ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation

Kaan et al.

VLDB2019

Outline

- **Background & Motivation**
- **Proposed solutions & experimental evaluation**
  - Training
  - Data transformation
- **Conclusion**

Motivation

- **Column-store main-memory DBMS**
- **In database machine learning avoids data movement.**

Background

- **Stochastic coordinate descent**

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left(\frac{1}{m} \sum_{i=1}^{m} J(\langle \mathbf{x}, \mathbf{a}_i \rangle, b_i) \right) + \lambda \|\mathbf{x}\|_1 \quad (1)$$

$$J = \begin{cases} \frac{1}{2}(\langle \mathbf{x}, \mathbf{a}_i \rangle - b_i)^2 & \text{for Lasso} \\ -b_i \log(h_{\mathbf{x}}(\mathbf{a}_i)) - (1 - b_i) \log(1 - h_{\mathbf{x}}(\mathbf{a}_i)) & \text{for Logreg} \end{cases} \quad (2)$$

x: model parameters

a: training data

z: dot product of x and a

S: for computing model updates

T: model updates

Algorithm 1: Stochastic Coordinate Descent

Initialize:

· $\mathbf{x} = 0$, $\mathbf{z} = 0$, step size α

· $S(\mathbf{z}) = \begin{cases} \mathbf{z} & \text{for Lasso} \\ 1/(1 + \exp(-\mathbf{z})) & \text{for Logreg} \end{cases}$

· $T(x_j, g_j) = \begin{cases} \alpha g_j + \alpha\lambda & x_j - \alpha g_j > \alpha\lambda \\ \alpha g_j - \alpha\lambda & x_j - \alpha g_j < -\alpha\lambda \\ x_j & \text{else (to set } x_j = 0) \end{cases}$

for *epoch = 1, 2, ...* **do**

— randomly without replacement

for *j = shuffle(1, ..., n)* **do**

$g_j = \frac{1}{m}(S(\mathbf{z}) - \mathbf{b}) \cdot \mathbf{a}_{:,j}$ — partial gradient computation

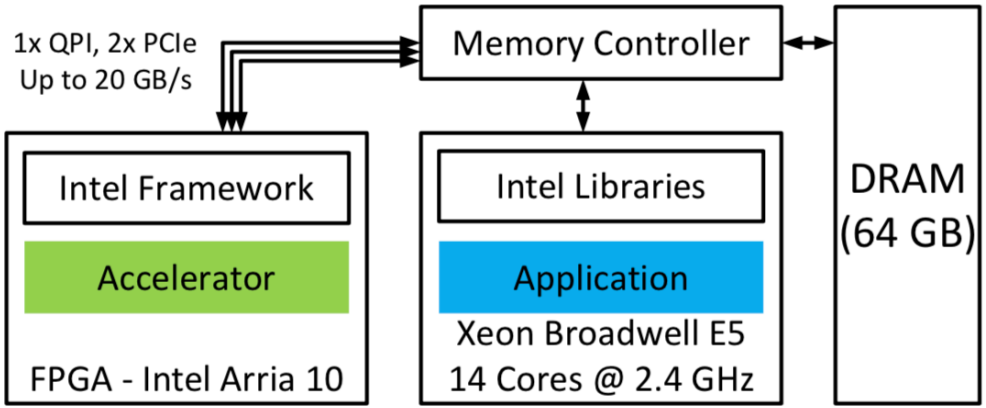
$\mu = T(x_j, g_j)$ — thresholding due to regularization

