Apache SystemML: Declarative Large-Scale Machine Learning

Matthias Boehm
IBM Research – Almaden

Acknowledgements:
Case Study: An Automobile Manufacturer

- **Goal:** Design a model to predict car reacquisition

- **Features:**
  - Warranty Claims
  - Repair History
  - Diagnostic Readouts

- **Labels:**
  - Machine Learning Algorithm
  - Algorithm
  - Algorithm
  - Algorithm

- **Predictive Models:**
  - Class skew
  - Low precision

- **Result:** 25x improvement in precision!
Common Patterns Across Customers

- Algorithm customization
- Changes in feature set
- Changes in data size
- Quick iteration
Abstraction: The Good, the Bad and the Ugly

[adapted from Peter Alvaro: "I See What You Mean", Strange Loop, 2015]

\[ q = t(X) \%*\% (w * (X \%*\% v)) \]

Platform Independence
Data Independence
Adaptivity

Simple & Analysis-Centric
Efficiency & Performance

The Ugly: Expectations ≠ Reality

(Missing) Rewrites
Operator Selection
(Missing) Size Information
Complex Control Flow
(Data Skew) Copy-on-Write
(Implicit) Distributed Operations

Local / Remote Memory Budgets
Load Imbalance
Distributed Storage

→ Understanding of optimizer and runtime techniques underpinning declarative, large-scale ML
Tutorial Outline

- Case Study and Motivation (Flash)  5min
- SystemML Overview, APIs, and Tools  30min
- Common Framework  15min
- SystemML’s Optimizer (w/ Hands-On-Labs)  45min
High-Level SystemML Architecture

DML Scripts

Language

Compiler

Runtime

This Talk

In-Memory Single Node (scale-up)

Hadoop or Spark Cluster (scale-out)

DML (Declarative Machine Learning Language)

- Java
- Since 2012
- Hadoop
- Since 2010/11
- Spark
- Since 2015
Running Example

- **Collaborative filtering**
  - Matrix completion
  - Low rank factorization \( X \approx U V^T \)

- **ALS-CG** (alternating least squares via conjugate gradient)
  - L2-regularized squared loss
  - Repeatedly fixes one factor and optimizes the other factor
  - Conjugate Gradient to solve least-squares problems jointly

```
1: X = read($inFile);
2: r = $rank; lambda = $lambda; mi = $maxiter;
3: U = rand(rows=nrow(X), cols=r, min=-1.0, max=1.0);
4: V = rand(rows=r, cols=ncol(X), min=-1.0, max=1.0);
5: W = (X != 0); mii = r; i = 0; is_U = TRUE;
6: while( i < mi ) {
7:   i = i + 1; ii = 1;
8:   if( is_U )
9:     G = (W *(U %*% V - X)) %% t(V) + lambda * U;
10:   else ...
11:     norm_G2 = sum(G^2); norm_R2 = norm_G2; ...
12:   while( norm_R2 > 10E-9 * norm_G2 & ii <= mii ) {
13:     if( is_U )
14:       HS = (W * (S %*% V)) %% t(V) + lambda * S;
15:       alpha = norm_R2 / sum(S * HS);
16:       U = U + alpha * S;
17:     } else {...
18:     ...
19:   }
20:   is_U = !is_U;
21: }
22: write(U, $outUFile, format="text");
23: write(V, $outVFile, format="text");
```
Spark MLContext Scala API:

```scala
import org.apache.sysml.api.mlcontext._
import org.apache.sysml.api.mlcontext.ScriptFactory._
val ml = new MLContext(sc)
val X = // ... RDD, DataFrame, etc.
val script = dmlFromFile("ALS-CG.dml").in("X", X).in(...).out("U", "V")
val (u, v) = ml.execute(script).getTuple[Matrix, Matrix]("U", "V")
```

Spark Command Line:

```
./spark-submit --master yarn-client SystemML.jar \
-f ALS-CG.dml –nvargs X=./in/X ...
```

SystemML Architecture and APIs

- **Command Line**
- **JMLC**
- **Spark MLContext**
- **Spark ML**

**Parser/Language**

**High-Level Operators (HOPs)**

**Low-Level Operators (LOPs)**

**Control Program**

- Recompiler
- Runtime Prog
- Buffer Pool
- Mem/FS IO
- DFS IO

**ParFor Optimizer/Runtime**

- CP Inst
- Spark Inst
- MR Inst
- Generic MR Jobs

**MatrixBlock Library**

(single/multi-threaded)

Cost-based optimizations

IBM Research
Basic Setup and Hands-On-Lab

- **Downloads** (https://systemml.apache.org/download.html)
  - `systemml-0.10.0-incubating` (default cluster setup)
  - `systemml-0.10.0-incubating-standalone` (self-contained local setup)

- **Example script**
  - Test.dml: `print(sum(rand(rows=1000,cols=1000)))`;

- **Basic invocation** (with various execution types)
  - Hadoop (hybrid)
    
    `hadoop jar SystemML.jar -f Test.dml` ...

  - Spark (hybrid_spark)
    
    `./spark_submit -master yarn-client SystemML.jar -f Test.dml` ...

  - Standalone (singlenode, ...)
    
