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

[2] University of Maryland, College Park

Motivation

- **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)**

X

```
while(!converged) {
  … q = X %*% v …
}
```

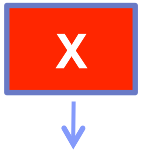
- **Goal: Improve performance of declarative ML algorithms via lossless compression**

- **Baseline solution**
  - Employ general-purpose compression techniques
  - Decompress matrix block-wise for each operation
  - Heavyweight (e.g., Gzip): **good compression ratio** / **too slow**
  - Lightweight (e.g., Snappy): **modest compression ratio** / **relatively fast**

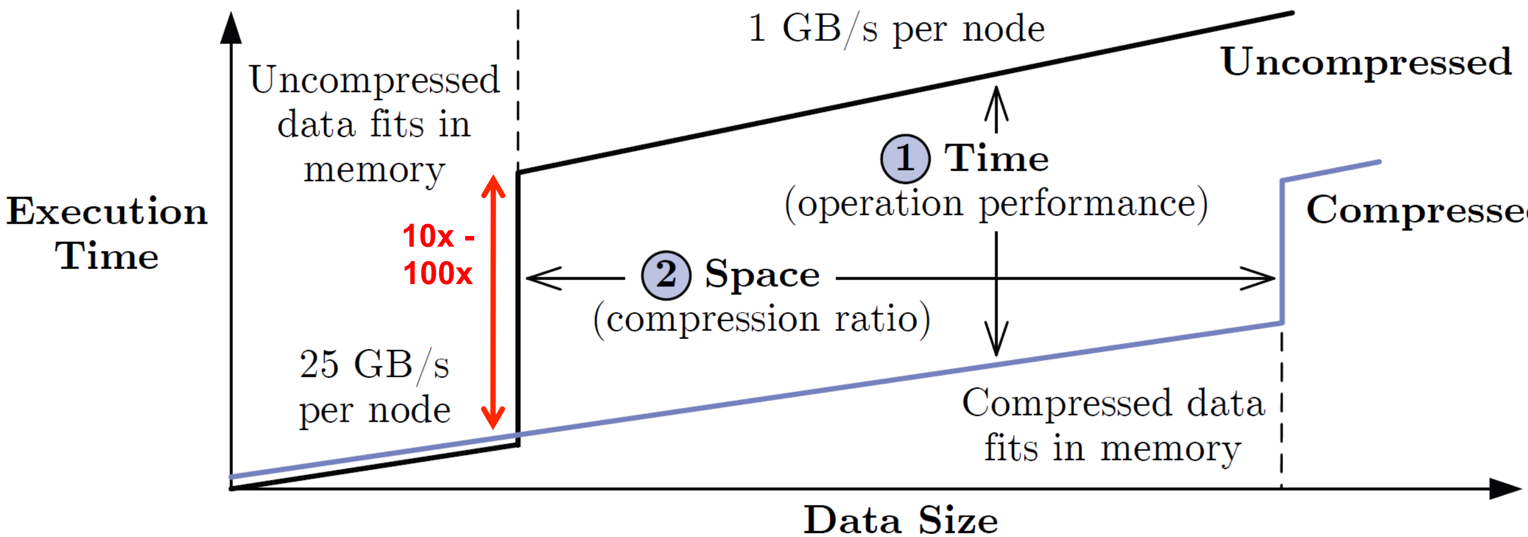
Our Approach: Compressed Linear Algebra (CLA)

- **Key idea**
  - Use lightweight database compression techniques
  - Perform LA operations **on compressed matrices**

- **Goals of CLA**
  - Operations performance close to uncompressed
  - Good compression ratios




```
while(!converged) {
  … q = X %*% v …
}
```



Our Setting: Apache SystemML

- **Overview**
  - Declarative ML algorithms with R-like syntax
  - Hybrid runtime plans single-node + MR/Spark
- **ML Program Compilation**
  - Statement blocks → DAGs
  - Optimizer rewrites
  - → **Automatic compression**
- **Distributed Matrices**
  - **Block matrices** (dense/sparse)
  - Single node: matrix = block
  - → **CLA integration via new block**
