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

Presented by Yuji Chai, Owen Niles, Demi Guo
CS265
Generalized Linear Model (GLM)
GLM

DBMS

COUNT

MIN

MAX

AVG

ColumnML machine learning interface

GLM
~/.data $ ls -1
aghhhh.csv
moms_brownie_recipe.csv
please_work.csv
please_work_73.csv
train.csv
train_2.csv
train_old.csv
train_old_old.csv
~/data $ ls -1
ahhhhh.csv
moms_brownie_recipe.csv
please_work.csv
please_work_73.csv
train.csv
train_2.csv
train_old.csv
train_old_old.csv
TRAINING ALGORITHM

DBMS
- compressed
- encrypted

DATA
- decompressed
- decrypted

~/data $ ls -1
aghhhh.csv
moms_brownie_recipe.csv
please_work.csv
please_work_73.csv
train.csv
train_2.csv
train_old.csv
train_old_old.csv

GLM
DBMS
- compressed
- encrypted

```
~/data $ ls -1
ahhhh.csv
moms_brownie_recipe.csv
please_work.csv
please_work_73.csv
train.csv
train_2.csv
train_old.csv
train_old_old.csv
```

DATA
- decompressed
- decrypted

GLM
DBMS

COUNT

MIN

MAX

AVG

ColumnML machine learning interface

on-the-fly decryption and decompression

GLM
Figure 1: The target platform: Intel Xeon+FPGA v2.
Figure 1: The target platform: Intel Xeon+FPGA v2.
Generalized Linear Model (GLM)
Generalized Linear Model (GLM)

```json
{
  "cute": 0.6,
  "friendly": -0.2,
  "furry": 0.83,
  "person's best friend": -1
}
```
\[ a_i \in [-1, 1]^n \] Generalized Linear Model (GLM)
Generalized Linear Model (GLM)

\[ a_i \in [-1, 1]^n \rightarrow b_i \in \{0, 1\} \]
Generalized Linear Model (GLM)

\[ a_i \in [-1, 1]^n \]

\[ x \in \mathbb{R}^n \]

\[ b_i \in \{0, 1\} \]
How do we train such a model?

Generalized Linear Model (GLM)

\[ a_i \in [-1, 1]^n \rightarrow x \in \mathbb{R}^n \rightarrow b_i \in \{0, 1\} \]
How do we train such a model?

Generalized Linear Model (GLM)

\[ a_i \in [-1, 1]^n \quad \rightarrow \quad \mathbf{x} \in \mathbb{R}^n \quad \rightarrow \quad b_i \in \{0, 1\} \]
Solve:

\[
\min_{x \in \mathbb{R}^n} \left( \frac{1}{m} \sum_{i=1}^{m} J(\langle x, a_i \rangle, b_i) \right) + \lambda \|x\|_1
\]

\[
J = \langle \text{optimized out} \rangle
\]
Solve:

\[
\min_{x \in \mathbb{R}^n} \left( \frac{1}{m} \sum_{i=1}^{m} J(\langle x, a_i \rangle, b_i) \right) + \lambda \|x\|_1
\]
Inner Product

\[ \langle x, a_i \rangle = \langle x_1, x_2, \ldots, x_n \rangle \cdot \langle a_{i1}, a_{i2}, \ldots, a_{in} \rangle \]
Inner Product

\[ \langle x, a_i \rangle = \langle x_1, x_2, \ldots, x_n \rangle \cdot \langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle \]

\[ = x_1 a_{i_1} + x_2 a_{i_2} + \ldots + x_n a_{i_n} \]
Inner Product

\[ \langle x, a_i \rangle = \langle x_1, x_2, \ldots, x_n \rangle \cdot \langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle = x_1 a_{i_1} + x_2 a_{i_2} + \ldots + x_n a_{i_n} \]
Inner Product

$$\langle x, a_i \rangle = \langle x_1, x_2, \ldots, x_n \rangle \cdot \langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle$$

$$= x_1 a_{i_1} + x_2 a_{i_2} + \ldots + x_n a_{i_n}$$
Stochastic Gradient Descent

$$\langle x, a_i \rangle = \langle x_1, x_2, \ldots, x_n \rangle \cdot \langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle$$

$$= x_1 a_{i_1} + x_2 a_{i_2} + \ldots + x_n a_{i_n}$$
Stochastic Coordinate Descent

| $z_1$ | $z_2$ | $z_3$ | ... | $z_{m-1}$ | $z_m$ |

```python
for col in {1..$n$}; do
    update $z$ $col$
done
```

