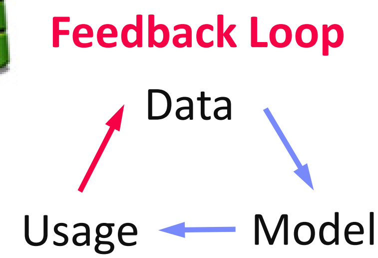
Efficient Data-Parallel Cumulative Aggregates for Large-Scale Machine Learning

Matthias Boehm[1], Alexandre V. Evfimievski[2], Berthold Reinwald[2]

[1] Graz University of Technology; Graz, Austria
[2] IBM Research – Almaden; San Jose, CA, USA

SCIENCE
PASSION
TECHNOLOGY

Motivation Large-Scale ML



- **Large-Scale Machine Learning**
  - **Variety of ML applications** (supervised, semi-/unsupervised)
  - **Large data collection** (labels: feedback, weak supervision)
- **State-of-the-art ML Systems**
  - Batch algorithms → **Data-/task-parallel operations**
  - Mini-batch algorithms → Parameter server

Apache SystemML™

- **Data-Parallel Distributed Operations**
  - Linear Algebra (matrix multiplication, element-wise operations, structural and grouping aggregations, statistical functions)
  - Meta learning (e.g., cross validation, ensembles, hyper-parameters)
  - **In practice:** also reorganizations and **cumulative aggregates**

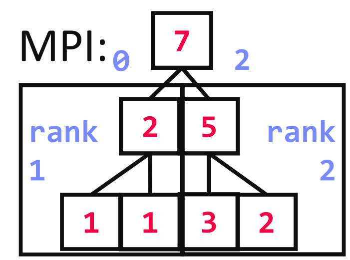
Motivation Cumulative Aggregates

- **Example Prefix Sums** `Z = cumsum(X)` with $Z_{ij} = \sum_{k=1}^{i} X_{kj} = X_{ij} + Z_{(i-1)j}$

$$\begin{bmatrix}1 & 2\\1 & 1\\3 & 4\\2 & 1\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2\\2 & 3\\5 & 7\\7 & 8\end{bmatrix}$$

- **Applications**
  - **#1 Iterative survival analysis:** Cox Regression / Kaplan-Meier
  - **#2 Spatial data processing** via linear algebra, cumulative histograms
  - **#3 Data preprocessing:** subsampling of rows / remove empty rows

- **Parallelization**
  - Recursive formulation looks inherently sequential
  - Classic example for **parallelization via aggregation trees** (message passing or shared memory HPC systems)



- **Question: Efficient, Data-Parallel Cumulative Aggregates?** (blocked matrices as **unordered collections** in Spark or Flink)

Outline

- **SystemML Overview and Related Work**
- **Data-Parallel Cumulative Aggregates**
- **System Integration**
- **Experimental Results**

SystemML Overview and Related Work

High-Level SystemML Architecture

DML Scripts

APIs: Command line, JMLC, Spark MLContext, Spark ML, (20+ scalable algorithms)

Language

Compiler

Runtime

DML (**D**eclarative **M**achine **L**earning **L**anguage)

Apache SystemML™

05/2017 Apache Top-Level Project
11/2015 Apache Incubator Project
08/2015 Open Source Release

[SIGMOD'15,'17,'19]
[PVLDB'14,'16a,'16b,'18]
[ICDE'11,'12,'15]
[CIDR'17]
[VLDBJ'18]
[DEBull'14]
[PPoPP'15]

In-Memory Single Node
(scale-up)

Hadoop or Spark Cluster
(scale-out)

In-Progress:

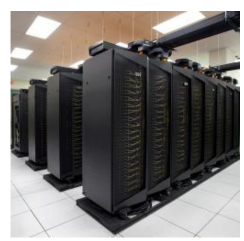
GPU

since 2014/16

Java

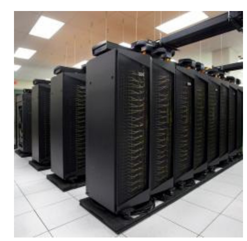
since 2012

hadoop



since 2010/11

APACHE Spark



since 2015

Basic HOP and LOP DAG Compilation

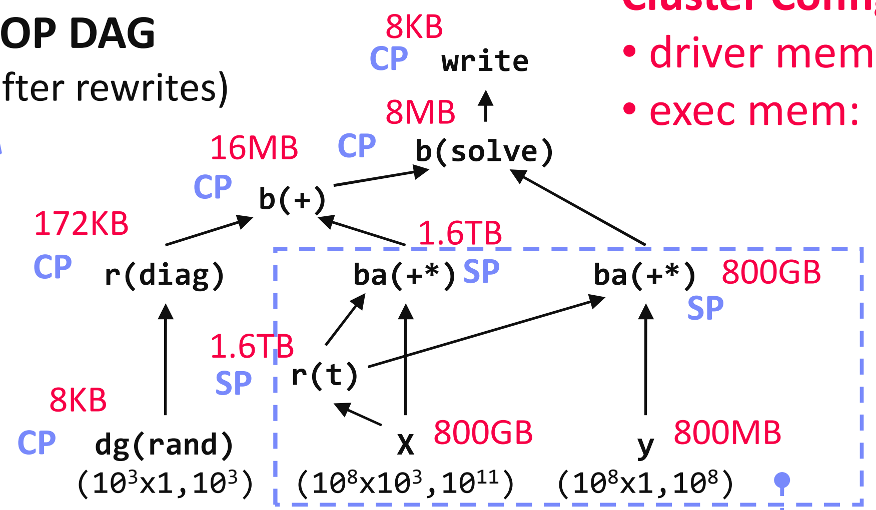
LinregDS (Direct Solve)

```
X = read($1);
y = 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);
```

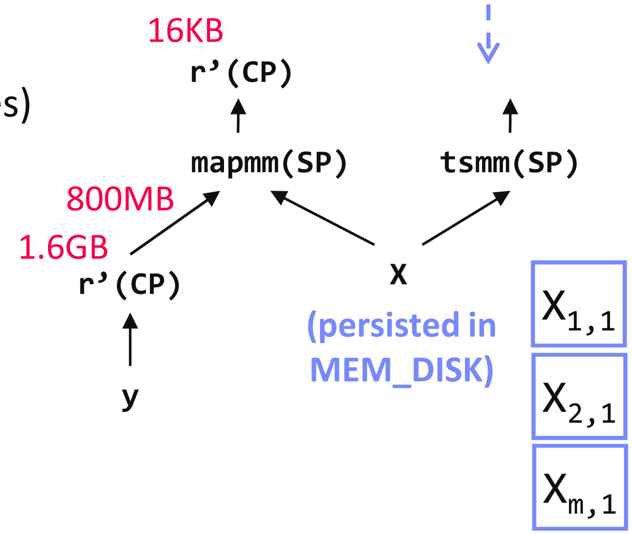
Scenario:
X: 10^8 x 10^3, 10^{11}
y: 10^8 x 1, 10^8

Cluster Config:
- driver mem: 20 GB
- exec mem: 60 GB

HOP DAG
(after rewrites)



LOP DAG
(after rewrites)



➔ **Hybrid Runtime Plans:**
- **Size propagation / memory estimates**
- **Integrated CP / Spark runtime**

➔ **Distributed Matrices**
- **Fixed-size (squared) matrix blocks**
- **Data-parallel operations**

Cumulative Aggregates in ML Systems

(Straw-man Scripts and Built-in Support)

```
cumsumN2 = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; csums = matrix(0,1,ncol(A));
  for( i in 1:nrow(A) ) {
    csums = csums + A[i,];
    B[i,] = csums;
  }
}
```

copy-on-write → O(n^2)

```
cumsumNlogN = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; m = nrow(A); k = 1;
  while( k < m ) {
    B[(k+1):m,] = B[(k+1):m,] + B[1:(m-k),];
    k = 2 * k;
  }
}
```

→ O(n log n)