- **Data Characteristics**
  - Tall & skinny; non-uniform sparsity
  - Low col. card.; col. correlations

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");
```

$\boxed{Xv}$

$\boxed{v^T X}$

$\boxed{X^T(w * (Xv))}$

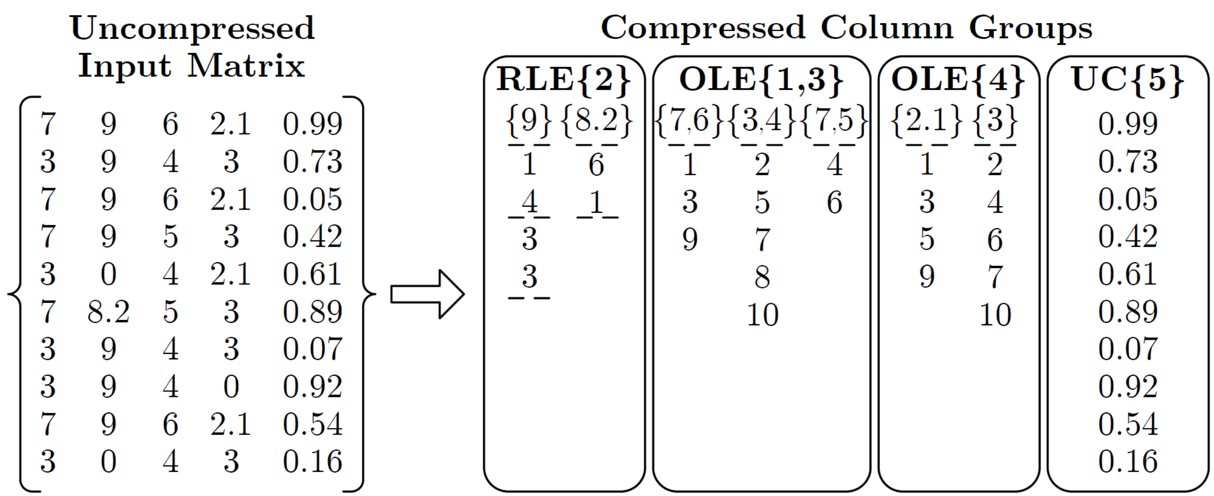
$X^T X$

→ **Column-based compression schemes**

Matrix Compression Framework

- **Overview compression framework**
  - Column-wise matrix compression (values + 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)

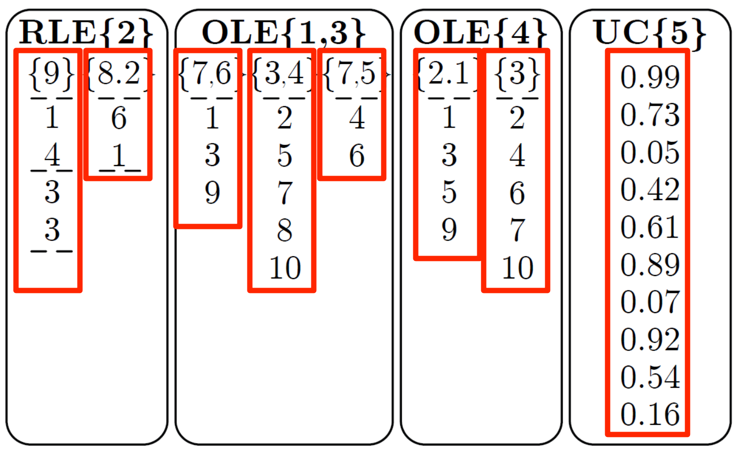


- **Automatic compression planning**
  - Selects column groups and encoding formats per group (data dependent)

Operations over Compressed Matrix Blocks

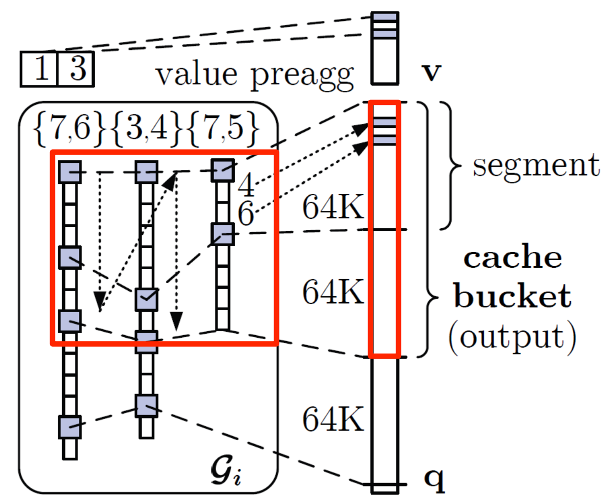
- **Matrix-vector multiplication**
  - Naïve: for each tuple, pre-aggregate values, add values at offsets to q
    Example: q = X v, with v = (7, 11, 1, 3, 2)

9*11=99.2 55 25 54 6.3 9



162.3
134.5
160.4
162.8
32.5
155
133.1
125.8
161.4
34.3

➔ cache unfriendly on output (q)



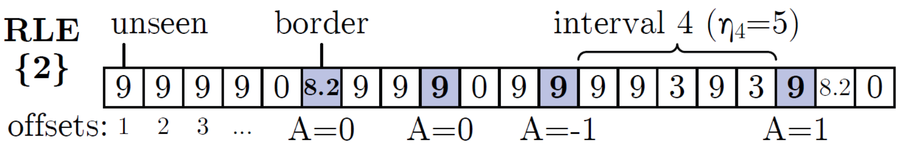
  - **Cache-conscious:** Horizontal, segment-aligned scans, maintain positions

- **Vector-matrix multiplication**
  - Naïve: **cache-unfriendly on input (v)**
  - Cache-conscious: again use horizontal, segment-aligned scans

Compression Planning

- **Goals and general principles**
  - Low planning costs ➔ **Sampling-based techniques**
