Compressed Linear Algebra for Large-Scale Machine Learning

Ahmed Elgohary[2], Matthias Boehm[1], Peter J. Haas[1], Frederick R. Reiss[1], Berthold Reinwald[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] University of Maryland; College Park, MD, USA

Contact: Matthias Boehm
mboehm@us.ibm.com

VLDB 2016
Session R3

Motivation

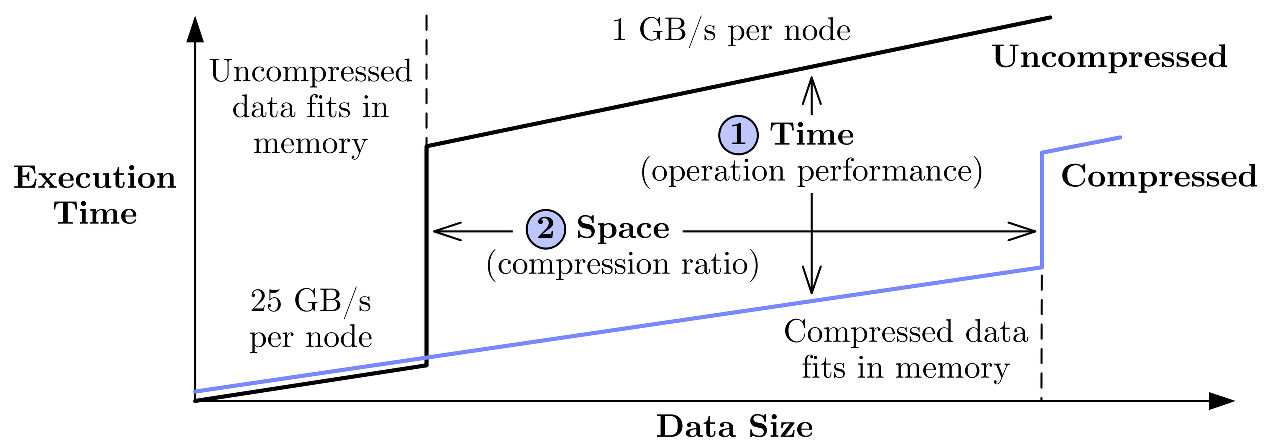
Problem Description

- **Problem of memory-centric performance**
  - Iterative ML algorithms with read-only data access
  - Bottleneck: I/O-bound matrix-vector multiplications

  ➔ **Crucial to fit matrix into memory** (single node, distributed, GPU)
- **Goal:** Improve performance of declarative ML algorithms via **lossless compression**
- **Baseline solution**
  - Employ general-purpose compression techniques
  - Decompress matrix block-wise for each operation
  - Heavyweight (e.g., Gzip): decompression too slow
  - Leightweight (e.g., Snappy): modest compression ratio

CLA: Compressed Linear Algebra

- **Key idea**
  - Use lightweight database compression techniques
  - Perform LA operations on compressed matrices
- **Goals of CLA**
  - Operations performance close to uncompressed
  - Good compression ratios



- **Distributed matrices:** block matrix ➔ CLA integration

Workload Characteristics

LinregCG (Conjugate Gradient)

```
X = read($1); # n x m matrix
y = read($2); # n x 1 vector
maxi = 50; lambda = 0.001;
intercept = $3;
...
r = -(t(X) %*% y);
norm_r2 = sum(r * r); p = -r;
w = matrix(0, ncol(X), 1); i = 0;
while(i<maxi & norm_r2>norm_r2_trgt) {
   q = (t(X) %*% (X %*% p)) + lambda * p;
   alpha = norm_r2 / sum(p * q);
   w = w + alpha * p;
   old_norm_r2 = norm_r2;
   r = r + alpha * q;
   norm_r2 = sum(r * r);
   beta = norm_r2 / old_norm_r2;
   p = -r + beta * p; i = i + 1;
}
write(w, $4, format="text");
```