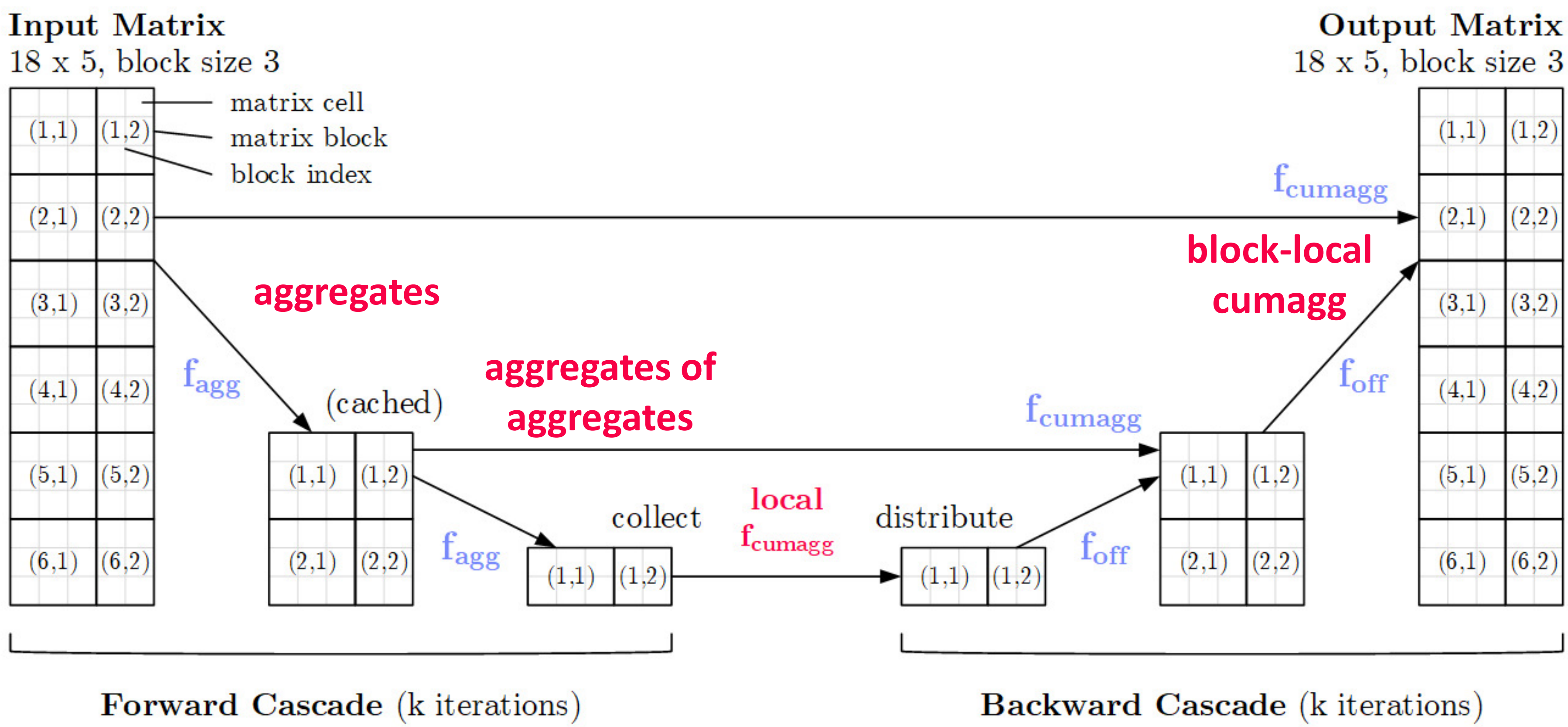
→ Qualify for update in-place, but still too slow

- **ML Systems**
  - Update in-place: R (ref count), SystemML (rewrites), Julia
  - Builtins in R, Matlab, Julia, NumPy, SystemML (since 2014)
    `cumsum()`, `cummin()`, `cummax()`, `cumprod()`
- **SQL**
  - `SELECT Rid, V, sum(V) OVER(ORDER BY Rid) AS cumsum FROM X`
  - Sequential and parallelized execution (e.g., [Leis et al, PVLDB'15])

Data-Parallel Cumulative Aggregates

DistCumAgg Framework

- **Basic Idea:** self-similar operator chain (forward, local, backward)