  - Conservative approach ➔ **Prefer underestimating S^{UC}/S^{C} + 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.** (r_{ijk} in interval k ~ Ising-Stevens + border effects)



- **Column Group Partitioning**
  - Exhaustive grouping: **$O(m^m)$**
  - Brute-force greedy grouping: **$O(m^3)$**
    - Start with singleton groups, execute merging iterations
    - Merge groups with max compression ratio

➔ **Bin-packing-based grouping**

Compression Algorithm

- **Transpose input X**
- **Draw random sample of rows S**
- **Classify**
  - For each column
    - Estimate compression ratio (with $S^{UC} = z_i\alpha$)
    - Classify into C^C and C^{UC}
- **Group**
  - Bin packing of columns
  - Brute-force greedy per bin
- **Compress**
  - Extract uncomp. offset lists
  - Get exact compression ratio
  - Apply graceful corrections
  - Create UC Group

Algorithm 2 Matrix Block Compression

Input: Matrix block $\mathbf{X}$ of size $n \times m$

Output: A set of compressed column groups $\mathcal{X}$

$C^{\mathrm{C}} \leftarrow \emptyset,\ C^{\mathrm{UC}} \leftarrow \emptyset,\ \mathcal{G} \leftarrow \emptyset,\ \mathcal{X} \leftarrow \emptyset$

// Planning phase –

$\mathcal{S} \leftarrow$ sampleRowsUniform($\mathbf{X}$, *sample_size*)

for all column k **in** $\mathbf{X}$ **do** *// classify*

    $cmp\_ratio \leftarrow \hat{z}_i\alpha / \min(\hat{S}_k^{\mathrm{RLE}}, \hat{S}_k^{\mathrm{OLE}})$