    `./runStandaloneSystemML.sh Test.dml` ...
    `java -cp ... -f Test.dml` ...
SystemML’s Compilation Chain / Overview Tools

DML Script

DEBUG

Language

Parsing (syntactic analysis)
Live Variable Analysis
Validate (semantic analysis)

HOPs

Construct HOP DAGs
Static Rewrites HOP DAGs
Intra-/Inter-Procedural Analysis
Dynamic Rewrites HOP DAGs
Compute Memory Estimates

LOPs

Construct LOP DAGs (incl. operator selection, hop-lop rewrites)
Generate Runtime Program (incl. piggybacking)

EXPLAIN

hops

EXPLAIN

runtime

STATS

Executable Runtime Program (Execution Plan)

Dynamic Recompile

HOP (High-level operator)
LOP (Low-level operator)

Explain (Understanding Execution Plans)

- **Overview**
  - Shows generated execution plan
  - Introduced 05/2014 for internal usage
  ➔ Important tool for understanding/debugging optimizer choices!

- **Usage**
  
  hadoop jar SystemML.jar -f test.dml -explain
  
  [hops | runtime | hops_recompile | runtime_recompile]

  - **Hops**: Program with hop dags after optimization
  - **Runtime** (default): Program with runtime instructions
  - **Hops_recompile**: Hops + hop dag after every recompilation
  - **Runtime_recompile**: Runtime instructions after every recompilation
Explain: Understanding HOP DAGs

- **Example DML script** (simplified LinregDS)

```r
X = read($1);
y = read($2);
intercept = $3;
lambda = $4;
...
if( intercept == 1 ) {
    ones = matrix(1,nrow(X),1);
    X = append(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $5);
```

**Invocation:**
```
hadoop jar SystemML.jar -f LinregDS.dml -args mboehm/X mboehm/y 0 0 mboehm/beta
```

**Scenario:**
- X: 100,000 x 1,000, 1.0
- y: 100,000 x 1, 1.0
- (800MB, 200+GFlop)
Explain: Understanding HOP DAGs (2)

- Explain Hops

16/09/08 09:43:13 INFO api.DMLScript: EXPLAIN (HOPS):
# Memory Budget local/remote = 56627MB/1434MB/1434MB
# Degree of Parallelism (vcores) local/remote = 24/96/48

PROGRAM
--MAIN PROGRAM
-----GENERIC (lines 1-4) [recompile=false]

------(10) PRead X [100000,1000,1000,1000,100000000] [0,0,763 -> 763MB], CP
------(11) TWrite X (10) [100000,1000,1000,1000,100000000] [763,0,0 -> 763MB], CP
------(21) PRead y [100000,1,1000,1000,100000] [0,0,1 -> 1MB], CP
------(22) TWrite y (21) [100000,1,1000,1000,100000] [1,0,0 -> 1MB], CP
------(24) TWrite intercept [0,0,-1,-1,-1] [0,0,0 -> 0MB], CP
------(26) TWrite lambda [0,0,-1,-1,-1] [0,0,0 -> 0MB], CP

-----GENERIC (lines 11-16) [recompile=false]

------(42) TRead X [100000,1000,1000,1000,100000000] [0,0,763 -> 763MB], CP
------(54) r(t) (42) [1000,100000,1000,1000,100000000] [763,0,763 -> 1526MB]
------(55) ba(+) (54,42) [1000,1000,1000,1000,-1] [1526,8,8 -> 778MB], CP
------(43) TRead y [100000,1,1000,1000,100000] [0,0,1 -> 1MB], CP
------(61) ba(+) (54,43) [1000,1,1000,1000,-1] [764,0,0 -> 764MB], CP
------(62) b(solve) (55,61) [1000,1,1000,1000,-1] [8,8,0 -> 15MB], CP
------(68) PWrite beta (62) [1000,1,-1,-1,-1] [0,0,0 -> 0MB], CP

Cluster Characteristics
Program Structure (incl recompile)

Unrolled HOP DAG
Explain: Understanding HOP DAGs (3)

- Explain Hops (cont’)
  
  ----GENERIC (lines 11-16) [recompile=false]
  ------(42) TRead X [100000,1000,1000,1000,100000000] [0,0,763 -> 763MB], CP
  ------(54) r(t) (42) [1000,100000,1000,1000,100000000] [763,0,763 -> 1526MB]
  ------(55) ba(+) (54,42) [1000,1000,1000,1000,-1] [1526,8,8 -> 778MB], CP

- HOP ID
- HOP opcode
- HOP input data dependencies (via HOP IDs)
- HOP output matrix characteristics (rlen, clen, brlen, bclen, nnz)
- Hop memory estimates (inputs, intermediates, output → operation mem)
- Hop execution type (CP/SP/MR)
- Optional: indicators of rblk, chkpt, repart, in-place, etc

- Notes
  - Not all worst-case estimates for dims/memory visible
  - Hops without execution type don’t have corresponding lops (e.g., r(t))
Explain: Understanding Runtime Plans (1)

- Explain Runtime (simplified filenames, removed rmvar)