$x_j = x_j - \mu$ — coordinate update

$\mathbf{z} = \mathbf{z} - \mu \mathbf{a}_{:,j}$ — inner-product vector update

Background

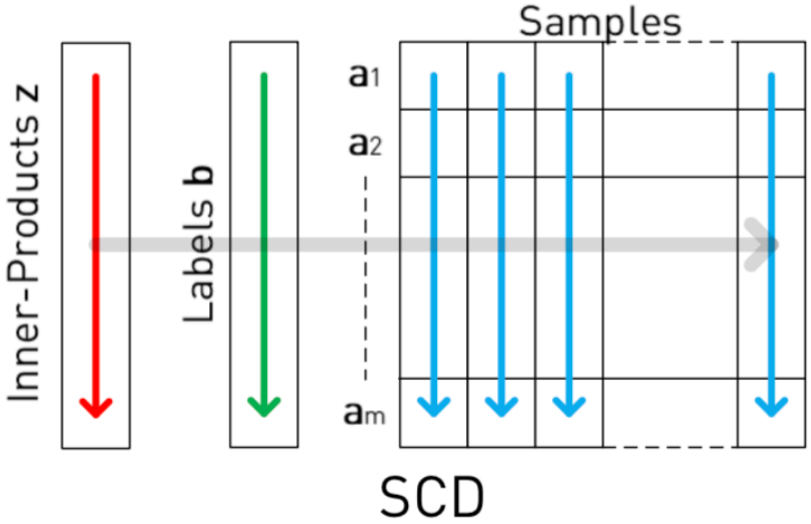
- **Target Platform**



multi-core CPU: cache matters
FPGA: pipeline and parallelism

Figure 1: The target platform: Intel Xeon+FPGA v2.

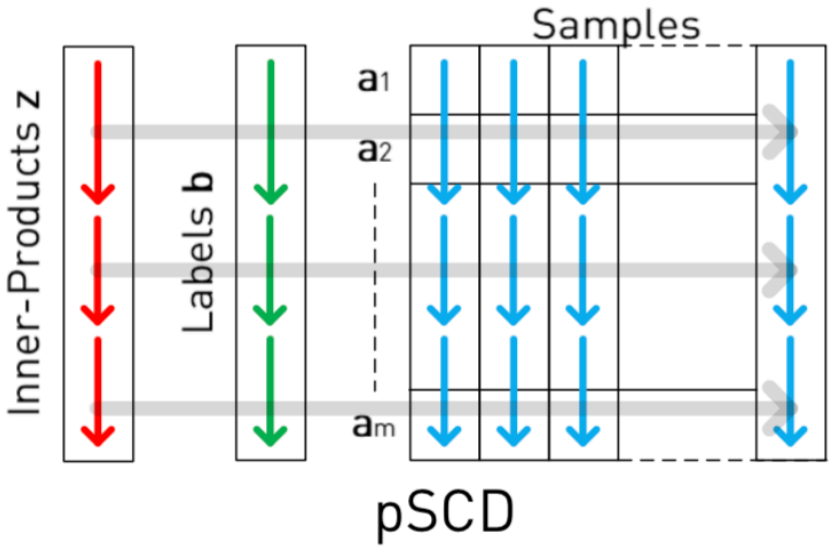
SCD



Challenges:

The dot product is too large.

Partitioned SCD



1. split the training data into several partitions.
2. train a separate model on each partition.
3. average the models periodically.

Algorithm 2: Partitioned SCD

Initialize:

- $\mathbf{x}[K] = 0$, $\mathbf{z} = 0$, step size α
- $S(\mathbf{z})$ and $T(x_j, g_j)$ as in Algorithm 1
- partition size M, number of partitions $K = m/M$
- inner-product update period P

for *epoch = 1, 2, ...* **do**

  for k *= 0, ..., K-1 (each partition)* **do**

    — randomly without replacement

    for j *= shuffle(1, ..., n)* **do**

      $subset = kM + 1, ..., kM + M$

      — partial gradient computation

      $g_j = (S(\mathbf{z}_{subset}) - \mathbf{b}_{subset}) \cdot \mathbf{a}_{subset,j}$

      — thresholding due to regularization

      $\mu = T(x[k]_j, g_j)$

      — coordinate update

      $x[k]_j = x[k]_j - \mu$

      — inner-product vector update

      $\mathbf{z}_{subset} = \mathbf{z}_{subset} - \mu \mathbf{a}_{subset,j}$

  } (1)