    if $cmp\_ratio > 1$ **then**

        $C^{\mathrm{C}} \leftarrow C^{\mathrm{C}} \cup k$

    else

        $C^{\mathrm{UC}} \leftarrow C^{\mathrm{UC}} \cup k$

bins $\leftarrow$ runBinPacking(C^{C}) *// group*

for all bin b **in** *bins* **do**

    $\mathcal{G} \leftarrow \mathcal{G} \cup$ groupBruteForce(b)

// Compression phase – – – – – – – – – – – – – – – – – – –

for all column group $\mathcal{G}_i$ **in** $\mathcal{G}$ **do** *// compress*

    do

        biglist $\leftarrow$ extractBigList($\mathbf{X}, \mathcal{G}_i$)

        cmp_ratio $\leftarrow$ getExactCmpRatio(*biglist*)

        if $cmp\_ratio > 1$ **then**

            $\mathcal{X} \leftarrow \mathcal{X} \cup$ compressBigList(*biglist*), **break**

        $k \leftarrow$ removeLargestColumn($\mathcal{G}_i$)

        $C^{\mathrm{UC}} \leftarrow C^{\mathrm{UC}} \cup k$

    while $|\mathcal{G}_i| > 0$

return $\mathcal{X} \leftarrow \mathcal{X} \cup$ createUCGroup(C^{UC})

Experimental Setting

- **Cluster setup**
  - 1 head node (2x4 Intel E5530, 64GB RAM), and
    6 worker nodes (2x6 Intel E5-2440, 96GB RAM, 12x2TB disks)
  - Spark 1.4 with 6 executors (24 cores, 60GB), 25GB driver memory

- **Implementation details**
  - CLA integrated into SystemML (new rewrite injects **`compress`** operator)
  - For Spark/MR: individual matrix blocks compressed independently

