Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU? Synchronous or Asynchronous?

Yujing Ma, Florin Rusu, and Martin Torres
University of California Merced
ML/AI Golden Age

- Big training data
  - Millions – billions examples
  - Thousands – millions features (dimensions)

- Highly-parallel hardware
  - Multi-core CPUs with 20+ cores
  - GPUs with 1000s of cores
  - Huge memory (100s GBs – 1s TB on a server)
Stochastic Gradient Descent (SGD)

- Powers ML/AI golden age
- 100s of variants/implementations/papers
- Implemented by any single ML/AI system
  - Google Brain, Microsoft Project Adam, IBM System ML, Spark Mllib, etc.
- CPU & GPU implementations
  - Caffe, TensorFlow, MXNet, BIDMach, SINGA, Theano, Torch, etc.
SGD in Databases

- It is not so much about deep learning
  - Regression (linear, logistic)
  - Classification (SVM)
  - Recommendation (LMF)
- Mostly about training
  - Inside DB, close to data
  - Over joins or factorized databases
  - Compressed data, (compressed) large models
- Selection of optimization algorithm and hyper-parameters
  - BGD vs. SGD vs. SCD
Contribution: SGD Study on Highly-Parallel Architectures

Exploratory axes

- Data sparsity: dense, sparse
- Model update: sync, async

Performance axes

- Hardware efficiency
- Time to convergence (tc)
- Statistical efficiency

CPU vs GPU computing architecture
ML Training with Gradient Descent

\[ \min_{\mathbf{w} \in \mathbb{R}^d} \left\{ \Lambda(\mathbf{w}) = \sum_{(\mathbf{x}_i, y_i) \in \text{data}} f(\mathbf{w}, \mathbf{x}_i; y_i) \right\} \]

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha^{(k)} \nabla \Lambda(\mathbf{w}^{(k)}) \]

\( \Lambda(\mathbf{w}) \) is the loss
\( \nabla \Lambda(\mathbf{w}) = \left[ \frac{\partial \Lambda(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial \Lambda(\mathbf{w})}{\partial w_d} \right] \) is the gradient
\( \alpha^{(k)} \) is step size or learning rate
\( \mathbf{w}^{(0)} \) is the starting point (random)

- Convergence to minimum guaranteed for convex objective function

\[ f_{LR}(\mathbf{w}) = \log \left( 1 + e^{-y_i \mathbf{x}_i \cdot \mathbf{w}} \right) \]

\[ \frac{\partial f_{LR}(\mathbf{w})}{\partial w_j} = x_{ij} \left( -y_i \frac{e^{-y_i \mathbf{x}_i \cdot \mathbf{w}}}{1 + e^{-y_i \mathbf{x}_i \cdot \mathbf{w}}} \right) \]

\[ f_{SVM}(\mathbf{w}) = \max(0, 1 - y_i \mathbf{x}_i \cdot \mathbf{w}) \]

\[ \frac{\partial f_{SVM}(\mathbf{w})}{\partial w_j} = \begin{cases} -y_i x_{ij}, & \text{if } y_i \mathbf{x}_i \cdot \mathbf{w} < 1 \\ 0, & \text{otherwise} \end{cases} \]
Algorithm 1 Stochastic Gradient Descent (SGD)

Require:
- Training examples $\vec{X} \in \mathbb{R}^{N \times d}$ and their labels $\vec{Y} \in \mathbb{R}^N$
- Loss function $f$ and its gradient $\nabla f$
- Initial model $\vec{w} \in \mathbb{R}^d$ and step size $\alpha \in \mathbb{R}$
- Number of epochs $t$ and batch size $B$

1. for $k = 1$ to $t$ do
   
   OPTIMIZATION EPOCH

2. Select a random subset of $B$ examples $\vec{X}_k = \{\vec{x}_{i_1}, \ldots, \vec{x}_{i_B}\}$ and their labels $\vec{Y}_k = \{y_{i_1}, \ldots, y_{i_B}\}$

3. Compute gradient estimate: $\vec{g} \leftarrow \sum_{\vec{X}_k, \vec{Y}_k} \nabla f (\vec{w})$

4. Update model: $\vec{w} \leftarrow \vec{w} - \alpha \vec{g}$

5. end for

6. return $\vec{w}$
**(Mini-)Batch SGD**

**Algorithm 2 Batch SGD Optimization Epoch**

1. Compute gradient:
   
   ```
   for \( i = 1 \) to \( N \) do \( \vec{g} \leftarrow \vec{g} + \nabla f (\vec{w}; \vec{x}_i, y_i) \)
   ```