16/09/08 09:44:22 INFO api.DMLScript: EXPLAIN (RUNTIME):
# Memory Budget local/remote = 56627MB/1434MB/1434MB
# Degree of Parallelism (vcores) local/remote = 24/96/48
PROGRAM ( size CP/MR = 0/0 )
--MAIN PROGRAM
-----GENERIC (lines 1-4) [recompile=false]
------CP createvar pREADX mboehm/X false MATRIX binaryblock 100000 1000 1000 1000 100000000
------CP createvar pREADy mboehm/y false MATRIX binaryblock 100000 1 1000 1000 100000
------CP assignvar 0.SCALAR.INT.true intercept.SCALAR.INT
------CP assignvar 0.SCALAR.INT.true lambda.SCALAR.INT
------CP cpvar pREADX X
------CP cpvar pREADy y
-----GENERIC (lines 11-16) [recompile=false]
------CP createvar _mVar2 .../_t0/temp1 true MATRIX binaryblock 1000 1000 1000 1000 -1
------CP tsmm X.MATRIX.DOUBLE _mVar2.MATRIX.DOUBLE LEFT 24
------CP createvar _mVar3 .../_t0/temp2 true MATRIX binaryblock 1 100000 1000 1000 100000 copy
------CP r' y.MATRIX.DOUBLE _mVar3.MATRIX.DOUBLE 24
------CP createvar _mVar4 .../_t0/temp3 true MATRIX binaryblock 1 1000 1000 1000 -1 copy
------CP ba+_mVar3.MATRIX.DOUBLE X.MATRIX.DOUBLE _mVar4.MATRIX.DOUBLE 24
------CP createvar _mVar5 .../_t0/temp4 true MATRIX binaryblock 1000 1 1000 1000 -1 copy
------CP r' _mVar4.MATRIX.DOUBLE _mVar5.MATRIX.DOUBLE 24
------CP createvar _mVar6 .../_t0/temp5 true MATRIX binaryblock 1000 1 1000 1000 -1 copy
------CP solve _mVar2.MATRIX.DOUBLE _mVar5.MATRIX.DOUBLE _mVar6.MATRIX.DOUBLE
------CP write _mVar6.MATRIX.DOUBLE mboehm/beta.SCALAR.STRING.true textcell

Literally a string representation of runtime instructions
Stats (Profiling Runtime Statistics)

- **Overview**
  - Profiles and shows aggregated runtime statistics
  - Introduced 01/2014 for internal usage
  - **Important tool for understanding runtime characteristics and profiling**

- **Usage**
  
  hadoop jar SystemML.jar -f test.dml -stats
Stats: Understanding Runtime Statistics

### Statistics

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total execution time</td>
<td>4.518 sec.</td>
</tr>
<tr>
<td>Number of compiled MR Jobs</td>
<td>0.</td>
</tr>
<tr>
<td>Number of executed MR Jobs</td>
<td>0.</td>
</tr>
<tr>
<td>Cache hits (Mem, WB, FS, HDFS)</td>
<td>5/0/0/2.</td>
</tr>
<tr>
<td>Cache writes (WB, FS, HDFS)</td>
<td>5/0/1.</td>
</tr>
<tr>
<td>Cache times (ACQr/m, RLS, EXP)</td>
<td>0.830/0.000/0.002/0.204 sec.</td>
</tr>
<tr>
<td>HOP DAGs recompiled (PRED, SB)</td>
<td>0/0.</td>
</tr>
<tr>
<td>HOP DAGs recompile time</td>
<td>0.000 sec.</td>
</tr>
<tr>
<td>Total JIT compile time</td>
<td>0.978 sec.</td>
</tr>
<tr>
<td>Total JVM GC count</td>
<td>2.</td>
</tr>
<tr>
<td>Total JVM GC time</td>
<td>0.184 sec.</td>
</tr>
</tbody>
</table>

**Heavy hitter instructions (name, time, count):**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsmm</td>
<td>3.602 sec</td>
<td>1</td>
</tr>
<tr>
<td>solve</td>
<td>0.585 sec</td>
<td>1</td>
</tr>
<tr>
<td>write</td>
<td>0.205 sec</td>
<td>1</td>
</tr>
<tr>
<td>ba+*</td>
<td>0.070 sec</td>
<td>1</td>
</tr>
<tr>
<td>r'</td>
<td>0.035 sec</td>
<td>2</td>
</tr>
<tr>
<td>createvar</td>
<td>0.000 sec</td>
<td>7</td>
</tr>
<tr>
<td>rmvar</td>
<td>0.000 sec</td>
<td>8</td>
</tr>
<tr>
<td>cpvar</td>
<td>0.000 sec</td>
<td>2</td>
</tr>
<tr>
<td>assignvar</td>
<td>0.000 sec</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Total exec time**
- **Buffer pool stats**
- **Dynamic recompilation stats**
- **JVM stats (JIT, GC)**
- **Heavy hitter instructions (incl. buffer pool times)**

Optional: parfor and update in-place stats (if applicable)
## Tutorial Outline

<table>
<thead>
<tr>
<th>Topic</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case Study and Motivation (Flash)</td>
<td>5min</td>
</tr>
<tr>
<td>SystemML Overview, APIs, and Tools</td>
<td>30min</td>
</tr>
<tr>
<td>Common Framework</td>
<td>15min</td>
</tr>
<tr>
<td>SystemML’s Optimizer (w/ Hands-On-Labs)</td>
<td>45min</td>
</tr>
</tbody>
</table>
ML Program Compilation