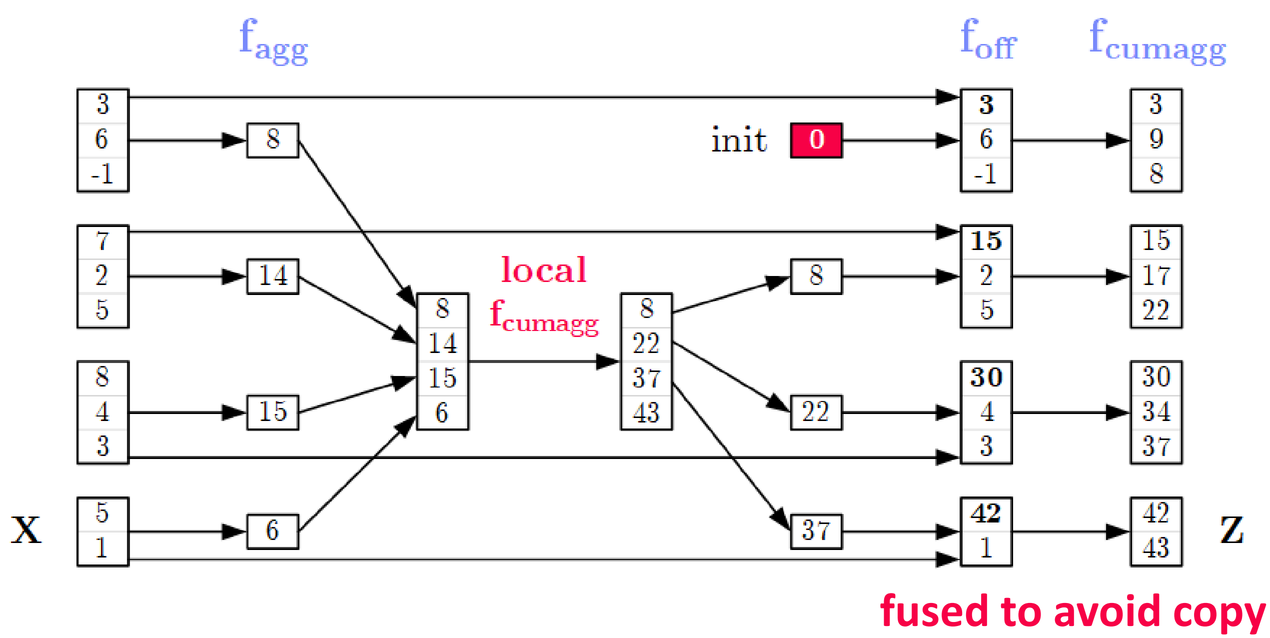


Basic Cumulative Aggregates

- **Instantiating Basic Cumulative Aggregates**

Operation	Init	f_{agg}	f_{off}	f_{cumagg}
cumsum(X)	0	colSums(B)	$B_{1:}$=$B_{1:}$+a	cumsum(B)
cummin(X)	∞	colMins(B)	$B_{1:}$=min($B_{1:}$,a)	cummin(B)
cummax(X)	-∞	colMaxs(B)	$B_{1:}$=max($B_{1:}$,a)	cummax(B)
cumprod(X)	1	colProds(B)	$B_{1:}$=$B_{1:}$*a	cumprod(B)

- **Example** `cumsum(X)`



Matthias Boehm, Alexandre V. Evfimievski, and Berthold Reinwald:
Efficient Data-Parallel Cumulative Aggregates for Large-Scale Machine Learning, **BTW 2019**

Complex Cumulative Aggregates

- **Instantiating Complex Recurrences Equations**

`Z = cumsumprod(X) = cumsumprod(Y, W)`
with $Z_i = Y_i + W_i * Z_{i-1}$, $Z_0 = 0$