2. Update model:
   
   \( \vec{w} \leftarrow \vec{w} - \alpha \vec{g} \)

---

http://www.olehouse.org/mlclass/17_Large_Scale_Machine_Learning.html
Synchronous Parallel SGD

Algorithm 2 Batch SGD Optimization Epoch

1. Compute gradient:
   \[
   \text{for } i = 1 \text{ to } N \text{ do } \vec{g} \leftarrow \vec{g} + \nabla f (\vec{w}; \vec{x}_i, y_i)
   \]

2. Update model:
   \[
   \vec{w} \leftarrow \vec{w} - \alpha \vec{g}
   \]

Parallel execution on CPU and/or GPU as linear algebra kernels

http://www.olehouse.org/mlclass/17_Large_Scale_Machine_Learning.html
Algorithm 3 Incremental SGD Optimization Epoch

1. **for** $i = 1$ **to** $N$ **do**
2. Compute gradient estimate: $\hat{g} \leftarrow \nabla f (\hat{w}; \hat{x}_i, y_i)$
3. Update model: $\hat{w} \leftarrow \hat{w} - \alpha \hat{g}$
4. **end for**
Asynchronous Parallel SGD (Hogwild!)

Algorithm 3 Incremental SGD Optimization Epoch

1. for $i = 1$ to $N$ do in parallel
2. Compute gradient estimate: $\tilde{g} \leftarrow \nabla f (\tilde{w}; \tilde{x}_i, y_i)$
3. Update model: $\tilde{w} \leftarrow \tilde{w} - \alpha \tilde{g}$
4. end for

$$\frac{\partial f_{LR}(\tilde{w})}{\partial w_j} = x_{ij} \left(-y_i \frac{e^{-y_i \tilde{x}_i \cdot \tilde{w}}}{1 + e^{-y_i \tilde{x}_i \cdot \tilde{w}}} \right)$$

No synchronization or locks
Batch v Incremental

(Mini-)Batch (BGD)

\[
\hat{w}^{(k+1)} = \hat{w}^{(k)} - \alpha^{(k)} \nabla \Lambda (\hat{w}^{(k)})
\]

- Exact/accurate gradient computation
- One step for per batch/iteration
- Faster convergence close to minimum

Stochastic (SGD)

\[
\hat{w}^{(k+1)} = \hat{w}^{(k)} - \beta^{(k)} \nabla f (\hat{w}^{(k)}, \bar{x}_{\eta^{(k)}}; y_{\eta^{(k)}})
\]

- Approximate gradient at each data point
- One step per data point
- Faster convergence far from minimum

http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html
NUMA CPU Architecture
NUMA CPU v GPU

<table>
<thead>
<tr>
<th></th>
<th>NUMA</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU/MP cores</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>cores</td>
<td>14 per CPU</td>
<td>192 per MP</td>
</tr>
<tr>
<td>blocks</td>
<td>-</td>
<td>16 per MP</td>
</tr>
<tr>
<td>threads</td>
<td>28 per CPU</td>
<td>2048 per MP</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32+32 KB</td>
<td>48 KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 KB</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>L3 /shared</td>
<td>35 MB</td>
<td>48 KB</td>
</tr>
<tr>
<td>RAM/global</td>
<td>256 GB</td>
<td>12 GB</td>
</tr>
</tbody>
</table>

- CPU: 2 x Intel Xeon E5-2660 (14 cores, 28 threads)
- GPU: Tesla K80 (use only one multiprocessor, ~K40)
# Datasets & Tasks

<table>
<thead>
<tr>
<th>dataset</th>
<th>#examples</th>
<th>#features</th>
<th>#nnz/exp (avg)</th>
<th>size (s/d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>covtype</td>
<td>581,012</td>
<td>54</td>
<td>54 to 54 (54)</td>
<td>– / 485MB</td>
</tr>
<tr>
<td>w8a</td>
<td>64,700</td>
<td>300</td>
<td>0 to 114 (12)</td>
<td>4.4MB / 155MB</td>
</tr>
<tr>
<td>real-sim</td>
<td>72,309</td>
<td>20,958</td>
<td>1 to 3,484 (51)</td>
<td>87MB / 12.1GB</td>
</tr>
<tr>
<td>rcv1</td>
<td>677,399</td>
<td>47,236</td>
<td>4 to 1,224 (73)</td>
<td>1.2GB / 256GB</td>
</tr>
<tr>
<td>news</td>
<td>19,996</td>
<td>1,355,191</td>
<td>1 to 16,423 (455)</td>
<td>134MB / 217GB</td>
</tr>
</tbody>
</table>