  — global inner-product update with the averaged model

  if *epoch* mod P **then**

    $\bar{\mathbf{x}} = (\mathbf{x}[0] + ... + \mathbf{x}[K-1])/K$

    $\mathbf{z} = 0$

    for k *= 0, ..., K-1 (each partition)* **do**

      $subset = kM + 1, ..., kM + M$

      for j*= 1, ..., n* **do**

        $\mathbf{z}_{subset} = \mathbf{z}_{subset} + \bar{x}_j \mathbf{a}_{subset,j}$

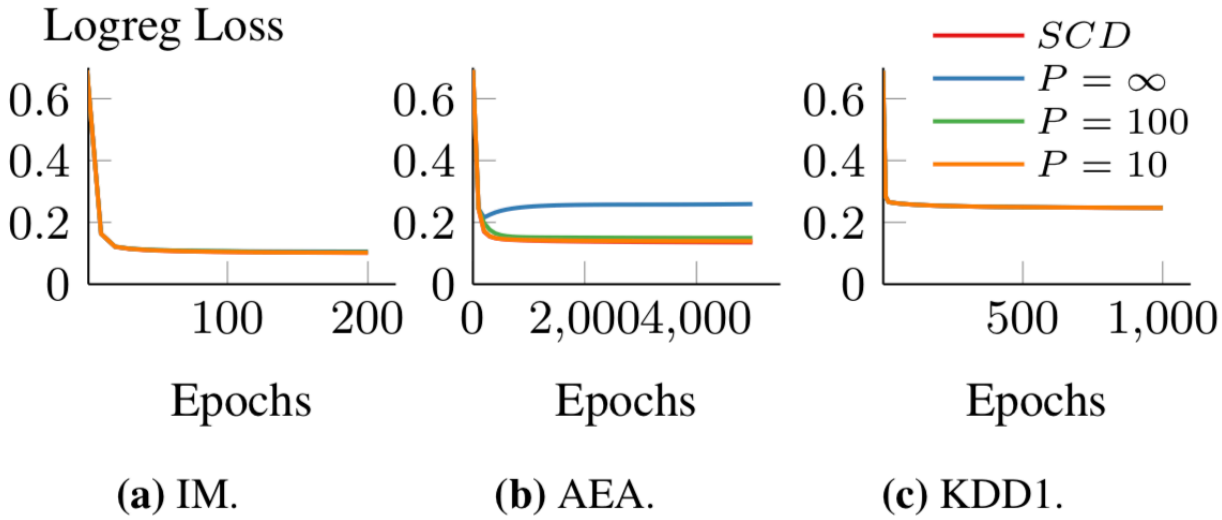
  } (2)

Experimental environment

Table 1: Data sets used in the evaluation.

Name	# Samples	# Features	Size	Type
IM	332,800	2,048	2,726 MB	classification
AEA	32,769	126	16,5 MB	classification
KDD1	391,088	2,399	2,188 MB	classification
KDD2	131,329	2,330	1,224 MB	classification
SYN1	33,554,432	16	2,147 MB	regression
SYN2	2,097,152	256	2,147 MB	regression

Analysis of pSCD (Statistical Efficiency)