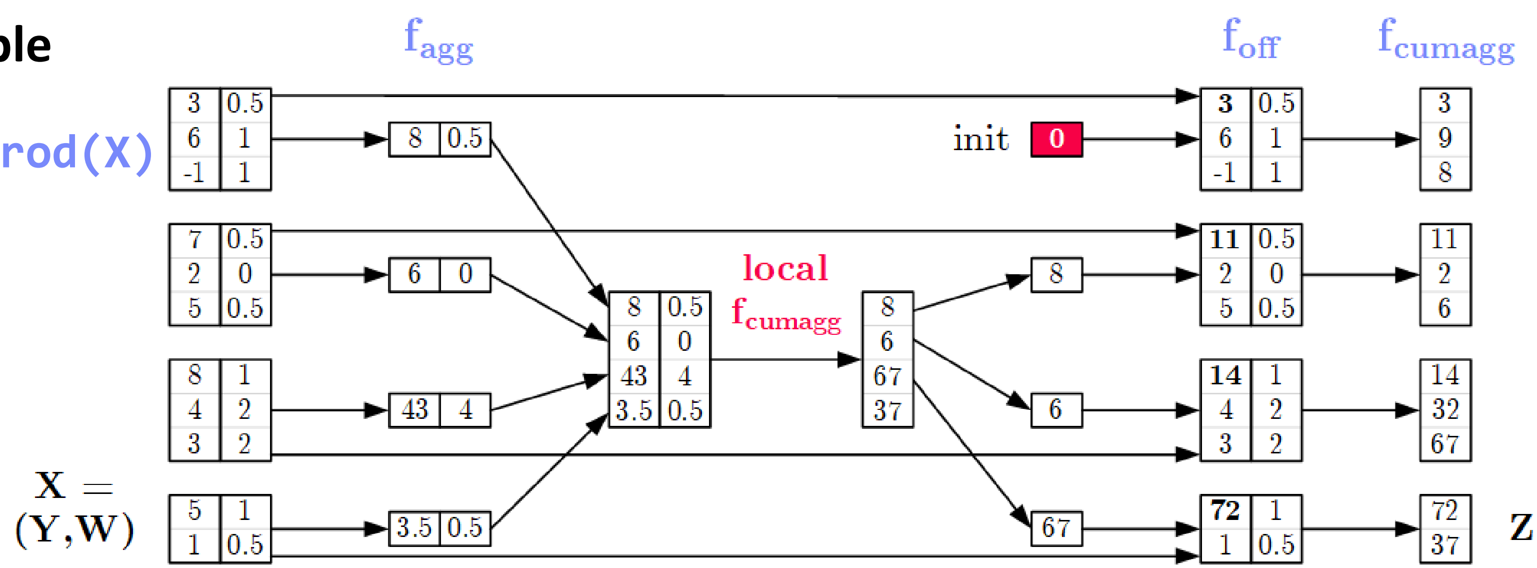
1	.2
1	.1
3	.0
2	.1

Exponential smoothing

Init	f_{agg}	f_{off}	f_{cumagg}
0	cbind(cumsumprod(B)$_{n1}$, prod(B$_{:2}$))	$B_{11}=B_{11}+B_{12}*a$	cumsumprod(B)

- **Example**

cumsumprod(X)

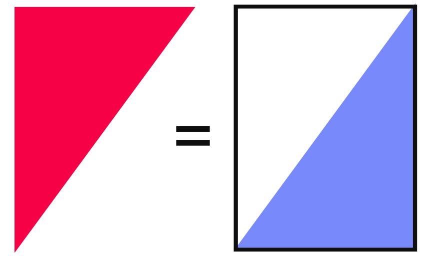


Matthias Boehm, Alexandre V. Evfimievski, and Berthold Reinwald:
Efficient Data-Parallel Cumulative Aggregates for Large-Scale Machine Learning, **BTW 2019**

System Integration

Simplification Rewrites

- **Example #1: Suffix Sums**
  - **Problem:** Distributed reverse causes data shuffling
  - Compute via column aggregates and prefix sums




```
rev(cumsum(rev(X))) → X + colSums(X) – cumsum(X)
                          (broadcast)   (partitioning-preserving)
```

- **Example #2: Extract Lower Triangular**
  - **Problem:** Indexing cumbersome/slow; cumsum densifying
  - Use dedicated operators

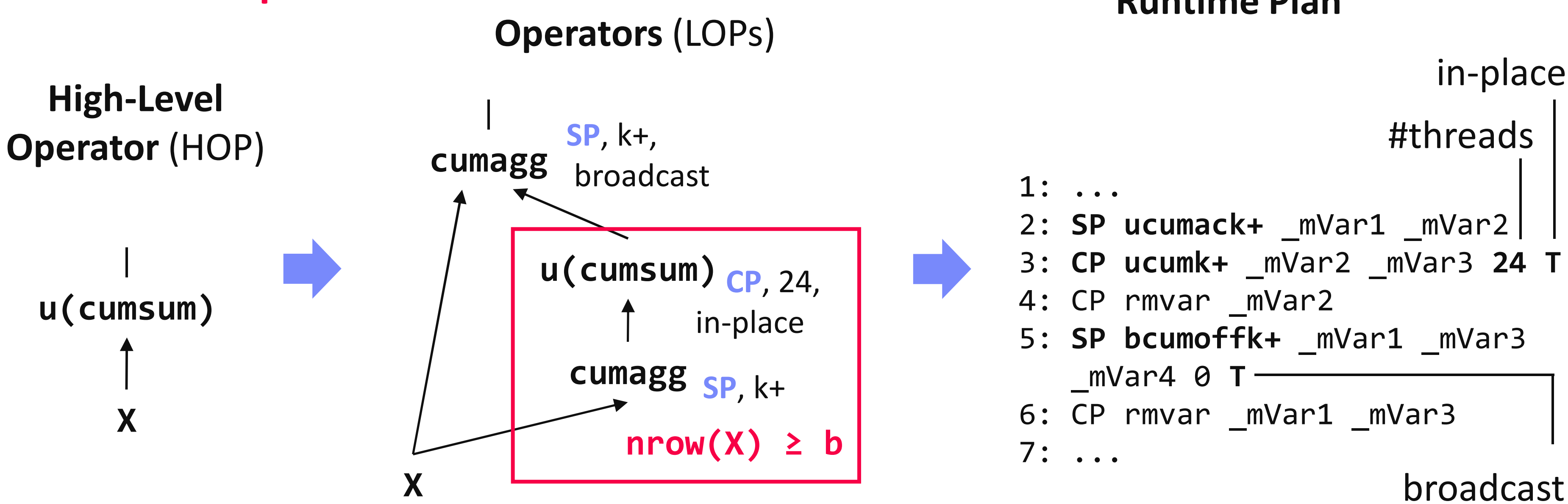
```
X * cumsum(diag(matrix(1,nrow(X),1)))
  → lower.tri(X)
```

```
1 0 0 0 0 0 0
1 1 0 0 0 0 0
1 1 1 0 0 0 0
1 1 1 1 0 0 0
1 1 1 1 1 0 0
1 1 1 1 1 1 0
1 1 1 1 1 1 1
```