## dense

## sparse

<table>
<thead>
<tr>
<th>dataset</th>
<th>MLP architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>covtype</td>
<td>54-10-5-2</td>
</tr>
<tr>
<td>w8a</td>
<td>300-10-5-2</td>
</tr>
<tr>
<td>real-sim</td>
<td>50-10-5-2</td>
</tr>
<tr>
<td>rcv1</td>
<td>50-10-5-2</td>
</tr>
<tr>
<td>news</td>
<td>300-10-5-2</td>
</tr>
</tbody>
</table>
Synchronous SGD Implementation

vector a = matrix-vector-product(data, model)
a = vector-vector-element-product(label, a)
a = vector-element-exponent(a)
vector b = vector-element-sum(1, a)
a = vector-vector-element-division(a, b)
a = vector-vector-element-product(a, -label)
gradient = matrix-vector-product(transpose(data), a)

ViennaCL (1.7.1) library kernels
• Same API for CPU and GPU
• Separate compilation for each architecture
Synchronous SGD Study (LR)

**dense**

**sparse**
## Synchronous SGD Results

<table>
<thead>
<tr>
<th>task</th>
<th>dataset</th>
<th>time-to-convergence (sec)</th>
<th>time-per-iteration (msec)</th>
<th>epochs</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>gpu</td>
<td>cpu-seq</td>
<td>cpu-par</td>
<td>gpu</td>
</tr>
<tr>
<td>LR</td>
<td>covtype</td>
<td>1.05</td>
<td>145.11</td>
<td>1.29</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>0.37</td>
<td>148.88</td>
<td>0.46</td>
<td>4.87</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>3.10</td>
<td>1,537.90</td>
<td>7.67</td>
<td>4.43</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>31.69</td>
<td>2,227.05</td>
<td>48.06</td>
<td>44.82</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>0.65</td>
<td>240.21</td>
<td>3.68</td>
<td>6.37</td>
</tr>
<tr>
<td>SVM</td>
<td>covtype</td>
<td>10.22</td>
<td>1,344.65</td>
<td>13.50</td>
<td>14.27</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>0.78</td>
<td>342.85</td>
<td>0.80</td>
<td>4.13</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>0.23</td>
<td>75.59</td>
<td>0.46</td>
<td>6.22</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>1.13</td>
<td>111.61</td>
<td>2.61</td>
<td>29.74</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>0.30</td>
<td>98.42</td>
<td>1.69</td>
<td>6.67</td>
</tr>
<tr>
<td>MLP</td>
<td>covtype</td>
<td>1.498</td>
<td>19,398</td>
<td>10,009</td>
<td>919</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>83.57</td>
<td>909</td>
<td>388</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>21.99</td>
<td>229</td>
<td>93.98</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>48.91</td>
<td>1,146</td>
<td>241</td>
<td>1,193</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>4.03</td>
<td>35.04</td>
<td>16.08</td>
<td>40.23</td>
</tr>
</tbody>
</table>
Asynchronous SGD Implementation (Hogwild!)

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access path</td>
<td>row-major round-robin (row-rr)</td>
</tr>
<tr>
<td></td>
<td>row-major chunking (row-ch)</td>
</tr>
<tr>
<td></td>
<td>column-major round-robin (col-rr)</td>
</tr>
<tr>
<td></td>
<td>column-major chunking (col-ch)</td>
</tr>
<tr>
<td>Model replication</td>
<td>kernel</td>
</tr>
<tr>
<td></td>
<td>block</td>
</tr>
<tr>
<td></td>
<td>thread</td>
</tr>
<tr>
<td></td>
<td>example</td>
</tr>
<tr>
<td>Data replication</td>
<td>no replication (no-rep)</td>
</tr>
<tr>
<td></td>
<td>k-wise replication (rep-2, rep-5, rep-10)</td>
</tr>
</tbody>
</table>

- NUMA CPU
- Extensive study in DimmWitted by Zhang and Re (PVLDB 2014)
- GPU
- Novel study
Map Hogwild! to GPU

**Algorithm 3** Incremental SGD Optimization Epoch

1. for $i = 1$ to $N$ do
2. Compute gradient estimate: $\bar{g} \leftarrow \nabla f (\bar{w}; x_i, y_i)$
3. Update model: $\bar{w} \leftarrow \bar{w} - \alpha \bar{g}$
4. end for