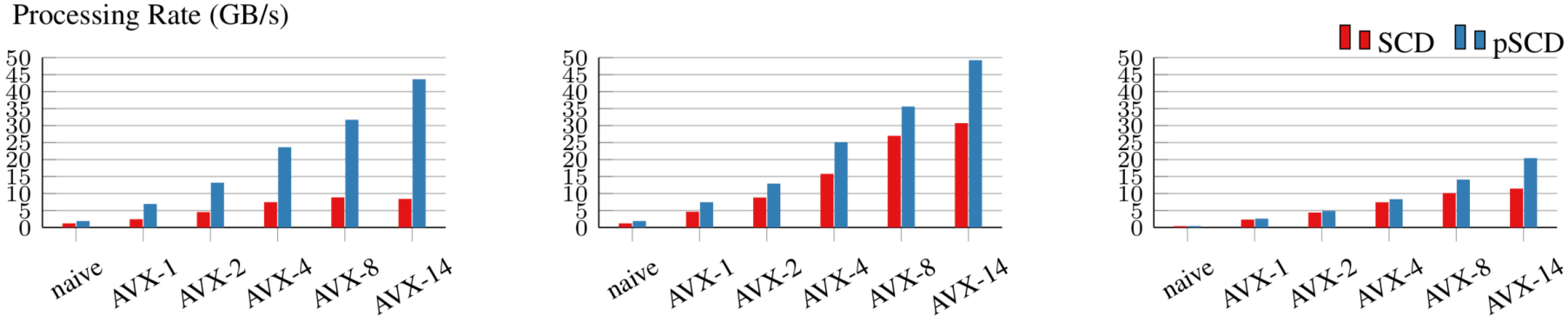


(a) IM. (b) AEA. (c) KDD1.

Figure 5: Convergence of the Logreg loss for three data sets trained with either SCD or pSCD with varying P.

Analysis of pSCD (Hardware Efficiency)

- **Multi-threaded implementation.**



(a) Lasso, SYN1 **(b)** Lasso, SYN2 **(c)** Logreg, IM

Figure 4: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. Partition size: 16384. For pSCD $P = 10$.

pSCD vs SGD

Table 3: Algorithms used in comparison analysis. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics.

Name	Minibatch Size	Step Size	Impl.	Storage
SGD-tf	512	0.1	default	row-store
SGD-row	1	0.01	AVX-1	row-store
SGD-col-1	1	0.01	AVX-1	column-store
SGD-col-8	8	0.1	AVX-1	column-store
SGD-col-64	64	0.5	AVX-1	column-store
SGD-col-512	512	0.9	AVX-1	column-store

Name	Partition Size	Step Size	Impl.	Storage
SCD	-	4	AVX-1	column-store
pSCD	16384	4	AVX-1	column-store

pSCD vs SGD (Hardware Efficiency)

- **Single-threaded implementation.**

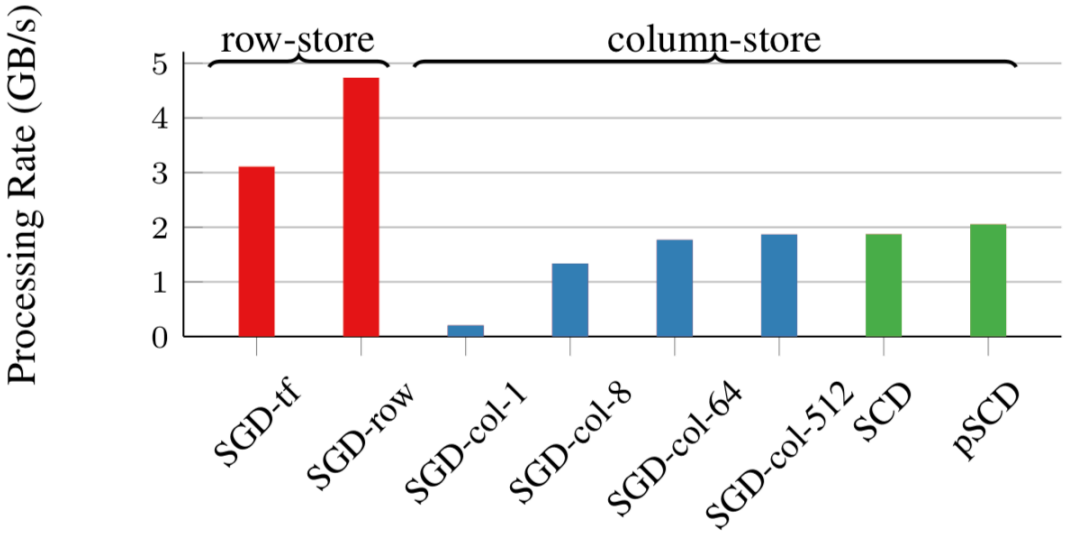
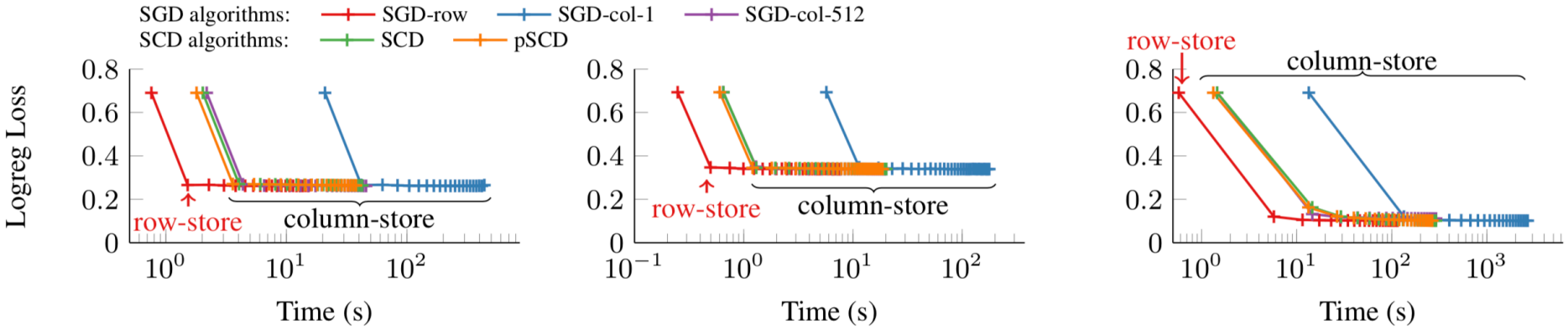