1 2 n 1 2 ...

m b x
Stochastic Coordinate Descent

<table>
<thead>
<tr>
<th>$z_1$</th>
<th>$z_2$</th>
<th>$z_3$</th>
<th>...</th>
</tr>
</thead>
</table>

```
for col in {1..$n}$; do
  update $z$ $col$
done
```
COLUMN-ML

Section 3 & 4 (Demi)
MOTIVATION

Of pSCD
WHY IS SGD NOT EFFICIENT ON COLUMN-STORE?

\[ g_j = \frac{1}{m} (S(z_j) - b_j) \cdot a_j \]

\[ z_j = \langle x, a_j \rangle \]
HOW DOES SCD WORK?

EPOCH = 1; J = 3

\[ g_j = \frac{1}{m} (S(z) - b) \cdot a_{:,j} \] — partial gradient computation
\[ \mu = T(x_j, g_j) \] — thresholding due to regularization
\[ x_j = x_j - \mu \] — coordinate update
\[ z = z - \mu a_{:,j} \] — inner-product vector update

\[ g_j = \frac{1}{m} (S(z) - b) \cdot a_{:,j} \] — partial gradient computation
\[ \mu = T(x_j, g_j) \] — thresholding due to regularization
\[ x_j = x_j - \mu \] — coordinate update
\[ z = z - \mu a_{:,j} \] — inner-product vector update

**Diagram:**
- **z**
- **Labels b**
- **Samples a**
- **Model x**

```
x1  x2  x3  x4  ...  xn-1  xn
```
HOW DOES SCD WORK?

$EPOCH = 1; J = 3$

$\mathbf{z} \quad \text{Labels } \mathbf{b} \quad \text{Samples } \mathbf{a}$

$$g_j = \frac{1}{m_0} (S(\mathbf{z}) - \mathbf{b}) \cdot \mathbf{a}_{:,j}$$  — 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
HOW DOES SCD WORK?

$EPOCH = 1; J = 3$

$g_j = \frac{1}{n} (S(z) - b) \cdot a_{i,j}$ — partial gradient computation

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

$x_j = x_j - \mu$ — coordinate update

$z = z - \mu a_{i,j}$ — inner-product vector update
WHAT IS pSCD
PSCD: CACHE-CONSCIOUS SCD

Run SCD on each partition of the data!

Different model for each partition,
Average every $P$ epochs!

$z$

Labels $b$

Samples

$k=1$

$\begin{array}{c}
\downarrow \\
\downarrow \\
\end{array}$

$a_1$

$a_2$

$k=2$

$\begin{array}{c}
\downarrow \\
\downarrow \\
\end{array}$

.$$

.$$

$k=3$

$\begin{array}{c}
\downarrow \\
\downarrow \\
\end{array}$

$\begin{array}{c}
\downarrow \\
\downarrow \\
\end{array}$

$am$

$x(1)$

$x(2)$

$x(3)$
**PSCD VS. SCD**

**Algorithm 1: Stochastic Coordinate Descent**

- Initialize:
  - $x = 0, z = 0$, step size $\alpha$
  - $S(z) = \begin{cases} 
    z & \text{for Lasso} \\
    1/(1 + \exp(-z)) & \text{for Logreg}
  \end{cases}$
  - $T(x_j, g_j) = \begin{cases} 
    \alpha g_j + \alpha \lambda & x_j - \alpha g_j > \alpha \lambda \\
    \alpha g_j - \alpha \lambda & x_j - \alpha g_j < -\alpha \lambda \\
    x_j & \text{else (to set } x_j = 0) 
  \end{cases}$

