13]. The closest work to sum-product optimization in SPOOF, however, is the AMF (Abstract Matrix Form) framework [35]. Similar to our elementary rewrites, AMF uses a set of axioms, scalar algebraic properties, and axiom-driven transformation rules. Our work differs by using restricted relational algebra plans, and holistically reasoning about sum-product optimization and operator fusion.

Automatic Operator Fusion: Loop or operator fusion has been investigated in high performance computing and database research. First, example systems with hand-coded fused operators are SystemML [9] and Cumulon [17]. SystemML provides various fused operators to exploit sparsity [9] and reduce the number of intermediates or scans over the input [2, 7]. Cumulon uses a so-called `MaskMult` operator to exploit sparsity [17] over sparse-safe, element-wise operations. Ashari et al. [2] relies on source code generation to specialize SystemML's `mmchain` operator for GPUs. Second, Tupleware [12], Kasen [52], and Latte [48] support automatic operator fusion for distributed UDF- and vertex-centric applications. Prior work also tackled the automatic fusion of BLAS level 1 and 2 kernels with an cost-based refine and optimize approach [4]. However, all of these systems neither exploit sparsity nor do they apply algebraic simplifications. Finally, OptiML [47] provides automatic operator fusion for CPUs and GPUs but applicable cases are limited by pattern matching and generated operators do not exploit sparsity. Third, there is also work on generating sparse linear algebra kernels, for different formats, from dense specifications [6, 40]. Mateev et al. further introduced a generic programming approach based on dual APIs, where a restructuring compiler translates from high- to low-level APIs [34]. Interestingly, this work uses relational algebra to model the loop structure within sparse kernels. However, these works have a scope of individual operations and hence cannot exploit sparsity across operations. Recently, Sparso [42] optimizes entire sparse programs by passing context information along chains of operations, but still relies on BLAS libraries.

7. CONCLUSIONS

To summarize, we introduced SPOOF, an architecture for automatic rewrite and fused operator generation, where we discussed a non-invasive compiler integration that complements the existing SystemML compiler and runtime infrastructure. We presented plan representations and compilation techniques for sum-product optimization and source code generation. Our preliminary results show performance close to hand-coded fused operators, huge end-to-end improvements in case of non-existing rewrites or operators, and moderate compilation overhead, even in the context of dynamic recompilation. This paper describes a snapshot of an initial study. We are working on extended operation support for code generation, improved optimization algorithms for sum-product optimization and operator fusion, as well as code generation for distributed Spark operations.

Acknowledgments: We thank Guy Lohman and our reviewers for their valuable comments and suggestions.

8. REFERENCES

[1] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.
[2] A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, 2015.
[3] N. Bakibayev et al. Aggregation and Ordering in Factorised Databases. *PVLDB*, 6(14), 2013.
[4] G. Belter et al. Automating the Generation of Composed Linear Algebra Kernels. In *SC*, 2009.
[5] J. Bezanson et al. Julia: A Fresh Approach to Numerical Computing. *CoRR*, 2014.
[6] A. J. C. Bik et al. The Automatic Generation of Sparse Primitives. *ACM Trans. Math. Softw.*, 24(2), 1998.
[7] M. Boehm et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.
[8] M. Boehm et al. Declarative Machine Learning – A Classification of Basic Properties and Types. *CoRR*, 2016.
[9] M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.
[10] D. D. Chamberlin et al. A History and Evaluation of System R. *Commun. ACM*, 24(10), 1981.
[11] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *VLDB*, 1994.
[12] A. Crotty et al. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12), 2015.
[13] J. Desharnais et al. Relational Style Laws and Constructs of Linear Algebra. *J. Log. Algebr. Meth. Prog.*, 83(2), 2014.
[14] Elgohary et al. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12), 2016.
[15] X. Feng et al. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, 2012.
[16] C. Freedman et al. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.*, 37(1), 2014.
[17] B. Huang et al. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.
[18] B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.
[19] I. F. Ilyas et al. Estimating Compilation Time of a Query Optimizer. In *SIGMOD*, 2003.
[20] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2016.
[21] Z. G. Ives and N. E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *ICDE*, 2008.
[22] M. Karpathiotakis et al. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12), 2016.
[23] O. Kennedy et al. DBToaster: Agile Views for a Dynamic Data Management System. In *CIDR*, 2011.
[24] M. A. Khamis et al. FAQ: Questions Asked Frequently. In *PODS*, 2016.
[25] Y. Klonatos et al. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10), 2014.
[26] K. Krikellas et al. Generating code for holistic query evaluation. In *ICDE*, 2010.
[27] F. R. Kschischang et al. Factor Graphs and the Sum-Product Algorithm. *IEEE Trans. Information Theory*, 47(2), 2001.
[28] A. Kumar et al. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*, 2015.
[29] A. Kumar et al. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 44(4), 2015.
[30] A. Kunft et al. Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In *BeyondMR*, 2016.
[31] H. Lang et al. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, 2016.
[32] D. D. Lee and H. S. Seung. Algorithms for Non-negative Matrix Factorization. In *NIPS*, 2000.
[33] C. Liu et al. Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In *WWW*, 2010.
[34] N. Mateev et al. Next-Generation Generic Programming and its Application to Sparse Matrix Computations. In *ICS*, 2000.
[35] V. Menon and K. Pingali. High-Level Semantic Optimization of Numerical Codes. In *ICS*, 1999.
[36] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9), 2011.
[37] T. Neumann and V. Leis. Compiling Database Queries into Machine Code. *IEEE Data Eng. Bull.*, 37(1), 2014.
[38] T. Neumann and G. Moerkotte. Generating Optimal DAG-Structured Query Evaluation Plans. *Computer Science - R&D*, 24(3), 2009.
[39] H. Poon and P. M. Domingos. Sum-Product Networks: A New Deep Architecture. In *UAI*, 2011.
[40] W. Pugh and T. Shpeisman. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *LCPC*, 1998.
[41] S. Rendle. Scaling Factorization Machines to Relational Data. *PVLDB*, 6(5), 2013.
[42] H. Rong et al. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *PACT*, 2016.
[43] L. D. Rose et al. FALCON: A MATLAB Interactive Restructuring Compiler. In *LCPC*, 1995.
[44] M. Schleich et al. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, 2016.
[45] A. Shaikhha et al. How to Architect a Query Compiler. In *SIGMOD*, 2016.
[46] S. Sridharan and J. M. Patel. Profiling R on a Contemporary Processor. *PVLDB*, 8(2), 2014.
[47] A. K. Sujeeth et al. OptiML: An Implicitly Parallel Domain- Specific Language for Machine Learning. In *ICML*, 2011.
[48] L. Truong et al. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *PLDI*, 2016.
[49] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1), 2014.
[50] S. Williams et al. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4), 2009.
[51] W. P. Yan and P. Larson. Eager Aggregation and Lazy Aggregation. In *VLDB*, 1995.
[52] M. Zhang et al. Measuring and Optimizing Distributed Array Programs. *PVLDB*, 9(12), 2016.
[53] Y. Zhang et al. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.