Figure 6: Throughput of algorithms while running Logreg on IM. For pSCD $P = 10$.

pSCD vs SGD (Hardware + Statistical Efficiency)

- **Single-threaded implementation.**



(a) Convergence over time for KDD1.

(b) Convergence over time for KDD2.

(c) Convergence over time for IM.

Figure 7: Convergence of the Logreg loss using different training algorithms on three data sets, plotted over time to observe the combined effect of hardware and statistical efficiencies. Partition size: 16384. For pSCD $P = 10$.

What about compressed and encrypted column-stores?

- **Performance of data transformation on CPU.**

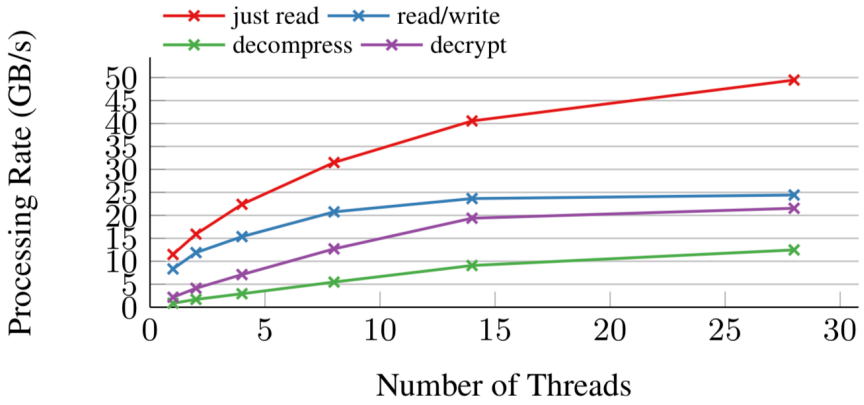


Figure 9: CPU scaling of read bandwidth, read/write bandwidth, decryption and decompression rates.