- **Script**
  ```
  while(...) {
    q = t(X) %*% (w * (X %*% v))
    ...
  }
  ```

- **Operator DAG**
  - a.k.a. “graph”
  - a.k.a. intermediate representation (IR)

- **Runtime plans**
  - Interpreted plans
  - Compiled runtime plans (e.g., instructions)

**Data dependency** (incl data flow properties)

- [Multiple] DAG roots (outputs)
- [Multiple] Consumers of intermediates
- [Multiple] DAG leafs (inputs)
- No cycles

SPARK `mapmmchain X.MATRIX.DOUBLE w.MATRIX.DOUBLE v.MATRIX.DOUBLE _mVar4.MATRIX.DOUBLE XtwXv`
Distributed Matrix Representation

- **Collection of “matrix blocks” (and keys)**
  - Bag semantics (duplicates, unordered)
  - Logical (fixed-size) blocking
    + join processing / independence
  - (sparsity skew)
  - E.g., SystemML on Spark:
    JavaPairRDD<MatrixIndexes,MatrixBlock>
  - Blocks encoded independently (dense/sparse)

- **Partitioning**
  - Logical partitioning
    (e.g., row-/column-wise)
  - Physical partitioning
    (e.g., Hash / Grid)
Distributed Matrix Representation (2)

- **Matrix block**
  - Most operations defined here
  - Local matrix: single block
  - Different representations

- **Common block representations**
  - Dense (linearized arrays)
  - MCSR (modified CSR)
  - CSR (compressed sparse rows), CSC
  - COO (Coordinate matrix)
  - ...

**Example 3x3 Matrix**

Dense (row-major)
```
0.7  0.1  0.2  0.4  0  0  0.3  0
```

MCSR
```
0 2
0.7 0.1
0 1
0.2 0.4
1
```

CSR
```
0 0 0.7
2 2 0.1
4 0 0.2
5 1 0.4
1
```

COO
```
0 0 0.7
0 2 0.1
1 0 0.2
1 1 0.4
2 1 0.3
```
Common Workload Characteristics

- **Common operations**
  - Matrix-Vector $X \mathbf{v}$ (e.g., LinregCG, Logreg, GLM, L2SVM, PCA)
  - Vector-Matrix $\mathbf{v}^T X$ (e.g., LinregCG, LinregDS, Logreg, GLM, L2SVM)
  - MMChain $X^T (\mathbf{w}^* \mathbf{X} \mathbf{v})$ (e.g., LinregCG, Logreg, GLM)
  - TSMM $X^T X$ (e.g., LinregDS, PCA)

- **Common data characteristics**
  - Tall and skinny matrices
  - Wide matrices often sparse
  - Non-uniform sparsity
  - Transformed data often w/ low column cardinality
  - Column correlations

### LinregCG (Conjugate Gradient)

1: $X = \text{read}(\$1)$; # $n \times m$ matrix
2: $y = \text{read}(\$2)$; # $n \times 1$ vector
3: maxi = 50; lambda = 0.001;
4: intercept = $\$3$;
5: ...
6: $r = (t(X) \%\% y)$;
7: norm_r2 = sum(r * r); p = -r;
8: $w = \text{matrix}(0, \text{ncol}(X), 1)$; i = 0;
9: while(i<maxi & norm_r2>norm_r2_trgt) {
10:     $q = (t(X) \%\% (X \%\% p)) + \text{lambda} * p$;
11:     alpha = norm_r2 / sum(p * q);
12:     $w = w + \text{alpha} * p$;
13:     old_norm_r2 = norm_r2;
14:     $r = r + \text{beta} * p$; i = i + 1;
15:     norm_r2 = sum(r * r);
16:     beta = norm_r2 / old_norm_r2;
17:     $p = -r + \text{beta} * p$; i = i + 1;
18: }
19: write($w$, $\$4$, format="text");
Excursus: Roofline Analysis Matrix-Vector Multiply

**Single Node**: 2x6 E5-2440 @2.4GHz–2.9GHz, DDR3 RAM @1.3GHz (ECC)
- Max mem bandwidth (local): 2 sock x 3 chan x 8B x 1.3G trans/s $\rightarrow$ 2 x 32GB/s
- Max mem bandwidth (QPI, full duplex) $\rightarrow$ 2 x 12.8GB/s
- Max floating point ops: 12 cores x 2*4dFP-units x 2.4GHz $\rightarrow$ 2 x 115.2GFlops/s

**Roofline Analysis**
- Processor performance
- Off-chip memory traffic

---


$\rightarrow$ IO-bound matrix-vector mult
Tutorial Outline

- Case Study and Motivation (Flash)  5min
- SystemML Overview, APIs, and Tools  30min
- Common Framework  15min
- SystemML’s Optimizer (w/ Hands-On-Labs)  45min
Recap: SystemML’s Compilation Chain

DML Script

Language
- Parsing (syntactic analysis)
- Live Variable Analysis
- Validate (semantic analysis)

HOPs
- Construct HOP DAGs
- Static Rewrites HOP DAGs
- Intra-/Inter-Procedural Analysis
- Dynamic Rewrites HOP DAGs
- Compute Memory Estimates