Xv

v^TX

$X^T(w*(Xv))$

X^TX

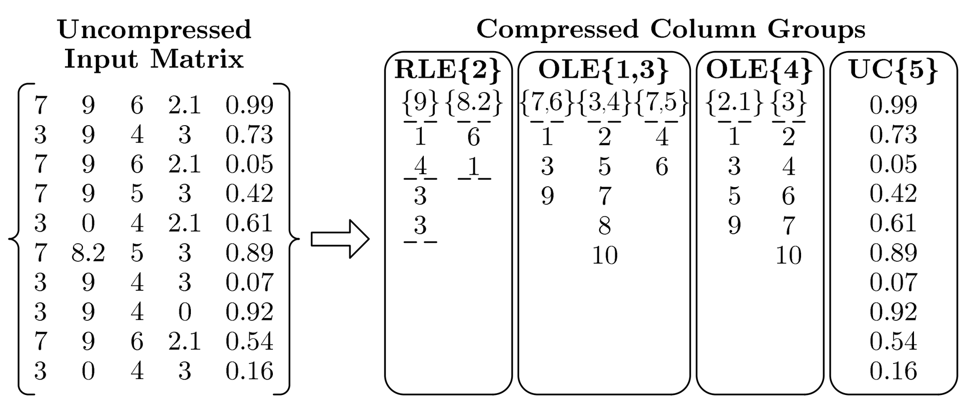
…

- **Common data characteristics**
  - Tall & skinny; non-uniform sparsity
  - Low column card.; column correlations

Compression Schemes

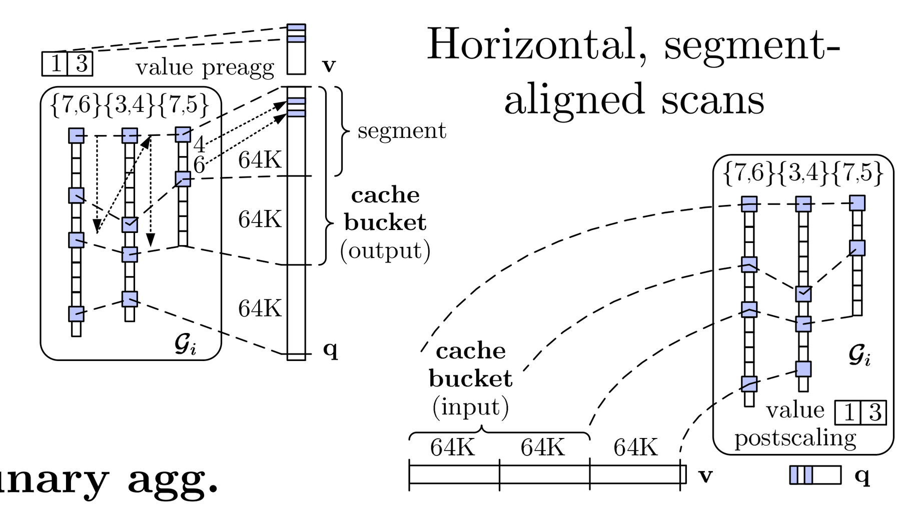
Matrix Compression Framework

- **Overview compression framework**
  - Column-wise matrix compression (values and compressed offset lists)
  - Column co-coding (column groups, encoded as single unit)
  - Heterogeneous column encoding formats
- **Column encoding formats**
  - Offset-List (OLE)
  - Run-Length (RLE)
  - Uncompressed Columns (UC)
- **Compression planning**
  - Selects column groups and encoding formats per group (data dependent)



Operations over Compressed Matrix Blocks

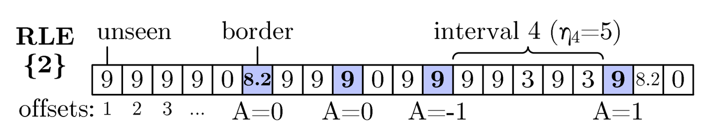
- **Matrix-vector multiplication**
  - Naïve: cache unfriendly on output
  - Cache-conscious scheme
- **Vector-matrix multiplication**
  - Naïve: cache unfriendly on input
  - Cache-conscious scheme
- **Other operations**
  - tsmm, mmchain, append, **scalar, unary agg.**



Compression Planning

Compressed Size Estimation

- **Goals and general principles**
  - Low planning costs ➔ Sampling-based techniques
  - Conservative approach ➔ Prefer underestimating S^{UC}/S^{C} and corrections
- **Estimating compressed size: $S^C = \min(S^{OLE}, S^{RLE})$**
  - # of distinct tuples d_i: Hybrid generalized jackknife estimator [JASA'98]
  - # of OLE segments b_{ij}: Expected value under maximum-entropy model
  - # of non-zero tuples z_i: Scale from sample with coverage adjustment
  - # of runs r_{ij}: maxEnt model + independent-interval approx.