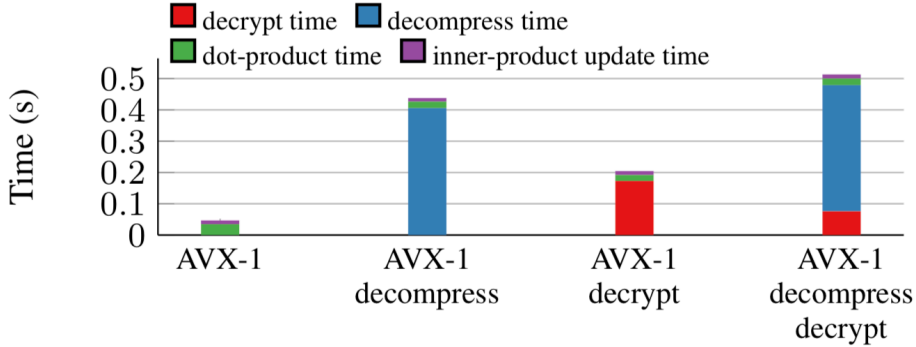
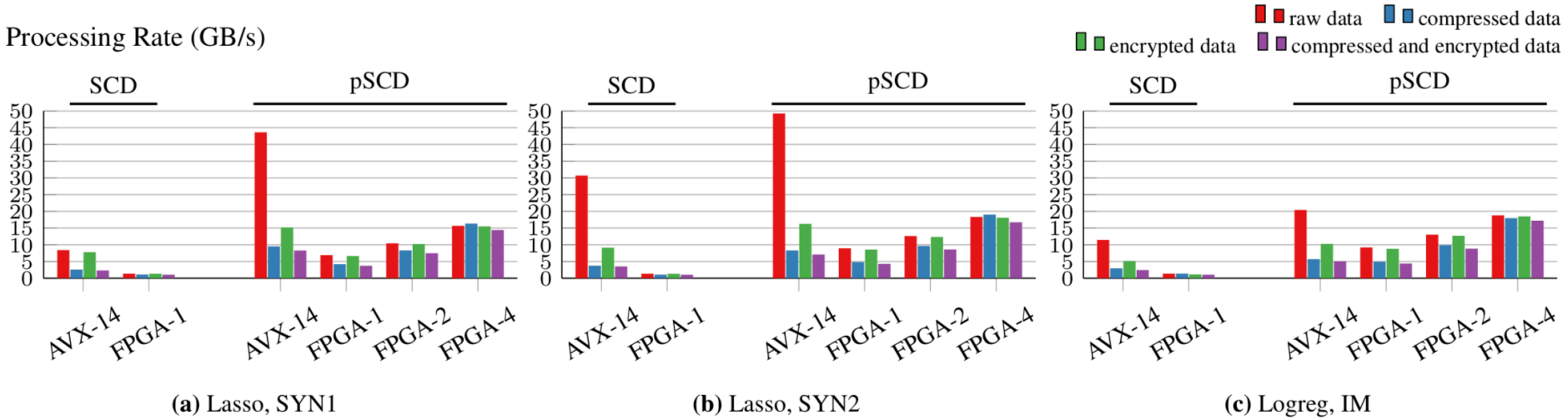


Figure 10: CPU breakdown analysis for pSCD with on-the-fly data transformation. Data: 1 Million samples, 90 features. Partition size: 8192. For pSCD $P = 10$.

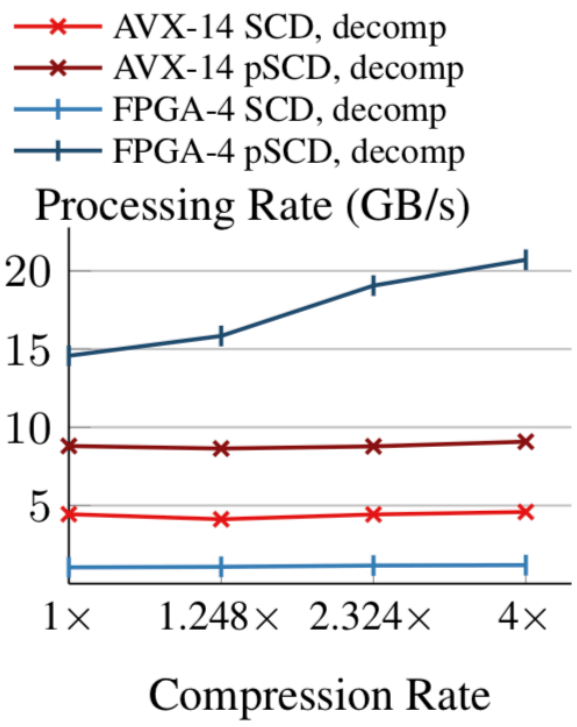
Conclusion

- **If row-store, SGD is the best.**
- **If column-store, pSCD and SGD with big batch size have similar performance on CPUs.**
- **On FPGAs, SGD with large batch size consume large memory. But, is pSCD is better?**