LOPs
- Construct LOP DAGs (incl operator selection, hop-lop rewrites)
- Generate Runtime Program (incl piggybacking)

Executable Runtime Program (Execution Plan)

Various optimization decisions at different compilation steps

[IPA
ParFor Opt
Resource Opt
GDF Opt]

HOP (High-level operator)
LOP (Low-level operator)

Basic HOP and LOP DAG Compilation

**Example LinregDS**

- **HOP DAG** (after rewrites)
  - \( X = \text{read}(\$1); \)
  - \( y = \text{read}(\$2); \)
  - intercept = \$3; lambda = 0.001;
  - if (intercept == 1) {
    - ones = matrix(1, nrow(X), 1);
    - X = append(X, ones);
  }
  - I = matrix(1, ncol(X), 1);
  - A = t(X) %*% X + diag(I)*lambda;
  - b = t(X) %*% y;
  - beta = solve(A, b);
  - write(beta, \$4);\
  - \( X: 10^8 \times 10^3, 10^{11} \)
  - \( y: 10^8 \times 1, 10^8 \)

- **LOP DAG** (after rewrites)
  - \( X: 10^8 \times 10^3, 10^{11} \)
  - \( y: 10^8 \times 1, 10^8 \)

**Cluster Config:**
- client mem: 4 GB
- map/red mem: 2 GB

**Hybrid Runtime Plans:**
- Size propagation over ML programs
- Worst-case sparsity / memory estimates
- Integrated CP / MR / Spark runtime
Static and Dynamic Rewrites

- **Types of Rewrites**
  - Static: size-independent rewrites
  - Dynamic: size-dependent rewrites

- **Examples Static Rewrites**
  - Common Subexpression Elimination
  - Constant Folding
  - Static Algebraic Simplification Rewrites
  - Branch Removal
  - Right/Left Indexing Vectorization
  - For Loop Vectorization
  - Checkpoint injection (caching)
  - Repartition injection

- **Examples Dynamic Rewrites**
  - Matrix Multiplication Chain Optimization
  - Dynamic Algebraic Simplification Rewrites

**Cascading rewrite effect**
(ensures other rewrites, IPA, operator selection)

**High performance impact**
(direct/indirect)
### Example Static Simplification Rewrites

#### Static Simplification Rewrites (size-independent patterns)

<table>
<thead>
<tr>
<th>Rewrite Category</th>
<th>Static Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Unnecessary Operations</td>
<td>t(t(X)), X/1, X*1, X-0, -(-X) → X</td>
</tr>
<tr>
<td></td>
<td>matrix(1,)/X → 1/X</td>
</tr>
<tr>
<td></td>
<td>sum(t(X)) → sum(X)</td>
</tr>
<tr>
<td></td>
<td>rand(,min=-1,max=1)*7 → rand(,min=-7,max=7)</td>
</tr>
<tr>
<td></td>
<td>-rand(,min=-2,max=1) → rand(,min=-1,max=2)</td>
</tr>
<tr>
<td></td>
<td>t(cbind(t(X),t(Y))) → rbind(X,Y)</td>
</tr>
<tr>
<td>Simplify Bushy Binary</td>
<td>(X*(Y*(Z%<em>%v))) → (X</em>Y)<em>(Z%</em>%v)</td>
</tr>
<tr>
<td>Binary to Unary</td>
<td>X+X → 2*X</td>
</tr>
<tr>
<td></td>
<td>X*X → X^2</td>
</tr>
<tr>
<td></td>
<td>X-X<em>Y → X</em>(1-Y)</td>
</tr>
<tr>
<td></td>
<td>X*(1-X) → sprop(X)</td>
</tr>
<tr>
<td></td>
<td>1/(1+exp(-X)) → sigmoid(X)</td>
</tr>
<tr>
<td></td>
<td>X*(X&gt;0) → selp(X)</td>
</tr>
<tr>
<td></td>
<td>(X-7)*(X!=0) → X -nz 7</td>
</tr>
<tr>
<td></td>
<td>(X!=0)*log(X) → log_nz(X)</td>
</tr>
<tr>
<td></td>
<td>aggregate(X,y,count) → aggregate(y,y,count)</td>
</tr>
<tr>
<td>Simplify Permutation Matrix Construction</td>
<td>outer(v,seq(1,N),&quot;==&quot;) → rexpand(v,max=N,row)</td>
</tr>
<tr>
<td></td>
<td>table(seq(1,nrow(v)),v,N) → rexpand(v,max=N,row)</td>
</tr>
<tr>
<td>Simplify Operation over Matrix Multiplication</td>
<td>trace(X%<em>%Y) → sum(X</em>t(Y))</td>
</tr>
<tr>
<td></td>
<td>(X%<em>%Y)[7,3] → X[7,] %</em>% Y[,3]</td>
</tr>
</tbody>
</table>
Example Dynamic Simplification Rewrites