- for epoch = 1, 2, ..., do
  - randomly without replacement
    - for $j = \text{shuffle}(1, ..., n)$ do
      - partial gradient computation $g_j = \frac{1}{m} (S(z) - b) \cdot a_{i,j}$
      - $\mu = T(x_j, g_j)$ — thresholding due to regularization
      - $x_j = x_j - \mu$ — coordinate update
      - $z = z - \mu a_{i,j}$ — inner-product vector update

- for epoch = 1, 2, ..., do
  - randomly without replacement
    - for $k = 0, ..., K-1$ (each partition) do
      - subset $= kM + 1, ..., kM + M$
        - partial gradient computation
        - $g_j = (S(z_{\text{subset}}) - b_{\text{subset}}) \cdot a_{\text{subset},j}$
        - $\mu = T(x[k], g_j)$ — thresholding due to regularization
        - $x[k]_j = x[k]_j - \mu$
        - $z_{\text{subset}} = z_{\text{subset}} - \mu a_{\text{subset},j}$
      - global inner-product update with the averaged model

- if epoch mod $P$ then
  - $\bar{x} = (x[0] + ... + x[K - 1]) / K$
  - $z = 0$
  - for $k = 0, ..., K-1$ (each partition) do
    - subset $= kM + 1, ..., kM + M$
      - for $j = 1, ..., n$ do
        - $z_{\text{subset}} = z_{\text{subset}} + \bar{x}_j a_{\text{subset},j}$

---

When $K = 1$, $pSCD = SCD$!

When $P$ is smaller, $pSCD$ is more similar to $SCD$!

“Batched” version of $SCD$!
EFFICIENCY

Of pSCD
TRADE-OFF BETWEEN STATISTICAL VS HARDWARE EFFICIENCY

➤ Statistical efficiency = Convergence (How fast? How good?)
➤ Hardware efficiency = Data Processing Rate
For small enough $P$ i.e. 10, pSCD is competitive with SCD.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Validation Metric</th>
<th>SCD</th>
<th>pSCD</th>
<th>pSCD</th>
<th>pSCD</th>
<th>pSCD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data set: IM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epoch = 200</td>
<td></td>
<td>0.10154</td>
<td>0.10575</td>
<td>+4.15%</td>
<td>0.10380</td>
<td>+2.23%</td>
</tr>
<tr>
<td>Train Size = 266k</td>
<td>Log Loss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Size = 66k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition Size = 16k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data set: AEA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epochs = 5k</td>
<td>Log Loss</td>
<td>0.13531</td>
<td>0.25927</td>
<td>+91.61%</td>
<td>0.14972</td>
<td>+10.65%</td>
</tr>
<tr>
<td>Train Size = 32k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Size = 59k</td>
<td>Test Accuracy</td>
<td>96.17%</td>
<td>96.071%</td>
<td>-0.10%</td>
<td>96.109%</td>
<td>-0.06%</td>
</tr>
<tr>
<td>Partition Size = 16k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data set: KDD1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epochs = 1k</td>
<td>Log Loss</td>
<td>0.24672</td>
<td>0.24712</td>
<td>+0.16%</td>
<td>0.24701</td>
<td>+0.12%</td>
</tr>
<tr>
<td>Train Size = 391k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Size = 45k</td>
<td>Test AUC</td>
<td>0.91029</td>
<td>0.86880</td>
<td>-4.56%</td>
<td>0.90891</td>
<td>-0.15%</td>
</tr>
<tr>
<td>Partition Size = 16k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data set: KDD2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epochs = 1k</td>
<td>Log Loss</td>
<td>0.32285</td>
<td>0.32294</td>
<td>+0.03%</td>
<td>0.32286</td>
<td>+0.003%</td>
</tr>
<tr>
<td>Train Size = 131k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Size = 45k</td>
<td>Test AUC</td>
<td>0.61145</td>
<td>0.61144</td>
<td>-0.002%</td>
<td>0.61145</td>
<td>0%</td>
</tr>
<tr>
<td>Partition Size = 16k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STATISTICAL EFFICIENCY: HOW FAST DOES IT CONVERGE?