Compression

- **Column group partitioning**
  - Exhaustive grouping: $O(m^m)$
  - Brute-force greedy grouping: $O(m^3)$
    - Start with singleton groups
    - Merge groups with max ratio

  ➔ **Bin-packing-based grouping**
- **Compression algorithm**
  - Transpose input **X**
  - Draw random sample of rows ***S***
  - Classify, group, compress

Algorithm 2 Matrix Block Compression

Input: Matrix block $\mathbf{X}$ of size $n \times m$
Output: A set of compressed column groups $\mathcal{X}$

```
C^C ← ∅, C^UC ← ∅, G ← ∅, X ← ∅
// Planning phase
S ← sampleRowsUniform(X, sample_size)
for all column k in X do                          // classify
   cmp_ratio ← ẑ_i α / min(Ŝ_k^RLE, Ŝ_k^OLE)
   if cmp_ratio > 1 then
      C^C ← C^C ∪ k
   else
      C^UC ← C^UC ∪ k
bins ← runBinPacking(C^C)                         // group
for all bin b in bins do
   G ← G ∪ groupBruteForce(b)
// Compression phase
for all column group G_i in G do                  // compress
   do
      biglist ← extractBigList(X, G_i)
      cmp_ratio ← getExactCmpRatio(biglist)
      if cmp_ratio > 1 then
         X ← X ∪ compressBigList(biglist), break
      k ← removeLargestColumn(G_i)
      C^UC ← C^UC ∪ k
   while |G_i| > 0
return X ← X ∪ createUCGroup(C^UC)
```

Experiments

Experimental Setting

- **Cluster setup**
  - 1 head node: 2x4 Intel E5530, 64 GB RAM,
  - 6 worker nodes: 2x6 Intel E5-2440, 96 GB RAM,
  - Spark 1.4, 6 exec. (24 cores, 60GB), 25GB driver
- **Selected baselines**
  - Apache SystemML 0.9, uncompressed LA (**ULA**)
  - General-purpose compression with ULA (**Gzip, Snappy**)

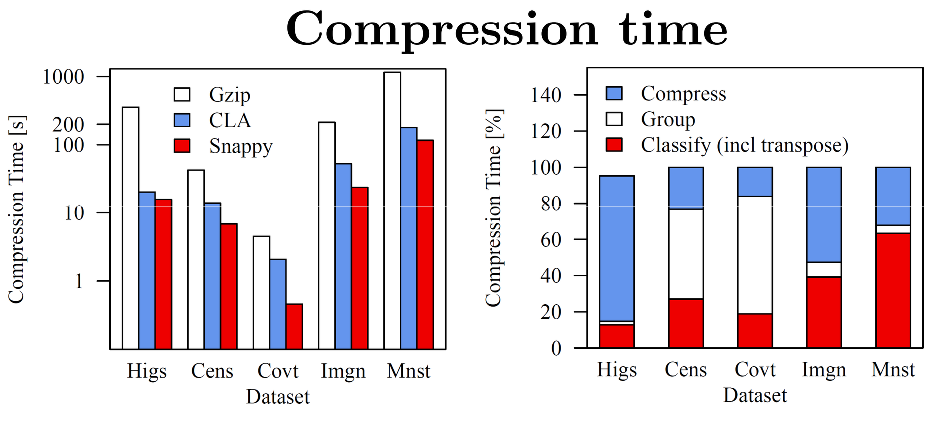
Compression Ratios

Dataset	Dims	Sparsity	Size	Gzip	Snappy	CLA
Higgs	11M x 28	0.92	2.5GB	1.93	1.38	**2.03**
Census	2.5M x 68	0.43	1.3GB	17.11	6.04	**27.46**
Covtype	600K x 54	0.22	0.14GB	10.40	6.13	**12.73**
ImageNet	1.2M x 900	0.31	4.4GB	5.54	3.35	**7.38**
Mnist8m	8.1M x 784	0.25	019GB	4.12	2.60	**6.14**