- **Dynamic Simplification Rewrites** *(size-dependent patterns)*

<table>
<thead>
<tr>
<th>Rewrite Category</th>
<th>Dynamic Patterns</th>
</tr>
</thead>
</table>
| **Remove / Simplify Unnecessary Indexing**   | X[a:b,c:d] = Y → X = Y **iff** dims(X)=dims(Y)  
X = Y[, 1] → X = Y **iff** dims(X)=dims(Y)  
X[,1]=Y;X[,2]=Z → X=cbind(Y,Z) **iff** ncol(X)=2,col |
| **Fuse / Pushdown Operations**                | t(rand(10, 1)) → rand(1, 10) **iff** nrow/ncol=1  
sum(diag(X)) → trace(X) **iff** ncol(X)>1  
diag(X)*7 → diag(X*7) **iff** ncol(X)=1  
sum(X^2) → t(X)%%X, →sumSq(X) **iff** ncol(X)=1, >1 |
| **Remove Empty / Unnecessary Operations**     | X**%Y → matrix(0,...) **iff** nnz(X)=0|nnz(Y)=0  
X*Y → matrix(0,...), X+Y→X, X-Y→X **iff** nnz(Y)=0  
round(X)→matrix(0), t(X)→matrix(0) **iff** nnz(X)=0  
X*(Y**%matrix(1,)) → X*Y **iff** ncol(Y)=1 |
| **Simplify Aggregates / Scalar Operations**   | rowSums(X) → sum(X) → X **iff** nrow(X)=1, ncol(X)=1  
rowSums(X*Y) → X**%t(Y) **iff** nrow(Y)=1  
X*Y → X*as.scalar(Y) **iff** nrow(Y)=1&ncol(Y)=1 |
| **Simplify Diag Matrix Multiplications**      | diag(X)**%Y → Y*X **iff** ncol(X)=1&ncol(Y)>1  
diag(X**%Y)->rowSums(X*t(Y)) **iff** ncol(Y)>1 |
Hands-On Labs: Rewrites and Handling of Size Information

- **Exercise 1: Sum-Product Rewrite:** \( \text{sum}(A \times t(B)) \)
  - a) What’s happening for \( A:=[900\times1000], B:=[700,1000] \)
  - b) What’s happening for \( A:=[900\times1], B:=[700\times1] \)

- **Exercise 2: Matrix Multiplication Chains:** \( A \times B \times C \times D \times E \)
  - What’s happening as we change dimensions of \( A, B, C, D, E \)
    (start with dimensions given on slide 17)

- **Exercise 3: Dynamic Recompilation**
  - What’s happening during compilation/runtime to gather size information

```r
if( $1 == 1 ) {
  Y = \text{rand}(\text{rows}=\text{nrow}(X), \text{cols}=1, \text{min}=1, \text{max}=\text{maxval});
  X = \text{cbind}(X, \text{table}(\text{seq}(1,\text{nrow}(Y)),Y));}
\text{print(sum}(X));
```
Matrix Multiplication Chain Optimization

- **Problem**
  - Given a matrix multiplication chain (sequence) of n matrices $M_1, M_2, \ldots M_n$
  - Matrix multiplication is associative
  - Find the optimal full parenthesization of the product $M_1M_2 \ldots M_n$

- **Search Space Characteristics**
  - Naïve exhaustive search: Catalan numbers $\Omega(4^n / n^{3/2})$
  - Few distinct subproblems: any i and j, w/ $1 \leq i \leq j \leq n$: $\Theta(n^2)$
  - DP characteristics apply: (1) optimal substructure, (2) overlapping subproblems
  - Text book dynamic programming algorithm: $\Theta(n^3)$ time, $\Theta(n^2)$ space
  - Best known algorithm: $O(n \log n)$

Matrix Multiplication Chain Optimization (2)

<table>
<thead>
<tr>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x7</td>
<td>7x5</td>
<td>5x1</td>
<td>1x3</td>
<td>3x9</td>
</tr>
</tbody>
</table>

Cost matrix $m$

$m[1,3] = \min(m[1,1] + m[2,3] + p_1 p_2 p_4, m[1,2] + m[3,3] + p_1 p_3 p_4) = \min(0 + 35 + 10 \times 7 \times 1, 350 + 0 + 10 \times 5 \times 1) = \min(105, 400)$
Matrix Multiplication Chain Optimization (3)

Cost matrix $m$

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10x7</td>
<td>7x5</td>
<td>5x1</td>
<td>1x3</td>
<td>3x9</td>
</tr>
</tbody>
</table>

Optimal split matrix $s$

$$\begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array}$$

$$(M_1 (M_2 M_3)) (M_4 M_5)$$

getOpt(s,1,5)  
getOpt(s,1,3)  
getOpt(s,4,5)

$$(M_1 M_2 M_3 M_4 M_5)$$

$$( (M_1 M_2 M_3) (M_4 M_5) )$$

$$( (M_1 (M_2 M_3)) (M_4 M_5) )$$
Hands-On Labs:
Rewrites and Handling of Size Information

- **Exercise 1: Sum-Product Rewrite:** \( \text{sum}(A \ %*% \ t(B)) \)
  - a) What’s happening for \( A:=[900x1000], B:=[700,1000] \)
  - b) What’s happening for \( A:=[900x1], B:=[700x1] \)

- **Exercise 2: Matrix Multiplication Chains:** \( A \ %*% B \ %*% C \ %*% D \ %*% E \)
  - What’s happening as we change dimensions of \( A, B, C, D, E \)
    (start with dimensions given on slide 17)

