Large Scale Matrix Factorization: Systems and Acceleration

Wei Tan
IBM T. J. Watson Research Center
wtan@us.ibm.com
http://github.com/cumf
http://researcher.ibm.com/person/us-wtan
Agenda

- Fei’s talk covers the formalism/theory/applications of MF
- My talk focuses on “how to make it fast, scalable and cost-efficient”
  - About computer architecture, systems, performance (not accuracy)

Agenda
- Matrix factorization, SGD and ALS (10 min)
- Parallelize and accelerate SGD and ALS (20 min)
- GPU accelerated SGD and ALS (20 min)
- Conclusion and QA (10 min)
Matrix Factorization

Ratings ($R$)

$m$ users × $n$ items

$X \approx X_u^T \Theta^T$

Users

Items
MF Explained using Recommender Systems

- **Input**: users ratings on some items
- **Output**: user/item features
- **Use**: predict missing ratings; use features for other tasks (e.g. clustering)

- **How**: factorize the rating matrix $R$ into

  \[ R \approx X \cdot \Theta^T \]

  and minimize the empirical lost:

  \[ J = \sum_{u,v} (r_{uv} - x_u^T \theta_v)^2 + \lambda (\sum_u n_{x_u} ||x_u||^2 + \sum_v n_{\theta_v} ||\theta_v||^2) \]
Matrix Factorization is a Key

**Recommender systems**
- Predict missing ratings
- Group similar users/items

**Complex network**
- Link prediction
- Vertices clustering

**Web search**
- Match query and document

**Natural language processing**
- Latent semantic model
- Word embedding as input to DNN

**Deep learning**
- Model compression
- Embedding layer

**Tensor decomposition**
- In machine learning and HPC applications

---

**Matrix Factorization**

\[
\mathbf{R} \approx \mathbf{X}^T \mathbf{X}
\]

- Supported in cuMF
- To be supported
Challenge: MF needs to be fast, scalable, economic

- **Fast**
  - Recommend/update timely

- **Scalable**
  - Facebook: 100 B ratings, 1 B users

- **Economic**
  - Avoid large infrastructure
To Solve MF: SGD [1]

\[ R \approx X \cdot \Theta^T \]

\[ J = \sum_{u,v} \left( r_{uv} - x_{u}^T \theta_{v} \right)^2 + \lambda \left( \sum_{u} n_{x_{u}} \| x_{u} \|^2 + \sum_{v} n_{\theta_{v}} \| \theta_{v} \|^2 \right) \]

Stochastic gradient descent (SGD)

\[ x_{u} = x_{u} - \alpha \left[ (x_{u}^T \theta_{v} - r_{uv}) \theta_{v} + \lambda x_{u} \right] \]

\[ \theta_{v} = \theta_{v} - \alpha \left[ (x_{u}^T \theta_{v} - r_{uv}) x_{u} + \lambda \theta_{v} \right] \]

- Update takes one rating at a time
- Vector inner product: memory bound
- Need many light epochs
- Parallelize: non-trivial
- Handle dense (implicit) ratings: no
To Solve MF: ALS [1]

\[ R \approx X \cdot \Theta^T \]

\[ J = \sum_{u,v} (r_{uv} - x_u^T \theta_v)^2 + \lambda \left( \sum_u n_{x_u} ||x_u||^2 + \sum_v n_{\theta_v} ||\theta_v||^2 \right) \]

Alternating Least Square (ALS)

\[ \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot x_u = \Theta^T \cdot R_u^T \]

\[ \sum_{r_{uv} \neq 0} (x_u x_u^T + \lambda I) \cdot \theta_v = X^T \cdot R_{uv} \]

- Update takes ALL rating at a time
- Vector outer product & solve: compute bound
- Need few heavy epochs
- Parallelize: straightforward
- Handle dense (implicit) ratings: yes
To Solve MF: CD [14]

\[ R \approx X \cdot \Theta^T \]

\[ J = \sum_{u,v} (r_{uv} - x_u^T \theta_v)^2 + \lambda(\sum u n_{x,u} ||x_u||^2 + \sum v n_{\theta,v} ||\theta_v||^2) \]

Coordinate descent (CD)
- Similar to ALS
- But update one coordinate of \(x_u\) and \(\theta_u\) at a time
Parallelize SGD: Hogwild! [15]

\[ x_u = x_u - \alpha [(x_u^T \theta_v - r_{uv}) \theta_v + \lambda x_u] \]
\[ \theta_v = \theta_v - \alpha [(x_u^T \theta_v - r_{uv}) x_u + \lambda \theta_v] \]

parallel SGD converges despite of (occasional) update conflict

update conflict!
Hogwild! Not Good Enough?