Copy data and model to GPU
While not converge do
Execute Hogwild! kernel
End while
Data Access Path
row-major round-robin (row-rr)
Data Access Path
row-major chunking (row-ch)

dense data

thread₁

thread₂

model

chunk size = 2

sparse data

thread₁

thread₂

model

row-ch
Data Access Path

column-major round-robin (col-rr)
Data Access Path
column-major chunking (col-ch)
Data Access Path Study

dense

sparse
Model Replication

kernel / thread / example

MP

block  block
shared  shared

Registers

L1 cache  Read-only

MP

block  block
shared  shared

Registers

L1 cache  Read-only

L2 cache

Global memory  Local memory
Model Replication

Diagram showing a block structure with MP (Manufacturing Process) blocks containing shared registers, L1 cache, and read-only sections. The diagram also includes an L2 cache and a global memory section.
Data Replication

- No replication (no-rep)
  - Thread 1
  - Thread 2

- K-wise replication (rep-2)
  - Thread 1
  - Thread 2

- Round-robin
Data Replication

- **No replication (no-rep)**
  - Thread 1
  - Thread 2

- **K-wise replication (rep-2)**
  - Thread 1
  - Thread 2

**Chunking**
Data Replication Study

dense

sparse
## Asynchronous SGD Results

<table>
<thead>
<tr>
<th>task</th>
<th>dataset</th>
<th>time-to-convergence (sec)</th>
<th>time-per-iteration (msec)</th>
<th>epochs</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>gpu</td>
<td>cpu-seq</td>
<td>cpu-par</td>
<td>gpu</td>
</tr>
<tr>
<td>LR</td>
<td>covtype</td>
<td>1.97</td>
<td>0.60</td>
<td>1.51</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>0.22</td>
<td>0.27</td>
<td>0.18</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>2.48</td>
<td>1.35</td>
<td>0.52</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>18.29</td>
<td>20.37</td>
<td>4.64</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>∞</td>
<td>5.47</td>
<td>∞</td>
<td>65</td>
</tr>
<tr>
<td>SVM</td>
<td>covtype</td>
<td>0.96</td>
<td>0.16</td>
<td>0.35</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>∞</td>
<td>0.54</td>
<td>1.89</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>3.46</td>
<td>1.82</td>
<td>1.28</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>10.25</td>
<td>22.71</td>
<td>7.57</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>∞</td>
<td>20.01</td>
<td>1.79</td>
<td>50</td>
</tr>
<tr>
<td>MLP</td>
<td>covtype</td>
<td>2,106</td>
<td>6,365</td>
<td>288</td>
<td>6,056</td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>495</td>
<td>1,284</td>
<td>986</td>
<td>635</td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>140</td>
<td>317</td>
<td>11.14</td>
<td>715</td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>352</td>
<td>724</td>
<td>34.47</td>
<td>8,326</td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>18.25</td>
<td>47.35</td>
<td>1.12</td>
<td>234</td>
</tr>
<tr>
<td>task</td>
<td>dataset</td>
<td>optimal configuration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td>covtype</td>
<td>col-rr + block + no-rep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>row-rr + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>row-ch + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>row-ch + kernel + no-rep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>row-rr + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVM</td>
<td>covtype</td>
<td>col-rr + block + no-rep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w8a</td>
<td>row-ch + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>real-sim</td>
<td>row-rr + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rcv1</td>
<td>row-rr + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>news</td>
<td>row-rr + kernel + rep-10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MLP Speedup (real-sim)

Speedup:
- cpu-seq/cpu-par: 2.46
- cpu-seq/gpu: 10.5
- cpu-par/gpu: 4.26

MLP architecture:
- 50-10-5-2
- 50-2304-2304-2
- 50-10752-10752-2

Speedup values:
- 73.29
- 4.38
- 26.29
Synchronous GPU v Asynchronous CPU
Speedup CPU v GPU
Conclusions

- **Synchronous SGD**
  - GPU is always faster than parallel CPU in time to convergence
  - Gap is larger for MLP (5X on average)
  - Results are on par or better than TensorFlow and BIDMach

- **Asynchronous SGD**
  - CPU always outperforms GPU in time to convergence, even when GPU has a speedup larger than 10X in hardware efficiency
  - Gap is higher than 5X on sparse data and deep nets

- **Synchronous GPU vs Asynchronous CPU**
  - The best is task- and dataset-dependent
  - CPU should not be discarded
  - GPU is more cost-effective alternative
Code:
https://github.com/YMA33/GradientDescent

Thank you.
Questions ???