- **Exercise 3: Dynamic Recompilation**
  - What’s happening during compilation/runtime to gather size information
    ```
    if( $1 == 1 ) {
      Y = \text{rand}(\text{rows}=\text{nrow}(X), \text{cols}=1, \text{min}=1, \text{max}=\text{maxval});
      X = \text{cbind}(X, \text{table}(\text{seq}(1,\text{nrow}(Y)), Y));
    }
    \text{print}(\text{sum}(X));
    ```
### Example Operator Selection: Matrix Multiplication

<table>
<thead>
<tr>
<th>Exec Type</th>
<th>MM Ops</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>MM</td>
<td>X %**% Y</td>
</tr>
<tr>
<td></td>
<td>MMChain</td>
<td>t(X) (w * (X %**% v))</td>
</tr>
<tr>
<td></td>
<td>TSMM</td>
<td>t(X) %**% X</td>
</tr>
<tr>
<td></td>
<td>PMM</td>
<td>rmr(diag(v)) %**% X</td>
</tr>
<tr>
<td>MR / Spark</td>
<td>MapMM</td>
<td>X %**% Y</td>
</tr>
<tr>
<td></td>
<td>MapMMChain</td>
<td>t(X) (w * (X %**% v))</td>
</tr>
<tr>
<td></td>
<td>TSMM</td>
<td>t(X) %**% X</td>
</tr>
<tr>
<td></td>
<td>ZipMM</td>
<td>t(X) %**% Y</td>
</tr>
<tr>
<td></td>
<td>CPMM</td>
<td>rmr(diag(v)) %**% X</td>
</tr>
<tr>
<td></td>
<td>RMM</td>
<td>X %**% Y</td>
</tr>
<tr>
<td></td>
<td>PMM</td>
<td>X %**% Y</td>
</tr>
</tbody>
</table>

- **Hop-Lop Rewrites**
  - Aggregation (w/o, singleblock/multiblock)
  - Partitioning (w/o, CP/MR, col/rowblock)
  - Empty block materialization in output
  - Transpose-MM rewrite $t(X)%**%y \rightarrow t(t(y)%**%X)$
  - CP degree of parallelism (multi-threaded mm)
Example Fused Operators (1): MMChain

- **Matrix Multiplication Chains:** \( q = t(X) \%\% (w \ast (X \%\% v)) \)
  - Very common pattern
  - MV-ops IO / memory-bandwidth bound
  - **Problem:** Data dependency forces two passes over \( X \)

- **Fused mmchain operator**
  - **Key observation:** values of \( D \) are row-aligned wrt to \( X \)
  - **Single-pass operation** (map-side in MR/Spark / cache-conscious in CP/GPU)

[Arash Ashari et al.: On optimizing machine learning workloads via kernel fusion. PPOPP 2015]
Example Fused Operators (2): WSLoss

- **Weighted Squared Loss:** \( \text{wsl} = \text{sum}(W * (X - L \%\% t(R))^2) \)
  - Common pattern for factorization algorithms
  - \( W \) and \( X \) usually very sparse (< 0.001)
  - **Problem:** “Outer” product of \( L \%\% t(R) \) creates **three dense** intermediates in the size of \( X \)

→ **Fused wsloss operator**
- **Key observations:** Sparse \( W^* \) allows selective computation, full aggregate significantly reduces memory requirements

[Matthias Boehm et al.: SystemML: Declarative Machine Learning on Spark. VLDB 2016]
Rewrites and Operator Selection in Action

- **Example:** Use case Mlogreg, X: $10^8 \times 10^3$, K=1 (2 classes), 2GB mem

- **Applied Rewrites**
  - Original DML snippet of inner loop:
    \[
    Q = P[, 1:K] \times (X \%\% ssX_V);
    HV = t(X) \%\% (Q - P[, 1:K] \times (rowSums(Q) \%\% matrix(1, rows=1, cols=K)));
    \]
  - After remove unnecessary (1) matrix multiply (2) unary aggregate
    \[
    Q = P[, 1:K] \times (X \%\% ssX_V);
    HV = t(X) \%\% (Q - P[, 1:K] \times Q);
    \]
  - After simplify distributive binary operation
    \[
    Q = P[, 1:K] \times (X \%\% ssX_V);
    HV = t(X) \%\% ((1 - P[, 1:K]) \times Q);
    \]
  - After simplify bushy binary operation
    \[
    HV = t(X) \%\% (((1 - P[, 1:K]) \times P[, 1:K]) \times (X \%\% ssX_V));
    \]
  - After fuse binary dag to unary operation (sample proportion)
    \[
    HV = t(X) \%\% (sprop(P[, 1:K] \times (X \%\% ssX_V));
    \]

- **Operator Selection**
  - Exec Type: **MR**, because mem estimate > 800GB
  - MM Type: **MapMMChain**, because $XtwXv$ and $w=sprop(P[,1:K]) < 2GB$
  - CP partitioning of $w$ into 32MB chunks of rowblocks

Recall: Cascading rewrite effect
Dynamic Recompilation – Motivation

- Problem of unknown/changing sizes
  - Unknown or changing sizes and sparsity of intermediates (across loop iterations / conditional control flow).
  - These unknowns lead to very conservative fallback plans.