- **ML programs and data**
  - 6 full-fledged ML algorithms
  - 5 real-world data sets + InfiMNIST data generator (**up to 1.1TB**)

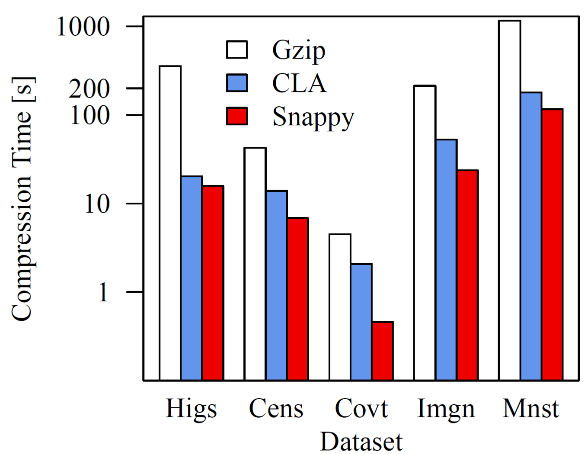
- **Selected baselines**
  - Apache SystemML 0.9 (Feb 2016) with uncompressed LA ops (**ULA**)
  - General-purpose compression with ULA (**Gzip, Snappy**)

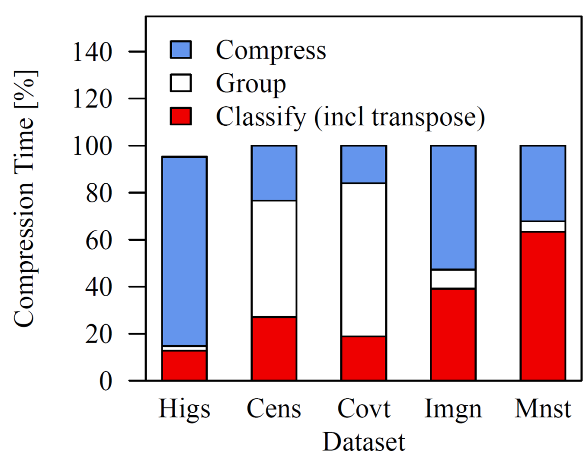
Micro-Benchmarks: Compression Ratios and Time

- **Compression ratios** (S^{UC}/S^{C}, compared to uncompressed in-memory size)

Dataset	Dimensions	Sparsity	Size (GB)	Gzip	Snappy	CLA
Higgs	11M x 28	0.92	2.5	1.93	1.38	**2.03**
Census	2.5M x 68	0.43	1.3	17.11	6.04	**27.46**
Covtype	600K x 54	0.22	0.14	10.40	6.13	**12.73**
ImageNet	1.2M x 900	0.31	4.4	5.54	3.35	**7.38**
Mnist8m	8.1M x 784	0.25	19	4.12	2.60	**6.14**

- **Compression time**

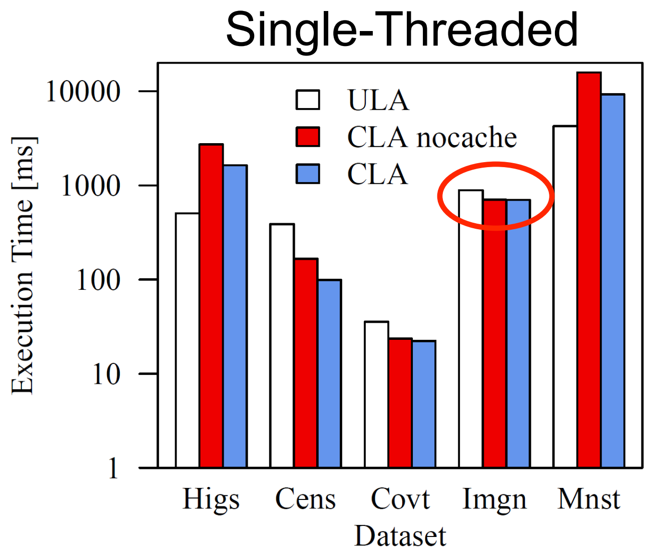


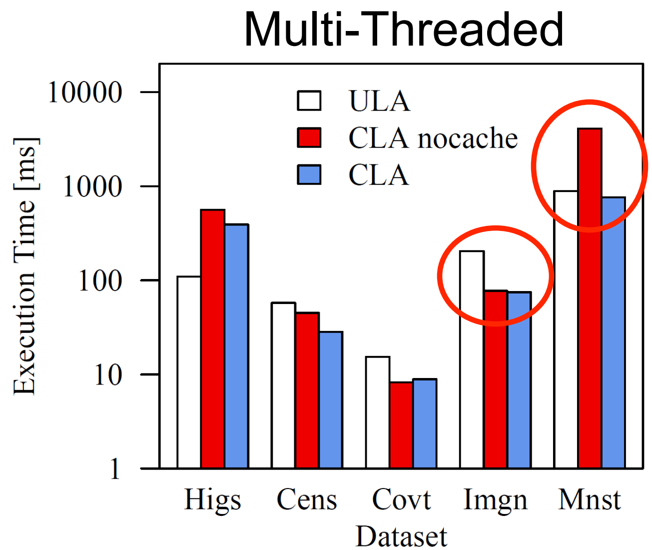


Decompression Time
(single-threaded, native libs, includes deserialization)

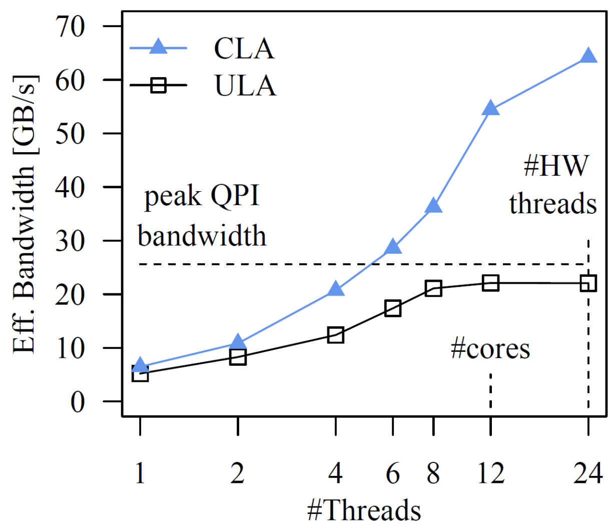
Gzip	88-291 MB/s
Snappy	232-639 MB/s
CLA	not required

Micro-Benchmarks: Vector-Matrix Multiplication





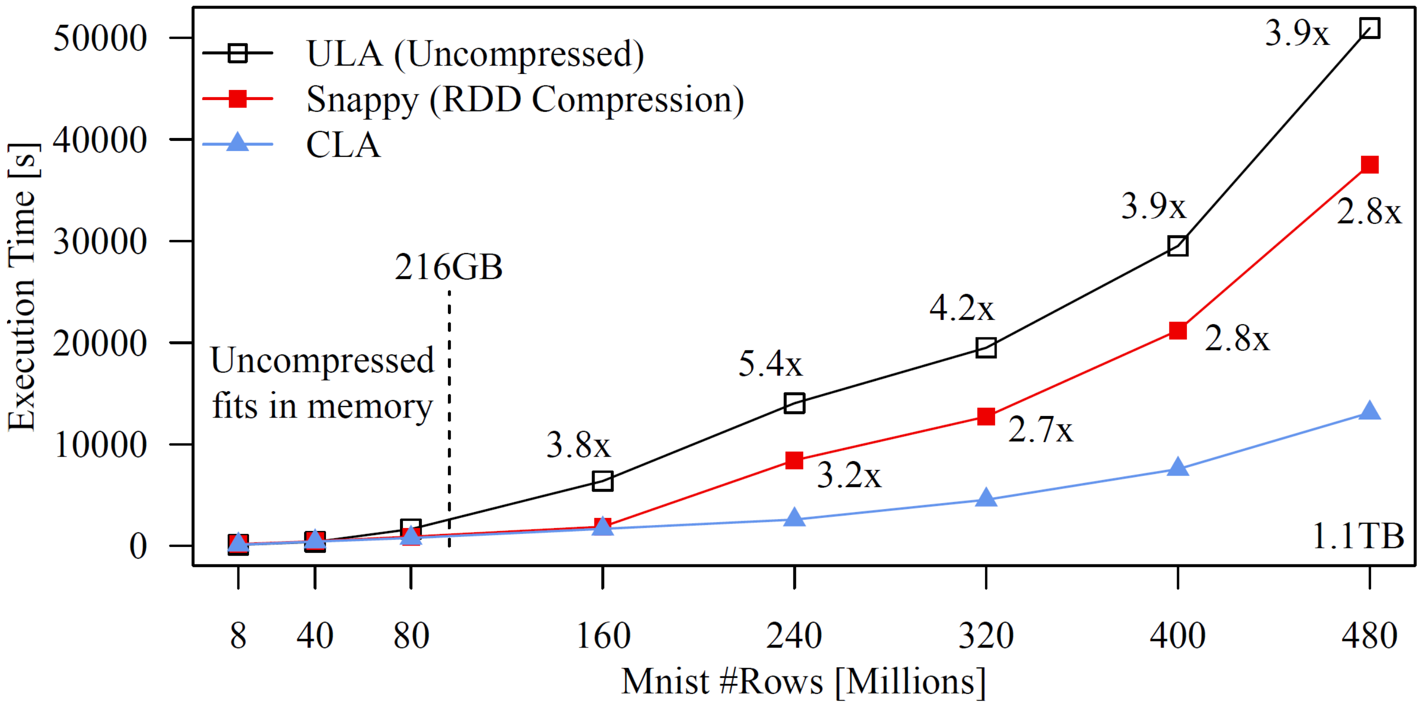
Up to **5.4x**



➔ Smaller memory bandwidth requirements of CLA

End-to-End Experiments: L2SVM

- **L2SVM over Mnist dataset**
  - End-to-end runtime, including HDFS read + **compression**