Execution Plan Generation

- **Compilation Chain of Cumulative Aggregates**
  - Execution type selection based on memory estimates
  - Physical operator config (broadcast, aggregation, in-place, #threads)

- **Example**



Runtime Operators

- **CP cumagg Operator:**
  - Local in-memory operator w/ copy-on-write or in-place
  - Multi-threading via static range partitioning

- **Spark Partial Cumulative Aggregate:**
  - Data-local block aggregation f_{agg} into row of column aggregates
  - Insert row into position of empty target block (sparse)
  - Global merge of partial blocks

- **Spark Cumulative Offset**
  - Join data and offsets (broadcast, co-partition, re-partition)
  - Applies the offsets f_{off} and performs block-local f_{cumagg} w/ zero-copy offset aggregation

Experimental Results

Experimental Setting

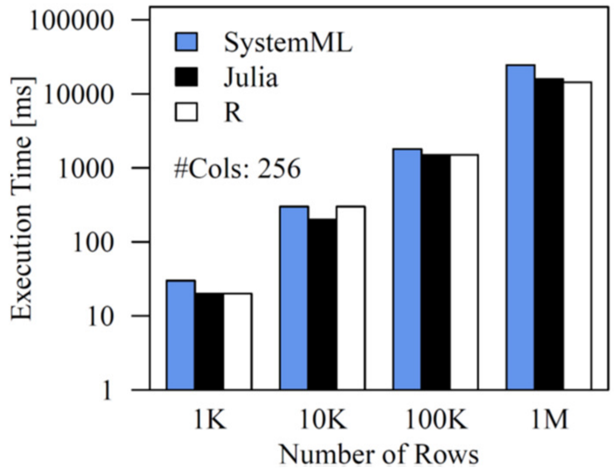
- **Cluster Setup**
  - 2+10 node cluster, 2x Intel Xeon E5-2620, 24 vcores, 128GB RAM
  - 1Gb Ethernet, CentOS 7.2, OpenJDK 1.8, Haddop 2.7.3, Spark 2.3.1
  - Yarn client mode, 40GB driver, 10 executors (19 cores, 60GB mem)
  - Aggregate memory: 10 * 60GB * [0.5,0.6] = [300GB, 360GB]

- **Baselines and Data**
  - Local: SystemML 1.2++,
    Julia 0.7 (08/2018), R 3.5 (04/2018)
  - Distributed: SystemML 1.2++,
    C-based MPI impl. (OpenMPI 3.1.3)
  - Double precision (**FP64**) synthetic data

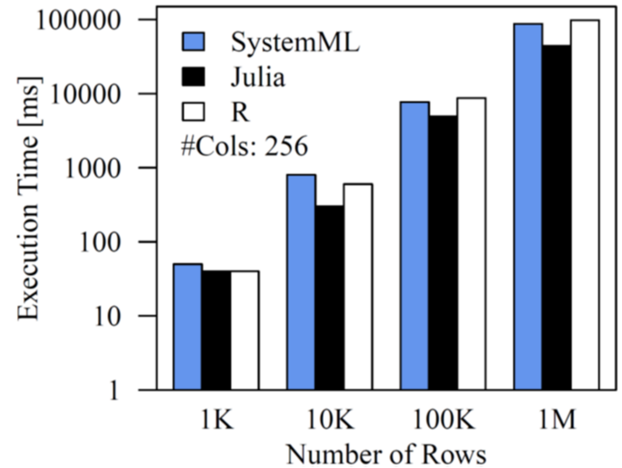
Local Baseline Comparisons

- **Strawmen Scripts** (w/ inplace)