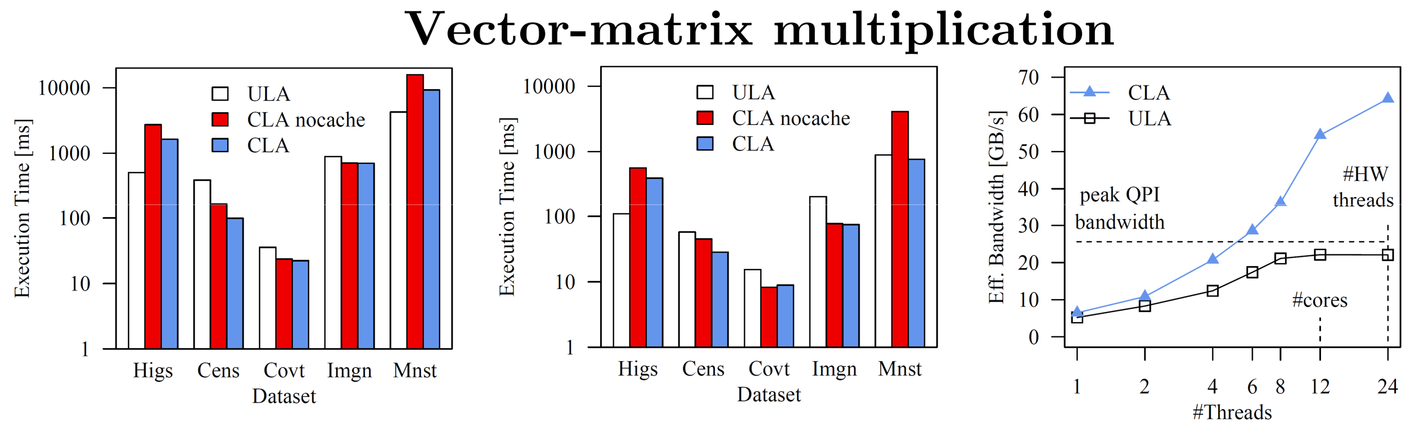
Micro-Benchmarks

SYSTEMML-449 Compressed Linear Algebra

Compression time

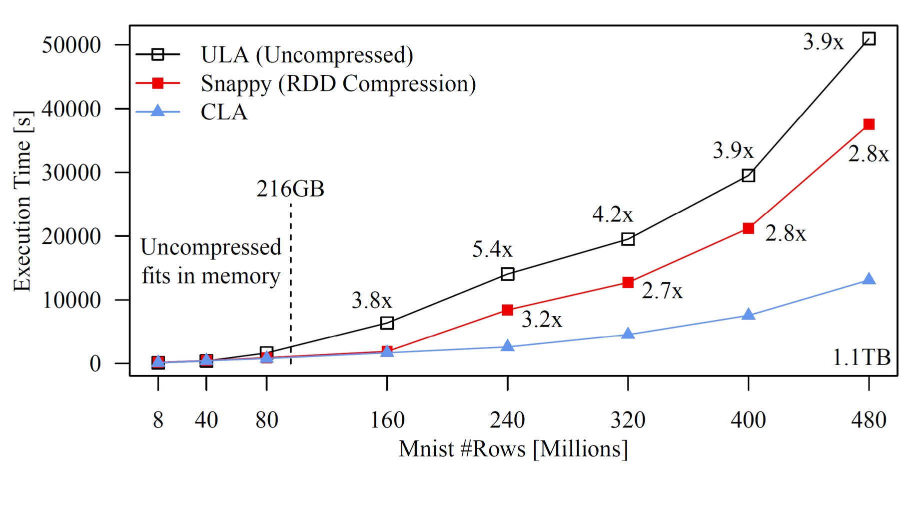


Vector-matrix multiplication



End-to-End Experiments

- **L2SVM over Mnist**



In-memory dataset Mnist40m (90GB)

Algorithm	ULA	Snappy	CLA
MLogreg	630s	875s	**622s**
GLM	409s	547s	**397s**
LinregCG	**173s**	220s	176s

Out-of-core dataset Mnist240m (540GB)

Algorithm	ULA	Snappy	CLA
MLogreg	83,153s	27,626s	**4,379s**
GLM	74,301s	23,717s	**2,787s**
LinregCG	2,959s	1,493s	**902s**