  - Aggregated mem: **216GB**



End-to-End Experiments: Other Iterative ML Algorithms

- **In-memory dataset Mnist40m** (90GB)

Algorithm	ULA	Snappy	CLA
MLogreg	630s	875s	**622s**
GLM	409s	647s	**397s**
LinregCG	**173s**	220s	176s

- **Out-of-core dataset Mnist240m** (540GB)
  - Up to **26x** and **8x**

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

Conclusions

- **Summary**
  – **CLA: Database compression + LA over compressed matrices**
  – Column-compression schemes and ops, sampling-based compression
  – Performance close to uncompressed + good compression ratios

- **Conclusions**
  – **General feasibility of CLA, enabled by declarative ML**
  – Broadly applicable (blocked matrices, LA, data independence)

- **SYSTEMML-449:** Compressed Linear Algebra
  – Transferred back into upcoming Apache SystemML 0.11 release
  – Testbed for extended compression schemes and operations

Upcoming:
Tue Sep 6, 2pm
I2: SystemML on Spark

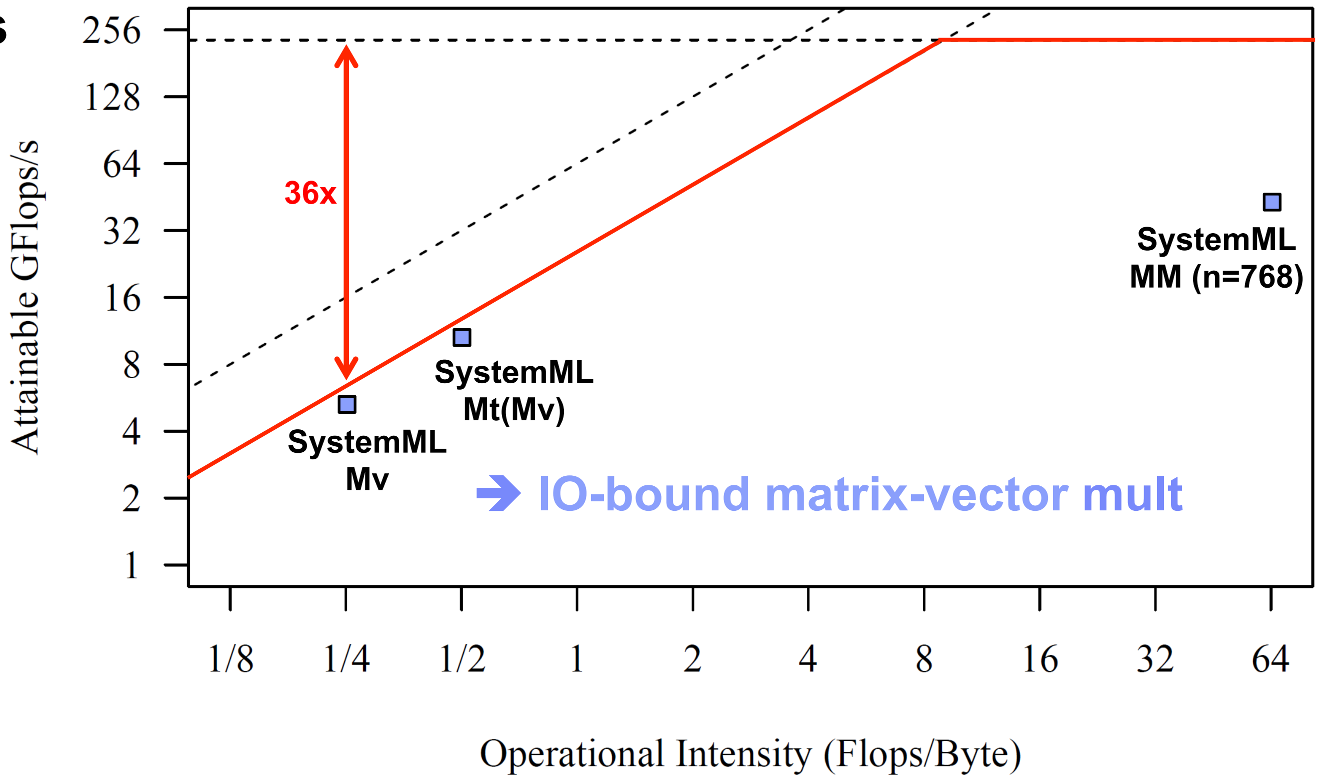
Wed Sep 7, 11.15am
D3b: CLA Poster

Fri Sep 9, 9am-5.30pm
Tutorial @BOSS

SystemML is Open Source:
Apache Incubator Project since 11/2015
Website: http://systemml.apache.org/
Sources: https://github.com/apache/incubator-systemml

Backup: Roofline Analysis Matrix-Vector Multiply

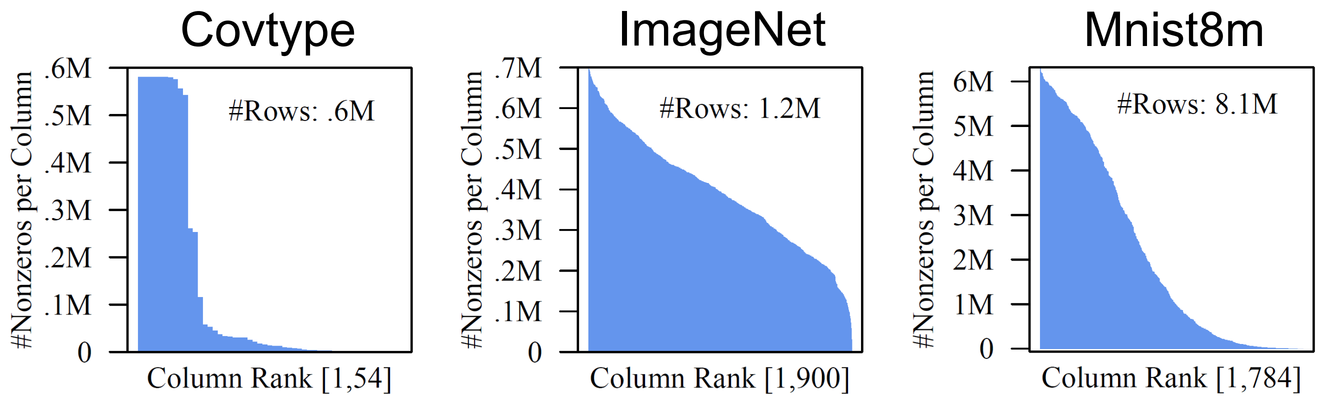
- **Single Node:** 2x6 E5-2440 @2.4GHz–2.9GHz, DDR3 RAM @1.3GHz (ECC)