cumsumN2

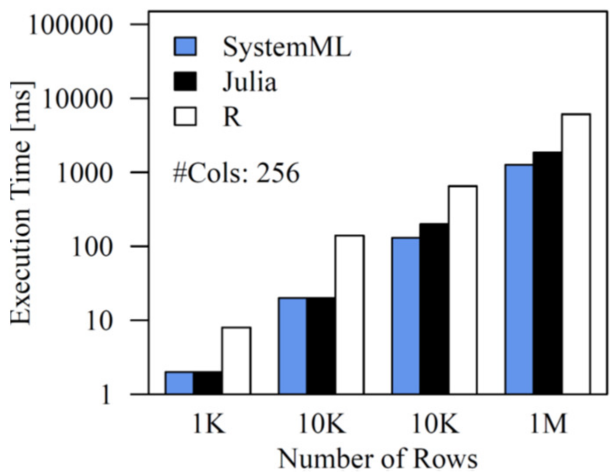


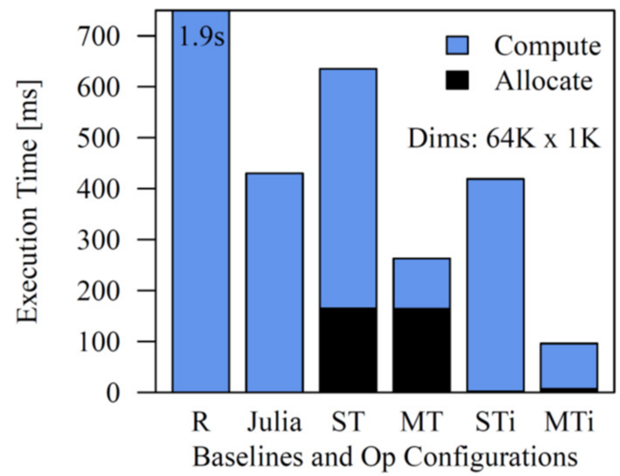
cumsumNlogN



- **Built-in cumsum**

competitive single-node performance

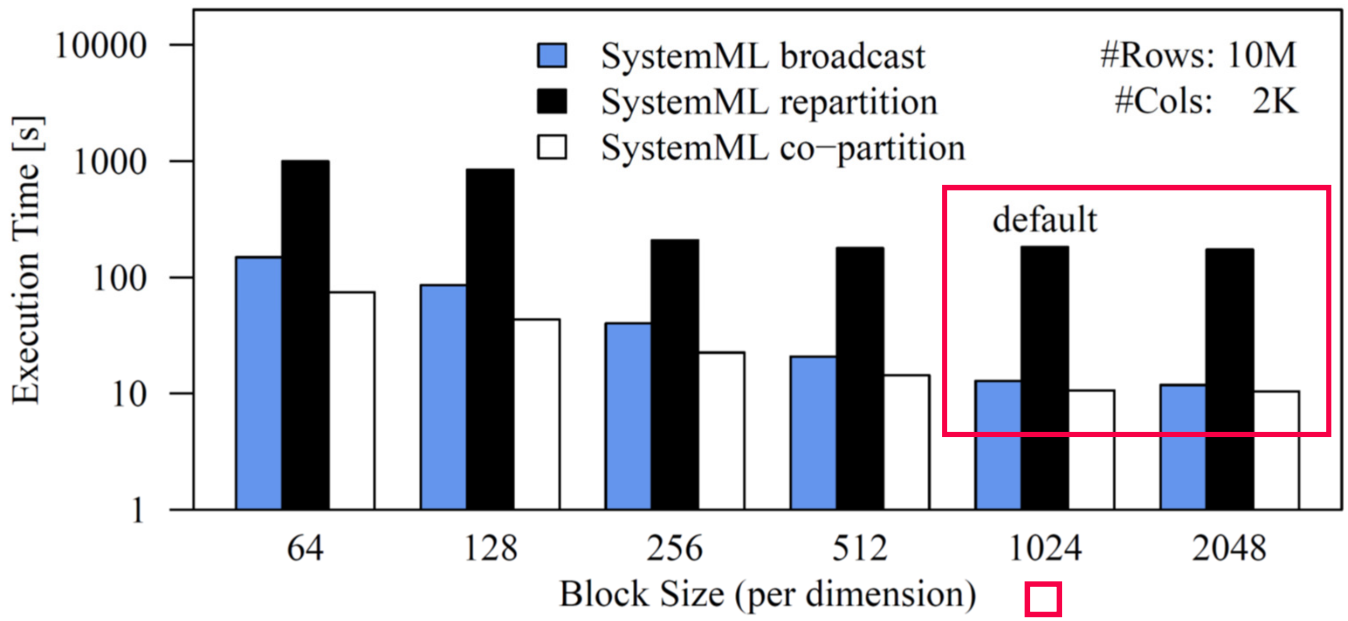




Broadcasting and Blocksizes

- **Setup:** Mean runtime of rep=100 `print(min(cumsum(X)))`, including I/O and Spark context creation (~15s) once

- **Results**



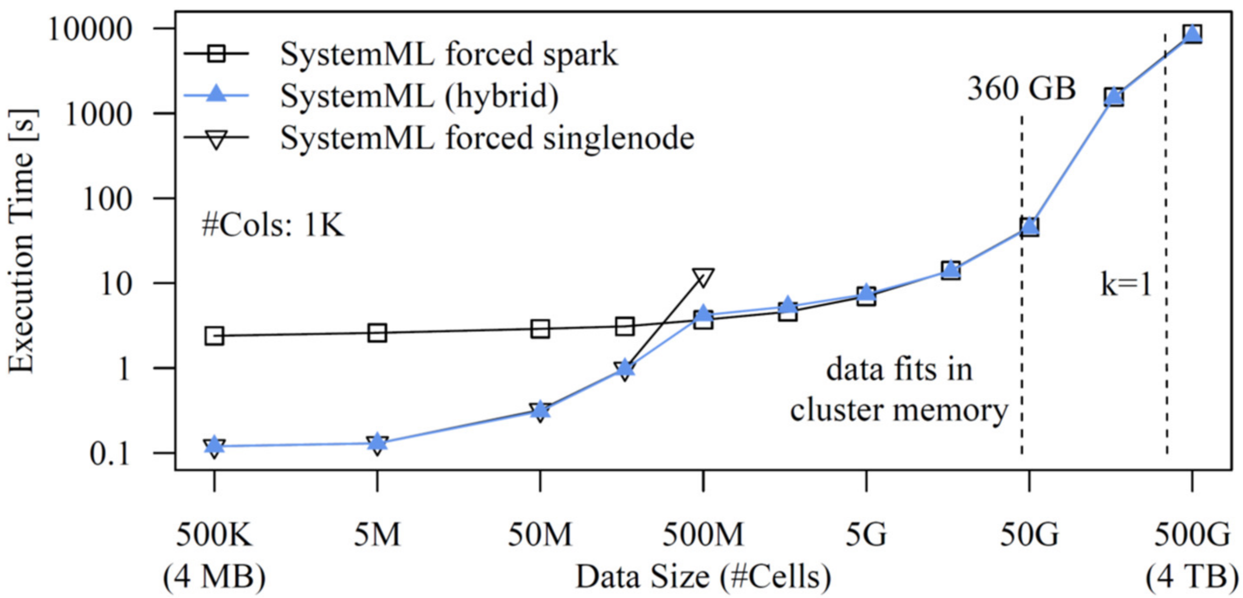
160GB

19.6x (17.3x @ default)

1K good compromise
(8MB, block overheads)

Scalability (from 4MB to 4TB)

- **Setup:** Mean runtime of rep=10 **`print(min(cumsum(X)))`**



#Cells	System ML	MPI
165M	0.97s	0.14s
500M	4.2s	0.26s
1.65G	5.3s	0.61s
5G	7.4s	1.96s
15.5G	13.9s	6.20s
50G	44.8s	19.8s
165G	1,531s	N/A
500G	8,291s	N/A

- **In the Paper**
  - Characterization of applicable operations; other operations: `cumsum` in `removeEmpty`
  - More baselines comparisons; weak and strong scaling results

Conclusions

- **Summary**
  - **DistCumAgg:** Efficient, data-parallel cumulative aggregates (**self-similar**)
  - End-to-end compiler and runtime integration in SystemML
  - Physical operators for hybrid (local/distribute) plans
- **Conclusions**
  - Practical ML systems need support for a **broad spectrum of operations**
  - **Efficient parallelization** of presumably sequential operations over blocked matrix representations on top frameworks like Spark or Flink
- **Future Work**
  - Integration with automatic sum-product rewrites
  - Operators for HW accelerators (dense and sparse)
  - Application to parallel **time series analysis / forecasting**