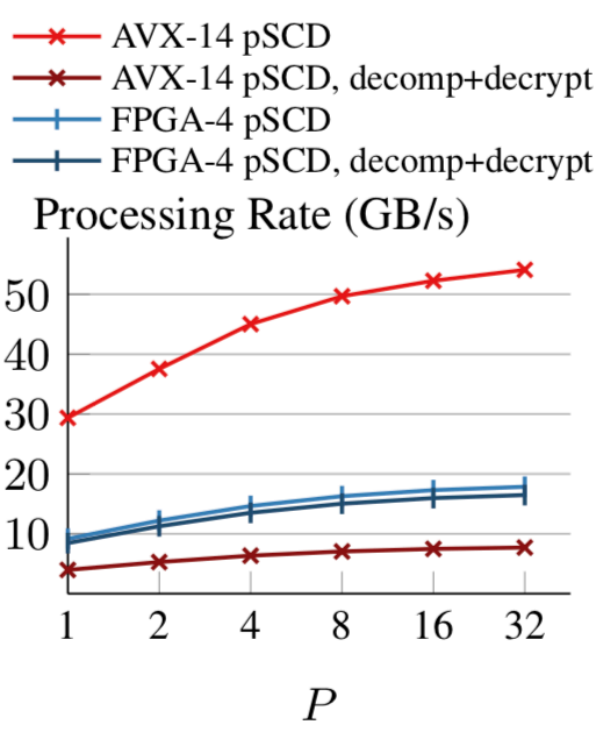


(a) Lasso, SYN1 **(b)** Lasso, SYN2 **(c)** Logreg, IM

Figure 14: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. FPGA-N denotes using N *SCD Engines* simultaneously. Partition size: 16384. For pSCD $P = 10$.



(a) Compression rate analysis. For pSCD $P = 10$.



(b) Global inner-product update period P analysis.

Figure 15: Sample processing rate shown at different compression rates and increasing global inner-product update periods P on the multi-core CPU and FPGA. Lasso, SYN2. Partition size: 16384.

Limitations:

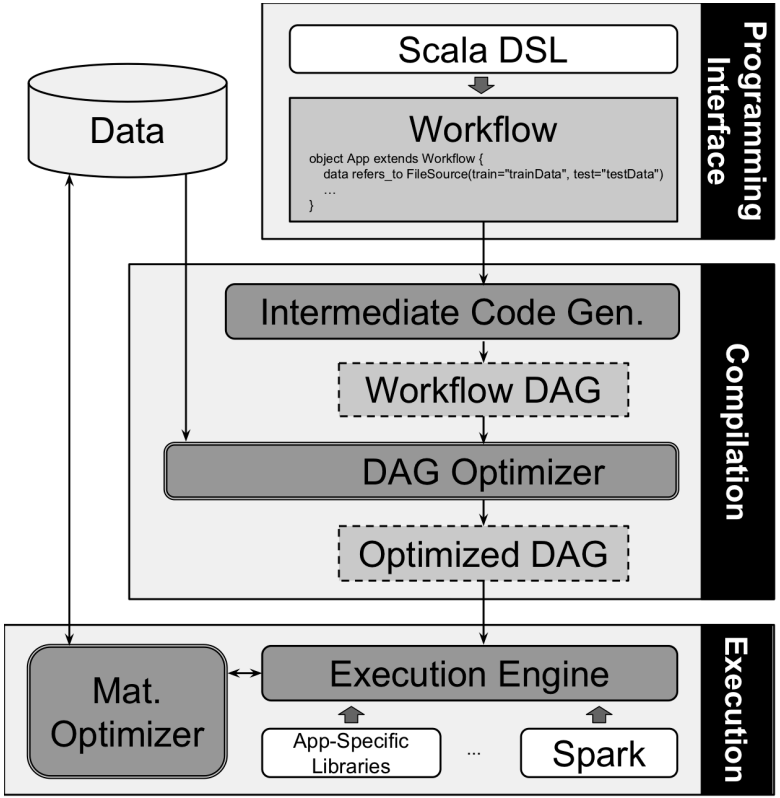
- **The data needs to be shuffled, i.e., the data items should be i.i.d.**
- **Model size is small.**

