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

Matthias Boehm¹, Alexandre V. Evfimievski², Berthold Reinwald²

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

- **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
  
  \[
  Z = \text{cumsum}(X)
  \]
  
  with
  \[
  Z_{ij} = \sum_{k=1}^{i} X_{kj} = X_{ij} + Z_{(i-1)j}
  \]

- **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

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

- DML Scripts
  - Language
  - Compiler
  - Runtime

- **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
- Spark
  - since 2015

**SystemML Overview and Related Work**

- 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] [VLDB’18] [DEBull’14] [PPoPP’15]
Basic HOP and LOP DAG Compilation

**LinregDS (Direct Solve)**

```r
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);
```

**HOP DAG** (after rewrites)

- 8KB CP write
- 8MB write
- 16MB CP b(+) 1.6TB CP b(solve)
- 172KB CP r(diag)
- 1.6TB CP ba(+)* SP ba(+) SP
- 8KB CP dg(rand) (10^3 x 1, 10^3)
- 1.6TB SP r(t) (10^8 x 10^3, 10^{11})
- 800GB SP y (10^8 x 1, 10^8)
- 800GB CP X
- 16KB CP r'(CP)

**LOP DAG** (after rewrites)

- 800MB SP
- 1.6GB CP r'(CP)
- 800MB SP

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

**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)

ML Systems
- Update in-place: R (ref count), SystemML (rewrites), Julia
- Builtins in R, Matlab, Julia, NumPy, SystemML (since 2014)
  - \texttt{cumsum()}, \texttt{cummin()}, \texttt{cummax()}, \texttt{cumprod()}

SQL
- \texttt{SELECT Rid, V, sum(V) OVER(ORDER BY Rid) AS cumsum FROM X}
- Sequential and parallelized execution (e.g., [Leis et al, PVLDB’15])

\begin{align*}
\texttt{cumsumN2} &= \text{function}(\text{Matrix}[\text{Double}] A) \\
2: \quad &\text{return}(\text{Matrix}[\text{Double}] B) \\
3: \quad &\{ \\
4: \quad &\quad B = A; \text{csums} = \text{matrix}(0,1, \text{ncol}(A)); \\
5: \quad &\quad \text{for}( i \in 1: \text{nrow}(A) ) \{ \\
6: \quad &\quad \quad \text{csums} = \text{csums} + A[i,]; \\
7: \quad &\quad \quad \text{B}[i,] = \text{csums}; \\
8: \quad &\quad \} \\
9: \quad \text{copy-on-write } \rightarrow \text{O}(n^2)
\end{align*}

\begin{align*}
\texttt{cumsumNlogN} &= \text{function}(\text{Matrix}[\text{Double}] A) \\
2: \quad &\text{return}(\text{Matrix}[\text{Double}] B) \\
3: \quad &\{ \\
4: \quad &\quad B = A; m = \text{nrow}(A); k = 1; \\
5: \quad &\quad \text{while}( k < m ) \{ \\
6: \quad &\quad \quad B[(k+1):m,] = B[(k+1):m,] + B[1:(m-k),]; \\
7: \quad &\quad \quad k = 2 * k; \\
8: \quad &\quad \} \\
9: \quad \} \\
\rightarrow \text{O}(n \log n)
\end{align*}

\Rightarrow \text{Qualify for update in-place, but still too slow}
Data-Parallel Cumulative Aggregates
DistCumAgg Framework

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

#### Instantiating Basic Cumulative Aggregates

<table>
<thead>
<tr>
<th>Operation</th>
<th>Init</th>
<th>( f_{agg} )</th>
<th>( f_{off} )</th>
<th>( f_{cumagg} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cumsum}(X) )</td>
<td>0</td>
<td>( \text{colSums}(B) )</td>
<td>( B_1 := B_1 + a )</td>
<td>( \text{cumsum}(B) )</td>
</tr>
<tr>
<td>( \text{cummin}(X) )</td>
<td>(-\infty)</td>
<td>( \text{colMins}(B) )</td>
<td>( B_1 := \min(B_1, a) )</td>
<td>( \text{cummin}(B) )</td>
</tr>
<tr>
<td>( \text{cummax}(X) )</td>
<td>(-\infty)</td>
<td>( \text{colMaxs}(B) )</td>
<td>( B_1 := \max(B_1, a) )</td>
<td>( \text{cummax}(B) )</td>
</tr>
<tr>
<td>( \text{cumprod}(X) )</td>
<td>1</td>
<td>( \text{colProds}(B) )</td>
<td>( B_1 := B_1 \times a )</td>
<td>( \text{cumprod}(B) )</td>
</tr>
</tbody>
</table>

#### Example \( \text{cumsum}(X) \)

The diagram illustrates the accumulation of sums for different values. The operations are updated locally and cumulatively. Fused operations are used to avoid data copies.

Mathias 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**
  
  \[ Z = \text{cumsumprod}(X) = \text{cumsumprod}(Y, W) \]
  with \( Z_i = Y_i + W_i \times Z_{i-1}, \ Z_0 = 0 \)

- **Example**

  \[
  \begin{array}{|c|c|c|c|}
  \hline
  \text{Init} & f_{agg} & f_{off} & f_{cumagg} \\
  \hline
  \emptyset & \text{cbind(cumsumprod}(B)_{n1}, \prod(B_{:2})) & B_{11} = B_{11} + B_{12} \times a & \text{cumsumprod}(B) \\
  \hline
  \end{array}
  \]

  \[
  X = (Y, W)
  \]

  \[
  Z
  \]

  Exponential smoothing

  \[
  1.2 \\
  1.1 \\
  3.0 \\
  2.1
  \]
System Integration
Simplification Rewrites

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

\[
\text{rev}(\text{cumsum}(\text{rev}(X))) \rightarrow X + \text{colSums}(X) - \text{cumsum}(X) \\
\text{(broadcast)} \quad \text{(partitioning-preserving)}
\]

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

\[
X \ast \text{cumsum} (\text{diag} (\text{matrix}(1, \text{nrow}(X), 1))) \rightarrow \text{lower.tri}(X)
\]
Execution Plan Generation

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

- **Example**

```
High-Level Operator (HOP)  |  Low-Level Operators (LOPs)  |  Runtime Plan
```

```
u(cumsum)  \rightarrow \text{cumagg} \rightarrow \text{cumagg} \rightarrow \text{ucumagg+} \rightarrow \text{ucumk+} \rightarrow \text{rmvar} \rightarrow \text{bcumoff}
```

```
in-place  
#threads
```

```
1: ...  
2: SP ucumack+ _mVar1 _mVar2  
3: CP ucumk+ _mVar2 _mVar3 24 T  
4: CP rmvar _mVar2  
5: SP bcumoffk+ _mVar1 _mVar3 _mVar4 0 T  
6: CP rmvar _mVar1 _mVar3  
7: ...
```

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

- **Built-in cumsum**

  competitive single-node performance
Broadcasting and Blocksizes

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

- **Results**

  ![Graph showing execution time vs block size](graph)

  - 160GB
  - 19.6x (17.3x @ default)
  - 1K good compromise (8MB, block overheads)
### Scalability (from 4MB to 4TB)

- **Setup:** Mean runtime of \( \text{rep}=10 \) \( \text{print}(\text{min}(\text{cumsum}(X))) \)

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

- **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*