Random sampling hurts cache performance (esp. on CPUs) -- hardware cannot prefetch
Parallelize SGD: Matrix blocking [13]

<table>
<thead>
<tr>
<th></th>
<th>0. Divide to blocks</th>
<th>1. wave 1</th>
<th>2. wave 2</th>
<th>3. wave 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="Matrix" /></td>
<td><img src="https://via.placeholder.com/150" alt="Matrix" /></td>
<td><img src="https://via.placeholder.com/150" alt="Matrix" /></td>
<td><img src="https://via.placeholder.com/150" alt="Matrix" /></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

- Divide $R$ into blocks, say 4*4
- 4 workers update 4 “non-overlapping” blocks concurrently
  - Workers do not need to communicate
Parallelize SGD: Matrix blocking [9]

- Cons: all 4 workers need to complete before the next wave
- E.g., 5 can start only when 1-4 are all done.

- Solution: more blocks than workers – 6*6 blocks, 4 workers
- Worker 0 can immediate pick up another block when T0 is done
- Cons: scheduling overhead
To Parallelize ALS \[23\]

\[ R \approx X \cdot \Theta^T \]

\[ J = \sum_{u,v} (R_{uv} - x_u^T \theta_v)^2 + \lambda \left( \sum_u n_{x,u} \|x_u\|^2 + \sum_v n_{\theta,v} \|\theta_v\|^2 \right) \]

## Alternating Least Square (ALS)

\[
\sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{uv}^T
\]

\[
\sum_{r_{uv} \neq 0} (x_u x_u^T + \lambda I) \cdot \theta_v = X^T \cdot R_{uv}
\]

- Solve \(x_u\)s independently (\(\theta_v\)s thereafter)
- Parallelize the solve of \(x_u\)s on multiple nodes
  - Replicate \(\Theta\)
  - Partially replicate \(\Theta\)
  - Split \(\Theta\) on multiple nodes
# MF methods with SGD, ALS and CCD

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SGD</strong></td>
<td><strong>lock-free:</strong> workers independently sample &amp; update</td>
<td>single and multiple GPUs:</td>
</tr>
<tr>
<td></td>
<td>single-node: HogWild! [15]</td>
<td>cuMF.SGD [27] implements lock-free and</td>
</tr>
<tr>
<td></td>
<td>multi-nodes: FactorBird [8], Petuum [28]</td>
<td>blocking, and is memory-optimized with reduced precision.</td>
</tr>
<tr>
<td></td>
<td><strong>blocking:</strong> workers on non-overlapping blocks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blockDim=#workers: DSGD [13]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blockDim&gt;#workers: LIBMF [9], NOMAD [12], DSGD++ [18]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nested blocking: dcMF [29], MLGF-MF [16]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partial replicate: SparkALS [31], GraphLab [32], Sparkler [11]</td>
<td>single and multiple GPUs: cuMF [23]</td>
</tr>
<tr>
<td></td>
<td>rotate: Facebook [5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>approximate ALS: [33]</td>
<td></td>
</tr>
<tr>
<td><strong>CCD</strong></td>
<td>multi-core and multi node: CCD++ [14]</td>
<td></td>
</tr>
</tbody>
</table>
Challenge: compute and memory capacity of CPU

Table 2: Compute and memory complexity per epoch: ALS vs. SGD

<table>
<thead>
<tr>
<th></th>
<th>Compute (C)</th>
<th>Memory (M)</th>
<th>C/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>$\mathcal{O}(Nzf^2)$</td>
<td>$\mathcal{O}(Nzf + (m+n)f^2)$</td>
<td>$f$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{O}((m+n)f^3)$</td>
<td>$\mathcal{O}((m+n)f^2)$</td>
<td>$f$</td>
</tr>
<tr>
<td>SGD</td>
<td>$\mathcal{O}(Nzf)$</td>
<td>$\mathcal{O}(Nzf)$</td>
<td>1</td>
</tr>
</tbody>
</table>