Other ML papers in VLDB 2019 (published)

Holistic Optimization for Accelerating Iterative Machine Learning

Doris Xin et al. [UIUC] VLDB 2019

Helix



Rafiki: Machine Learning as an Analytics Service System

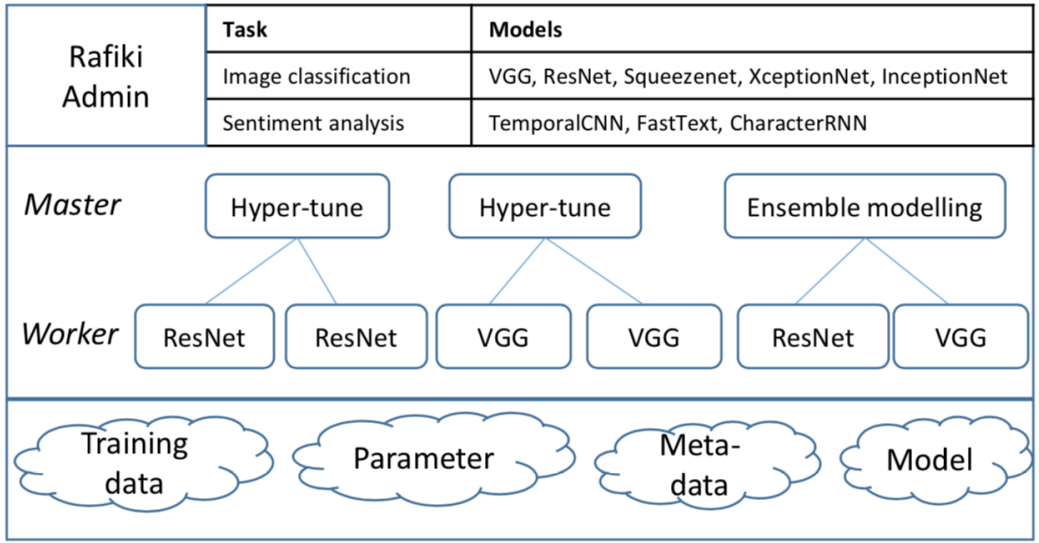
Wei Wang et al. [NUS] VLDB 2019

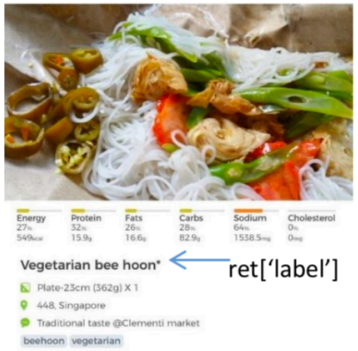
```
# train.py
import rafiki

data = rafiki.upload('train/')
job = rafiki.Train(app='food',
        data=data,
        task="ImageClassification",
        input_shape=(3,256,256),
        output_shape=(120,))
```

```
# infer.py
job = rafiki.Inference(app='food')
```

Rafiki user code






```
img = Image.load('test.jpg')
ret = rafiki.query(app='food',
        data={'img':img})
```

Application user code

Figure 2: Overview of Rafiki.

Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-precision Learning

Zeke Wang et al. [ETH] VLDB 2019