  - Max mem bandwidth (local): 2 sock x 3 chan x 8B x 1.3G trans/s → **2 x 32GB/s**
  - Max mem bandwidth (single-sock ECC / QPI full duplex) → **2 x 12.8GB/s**
  - Max floating point ops: 12 cores x 2*4dFP-units x 2.4GHz → **2 x 115.2GFlops/s**
- **Roofline Analysis**
  - Processor performance
  - Off-chip memory traffic



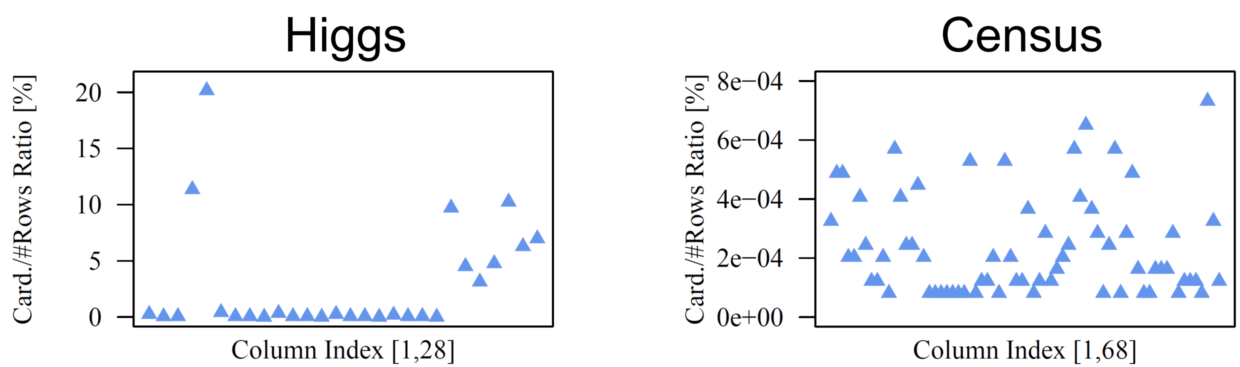
[S. Williams, A. Waterman, D. A. Patterson: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52(4): 65-76 (2009)]

Backup: Common Data Characteristics

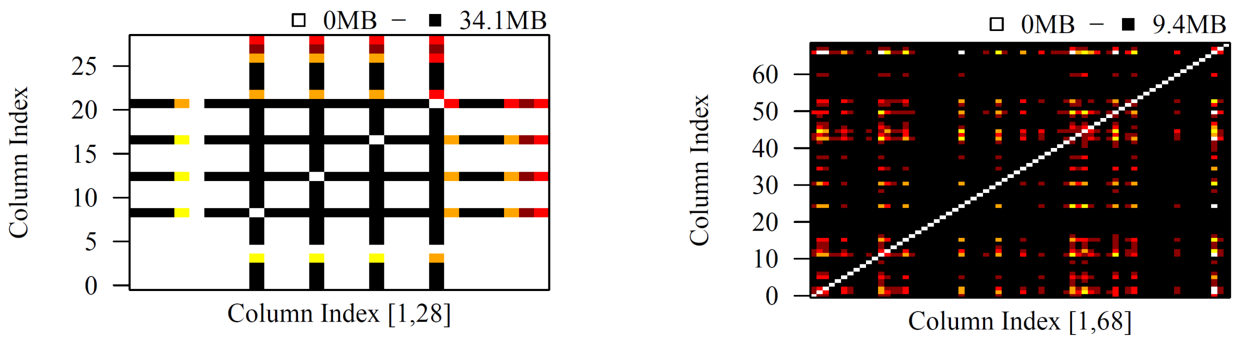
- **Non-Uniform Sparsity**



- **Low Column cardinalities**

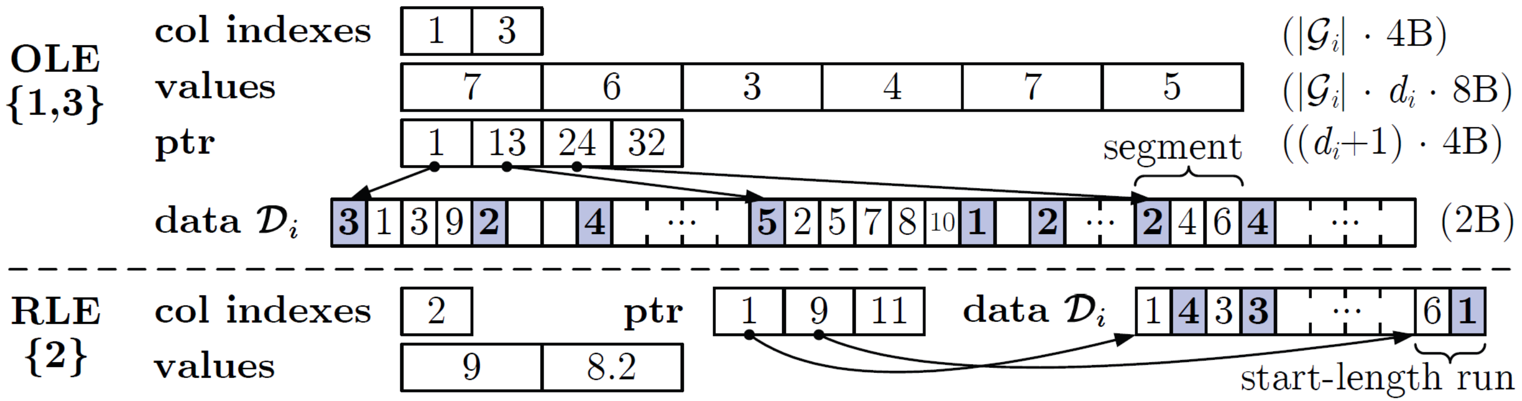


- **Column Correlation**
  - For Census: **10.1x → 27.4x**



Backup: Column Encoding Formats

- **Data Layout**
  - OLE
  - RLE

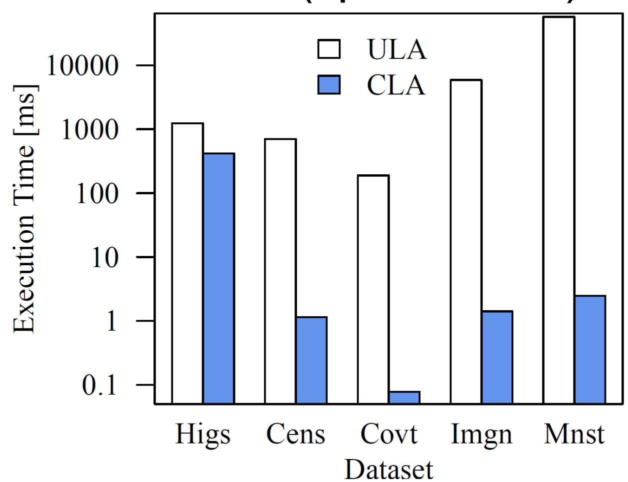


- **Offset-List Encoding**
  - Offset range divided into *segments* of fixed length $\Delta^s=2^{16}$
  - Offsets encoded as diff to beginning of its segment
  - Each segments encodes length w/ 2B, followed by 2B per offset
- **Run-Length Encoding**
  - Sorted list of offsets encoded as sequence of *runs*
  - Run starting offset encoded as diff to end of previous run
  - Runs encoded w/ 2B for starting offset and 2B for length
  - Empty/partitioned runs to deal with max 2^{16} diff

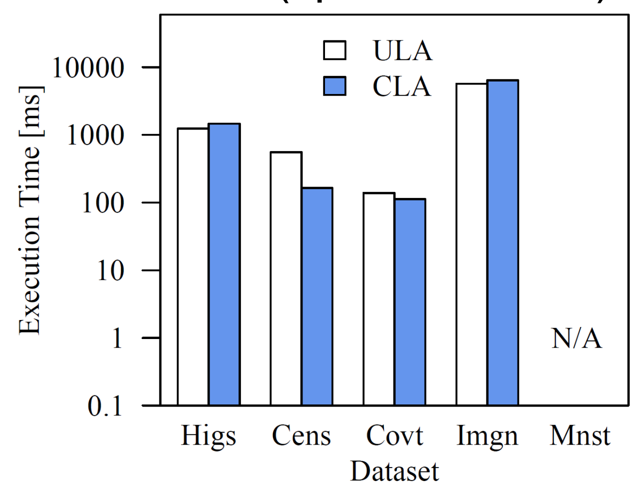