- Example ML Program Scenarios
  - Scripts w/ complex function call patterns
  - Scripts w/ UDFs
  - Data-dependent operators
    \[
    Y = \text{table}( \text{seq}(1, \text{nrow}(X)), y ) \\
    \text{grad} = \text{t}(X) \times (P - Y);
    \]
  - Computed size expressions
  - Changing dimensions or sparsity

- Dynamic recompilation techniques as robust fallback strategy
  - Shares goals and challenges with adaptive query processing
  - However, ML domain-specific techniques and rewrites

Ex: Stepwise LinregDS

```r
while( continue ) {
  parfor( i in 1:n ) {
    if( fixed[1,i]==0 ) {
      X = \text{cbind}(Xg, Xorig[,i])
      AIC[1,i] = \text{linregDS}(X,y)
    }
  }
  #select & append best to Xg
}
```
Dynamic Recompilation – Compiler and Runtime

- **Optimizer Recompilation Decisions**
  - **Split HOP DAGs for recompilation**: prevent unknowns but keep DAGs as large as possible; we split after reads w/ unknown sizes and specific operators
  - **Mark HOP DAGs for recompilation**: MR due to unknown sizes / sparsity

- **Dynamic Recompilation at Runtime** on recompilation hooks (last level program blocks, predicates, recompile once functions, specific MR jobs)
  - **Deep Copy DAG**: (e.g., for non-reversible dynamic rewrites)
  - **Update DAG Statistics**: (based on exact symbol table meta data)
  - **Dynamic Rewrites**: (exact stats allow very aggressive rewrites)
  - **Recompute Memory Estimates**: (w/ unconditional scope of single DAG)
  - **Generate Runtime Instructions**: (construct LOPs / instructions)
Inter-Procedural Analysis – Motivation

- **Challenges**
  - Multiple function calls with different inputs
  - Conditional control flow
  - Complex function call graphs (incl recursion)

- **Example (multiple calls w/ different inputs)**

```plaintext
X = read($X1)  1M x 1k
X = foo(X);  1M x 2
if( $X2 != " " ) {
  X2 = read($X2);
  X2 = foo(X2);
  X = cbind(X, X2);
} ...
```

**foo** = function (Matrix[Double] A) return (Matrix[Double] B)

- $B = A - \text{colSums}(A)$
- if( sum($B != B > 0$) )
  - print("NaNs encountered.");

Size propagation into foo() would be incorrect!
Inter-Procedural Analysis (2)

- **Collect IPA Function Candidates**
  - Functions called once
  - Functions called with consistent sizes (dims/nnz)
  - Unary size-preserving functions

- **Size Propagation (via dynamic recompilation)**
  - Inter- and intra-procedural size propagation (in execution order)
  - Control-flow-aware propagation and reconciliation

- **Additional IPA Passes**
  - Remove unused functions
  - Flag functions
    - “recompile once”
  - Remove constant binary operations

```r
foo = function (Matrix[Double] A)
  return (Matrix[Double] C)
  {
    recompile once on entry w/ A
    B = rand(nrow(A),1);
    while(...)
      C = A / rowSums(A) * B
  }
A = matrix(1, nrow(X), ncol(X));
while(...)
  ...*A...
```
Hands-On Labs: Rewrites and Handling of Size Information

- **Exercise 1: Sum-Product Rewrite:** `sum(A %*% t(B))`
  - a) What’s happening for A:=[900x1000], B:=[700,1000]
  - b) What’s happening for A:=[900x1], B:=[700x1]

- **Exercise 2: Matrix Multiplication Chains:** A %*% B %*% C %*% D %*% E
  - What’s happening as we change dimensions of A, B, C, D, E (start with dimensions given on slide 17)

- **Exercise 3: Dynamic Recompilation**
  - What’s happening during compilation/runtime to gather size information
    ```r
    if( $1 == 1 ) {
      Y = rand(rows=nrow(X), cols=1, min=1, max=maxval);
      X = cbind(X, table(seq(1,nrow(Y)),Y));
    }
    print(sum(X));
    ```
From SystemR to SystemML – A Comparison

**Similarities**
- **Declarative specification** (fixed semantics): SQL vs DML
- **Simplification rewrites** (Starburst QGM rewrites vs static/dynamic rewrites)
- **Operator selection** (physical operators for join vs matrix multiply)
- **Operator reordering** (join enumeration vs matrix multiplication chain opt)
- **Adaptive query processing** (progressive reop vs dynamic recompile)
- **Physical layout** (NSM/DSM/PAX page layouts vs dense/sparse block formats)
- **Buffer pool** (pull-based page cache vs anti-caching of in-memory variables)
- **Advanced optimizations** (source code gen, compression, GPUs, etc)
- **Cost model / stats** (est. time for IO/compute/latency; histograms vs dims/nnz)

**Differences**
- **Algebra** (relational algebra vs linear algebra)
- **Programs** (query trees vs DAGs, conditional control flow, often iterative)
- **Optimizations** (algebra-specific semantics, rewrites, and constraints)
- **Scale** (10s-100s vs 10s-10,000s of operators)
- **Data preparation** (ETL vs feature engineering)
- **Physical design, transactions processing, multi-tenancy**, etc
SystemML is Open Source:
Apache Incubator Project since 11/2015
Website: http://systemml.apache.org/
Sources: https://github.com/apache/incubator-systemml