➤ Not much visual difference especially for small $P = 10$

**Figure 5:** Convergence of the Logreg loss for three data sets trained with either SCD or pSCD with varying $P$. 
HARDWARE EFFICIENCY: HOW FAST DOES IT PROCESS DATA?

➤ pSCD is consistently faster than SCD.

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/SCD is significantly faster than SGD with small batch size, and slightly faster than SGD with large batch size.
COMPARISON WITH SGD: CONVERGENCE OVER TIME

➤ Large batch SGD has a faster convergence rate than SCD/pSCD.

➤ We will show why we want to use pSCD rather than SGD on FPGA in the next section!

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 $P = 10$. 
DISCUSSION

What are some realistic scenarios that you can think of that you would use pSCD?
Integration into DBMS
Decompression & Decryption

Raw Data

Algorithm

ML
Decompression & Decryption
Decompression & Decryption

Raw Data

Algorithm

DBMS
Decompression & Decryption

Decoding

Raw Data

Algorithm

DBMS

ML

101 011
System Performance

Discussion: Why the decryption runtime decreased?
FPGA Acceleration
What is FPGA?

- Any Tasks On the fly
- Parallel Tasks On the fly
- One Task ~ hours
- One Task ~ months

CPU
GPU
FPGA
ASIC

Flexibility
Efficiency
Cost per unit
FPGA & CPU Comparison

- CPU
  - Decompress
  - Decrypt
- SCD
- Decompress Decrypt
- Memory
FPGA & CPU Comparison

Memory

SCD Engine (FPGA)
- Write Back Engine
- Fetch Engine
- Compute Engine
- Decompress & Decrypt

CPU
- SCD
- Decompress Decrypt

Memory
SCD Engine Architecture

SCD Engine (FPGA)

- Decompress & Decrypt
- Compute Engine
- Write Back Engine
- Fetch Engine

Memory

Decompress & Decrypt

a) Decompression Pipeline
b) Decryption Pipeline
SCD Engine Architecture
SCD Engine Array Architecture

Better utilization means better performance.
FPGA & CPU Comparison

Memory

SCD Engine (FPGA)

Write Back Engine

Fetch Engine

Compute Engine

Decompress & Decrypt

CPU

SCD

Decompress Decrypt

Memory
FPGA & CPU Comparison

Discussion: Why FPGA is a better solution?
FPGA & CPU Comparison - Runtime

CPU:

- Decompress & Decrypt
- SCG

CPU Runtime:

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
FPGA & CPU Comparison - Runtime

CPU:

FPGA: CE0

Decompress & Decrypt

SCG
FPGA & CPU Comparison - Runtime

CPU:
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

FPGA: CE0
0 1 2 3 0 2 2 3

CE1
4 5 6 7 4 5 6 7

Decompress & Decrypt  SCG
FPGA & CPU Comparison - Throughputs

CPU:
- Encoded Data
- Raw Data
- Result

FPGA:
- Encoded Data
- Raw Data
- Result

Decoding Throughput

SCD Throughput

Output Throughput
FPGA & CPU Comparison - Throughputs

CPU:
- Encoded Data
- Decoding Throughput
- Raw Data
- SCD Throughput
- Result
- Output Throughput

FPGA:
- Encoded Data
- Decoding Throughput
- Raw Data
- SCD Throughput
- Result
FPGA & CPU Comparison - Throughputs

CPU:
- Encoded Data → Decoding Throughput → Raw Data
- Raw Data → SCD Throughput → Result

FPGA:
- Encoded Data → Raw Data
- Raw Data → Result

Throughput SCD
Throughput Decoding
Throughput
FPGA & CPU Comparison - Throughputs

Bottleneck determines the overall throughput.
FPGA & CPU Performance

FPGA prevents the performance degradation.
FPGA & CPU Performance

More complex analytic workloads demand custom and flexible computational solution.

FPGA prevents the performance degradation.
Thank you!
Backup Slides
Decompression & Decryption

- Block Based Delta-Encoding
  - Find difference
  - Encode into different formats
Decompression & Decryption

- AES-256 CBC
  - Three major steps
  - CPU instruction support
FPGA & CPU Performance

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

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

Discussion: Can we improve performance by adding more computational power?