Backup: Scalar Operations and Aggregates

- **Scalar Operations**
  - Single-threaded
  - Up to **1000x – 10,000x**

`X^2` (sparse-safe)

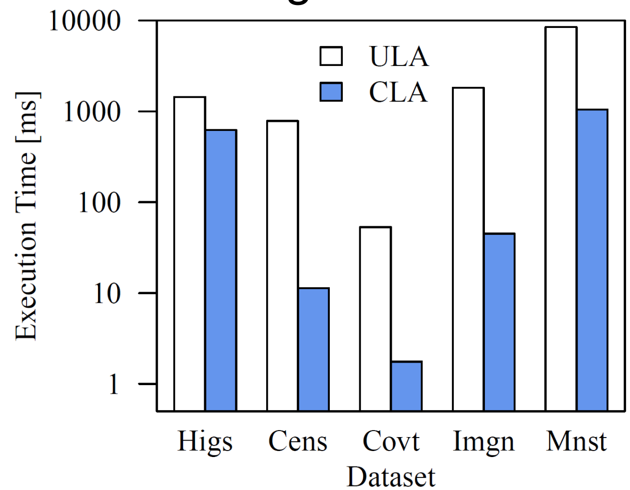


`X+7` (sparse-unsafe)

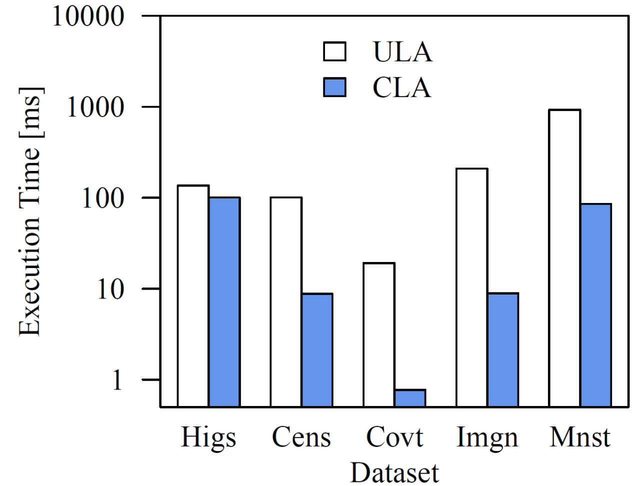


- **Unary Aggregates**
  - `sum(X)`
  - Up to **100x**

Single-Threaded



Multi-Threaded



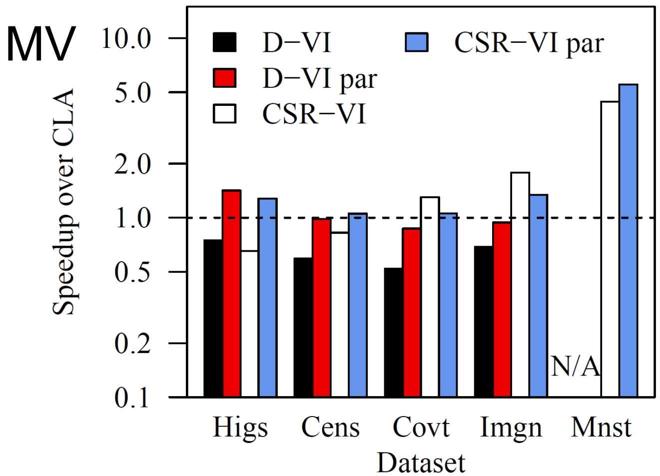
Backup: Comparison CSR-VI (CSR Value Indexed)

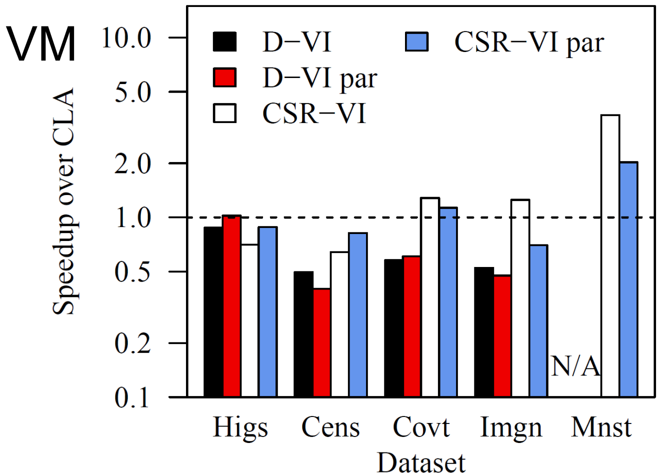
- **Compression Ratio**

Dataset	Sparse	#Distinct	CSR-VI	D-VI	CLA
Higgs	N	8,083,944	1.04	1.90	**2.03**
Census	N	46	3.62	7.99	**27.46**
Covtype	Y	6,682	3.56	2.48	**12.73**
ImageNet	Y	824	2.07	1.93	**7.38**
Mnist8m	Y	255	2.53	N/A	**6.14**

- **Operations Performance**

[K. Kourtis, G. I. Goumas, N. Koziris: Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression, CF 2008, 87-96]

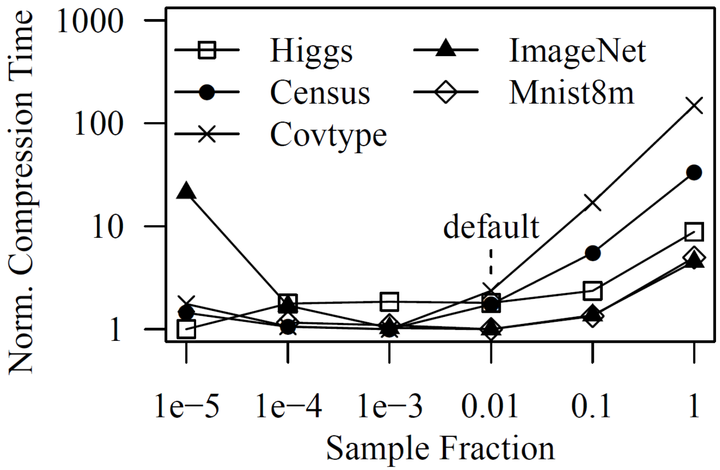




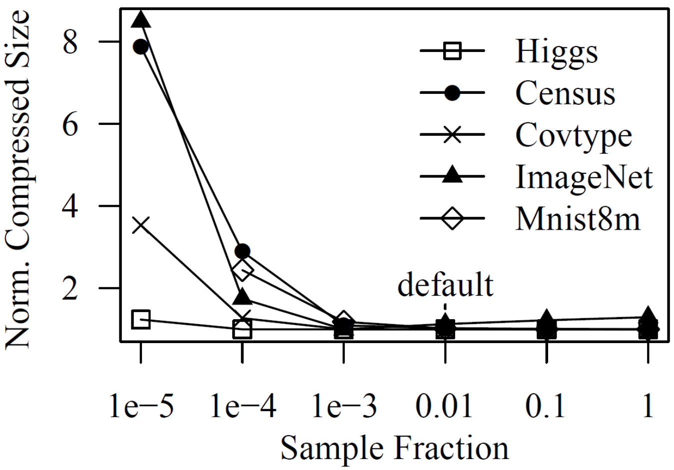
Backup: Parameter Influence and Accuracy

- **Sample Fraction**

Compression Time (minimum normalized)



Compressed Size (minimum normalized)



- **Estimation Accuracy**

Dataset	Higgs	Census	Covtype	ImageNet	Mnist8m
Excerpt	28.8%	173.8%	111.2%	24.6%	**12.1%**
CLA Est.	**16.0%**	**13.2%**	**56.6%**	**0.6%**	39.4%

[C. Constantinescu, M. Lu: Quick Estimation of Data Compression and De-duplication for Large Storage Systems. CCP 2011, 98-102]