CPU offers: 1 T flops, 80 GB/s

$f=100$, per epoch

- ALS floating-point operations
  - Netflix: 1.5 T
  - Hugewiki: 80 T
  - Facebook: 2000 T

- SGD memory transfer
  - Netflix: 80 GB
  - Hugewiki: 2.4 TB
  - Facebook: 80 TB

- >> CPU flops and BW capacity
GPU vs. CPU: compute FLOPS and memory bandwidth

- Raw performance: $1 \text{ GPU} \approx 10 \times \text{CPU}$
- Practical performance due to slow interconnection:
  - $1 \text{ GPU} > 10 \times \text{CPU}$
  - $4 \text{ GPU} \gg 40 \times \text{CPU}$

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
Goal: a CUDA library for MF

**cuMF**
Kernels for ALS and SGD

**CUDA**

---

**Fast**
- Fast training
- Update model quickly

---

**Scalable**
- Deal with big data
- Exploit fast interconnection

---

**Cost efficient**
- Fully utilize flops or BW
- Cheaper than CPU solutions

---

CUDA-based matrix factorization libraries.
CUDA-based matrix factorization libraries developed by IBM Research and friends.

- weitan@ieee.org
Challenges of ALS

- ALS needs to solve many:

\[
\sum_{r_{u,v} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{u*}^T
\]

- **Challenge 1:** access and aggregate many \( \theta_v \)'s: memory irregular and compute intensive

- **Challenge 2:** LU or Cholesky solver: compute intensive

- **Challenge 3:** Single GPU can NOT handle big \( m, n \) and \( N_z \)
Challenge 1: improve flops

- Nvidia Pascal: Memory BW: 740 GB/sec, compute: 11 Tflops
- Higher flops → higher op intensity (more flops per byte) → caching!

Address challenge 1: memory-optimized ALS

- To obtain

$$\sum_{r_{uv} \neq 0} (\theta_v \theta^T_p + \lambda I)$$

1. non-coalesced read

2. stage into smem

3. tile and aggregate in register
Address Challenge 2: exact solver is compute intensive

\[ \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{uv}^T \]

**Algorithm 1** The CG Solver.

1: procedure CGSOLVE(A, b, x, f_s, e)
2: \[ r = b - A \cdot x \]
3: \[ p = r \]
4: \[ r_{s_{old}} = r^T \cdot r \]
5: \[ \textbf{for } j = 1 : f_s \textbf{ do} \]
6: \[ a_p = A \cdot p \]
7: \[ \alpha = r_{s_{old}} / (p^T \cdot a_p) \]
8: \[ x = x + \alpha p \]
9: \[ r = r - \alpha p \]
10: \[ r_{s_{new}} = r^T \cdot r \]
11: \[ \textbf{if } \sqrt{r_{s_{new}}} < e \textbf{ then} \]
12: \[ \text{break} \]
13: \[ \textbf{end if} \]
14: \[ p = r + (r_{s_{new}} / r_{s_{old}}) p \]
15: \[ r_{s_{old}} = r_{s_{new}} \]
16: \[ \textbf{end for} \]
17: \[ \text{return } x \]
18: \[ \text{end procedure} \]

\( O(f^3) \quad O(f^2) \)

use \( f_s \ll f \)
Address Challenge 2: use CG solver

- Solver time: CG = \( \frac{1}{4} \) LU
- CG solver is memory- (instead of compute-) bound
  - CG w/ FP16 = \( \frac{1}{2} \) CG w/ FP32

**Algorithm 1** The CG Solver.

1. procedure CGSOLVE\((A, b, x, f_s, \epsilon)\)
2. \( r = b - A \cdot x \)
3. \( p = r \)
4. \( rs_{old} = r^T \cdot r \)
5. for \( j = 1 : f_s \) do
6. \( a_p = A \cdot p \)
7. \( \alpha = rs_{old} / (p^T \cdot a_p) \)
8. \( x = x + \alpha p \)
9. \( r = r - \alpha p \)
10. \( rs_{new} = r^T \cdot r \)
11. if \( \sqrt{rs_{new}} < \epsilon \) then
12. break
13. end if
14. \( p = r + (rs_{new} / rs_{old}) p \)
15. \( rs_{old} = rs_{new} \)
16. end for
17. return \( x \)
18. end procedure
Address Challenge 3: scale-up ALS on multiple GPUs

- **Model** parallel: solve a portion of the model

\[ \sum_{r_{u,v} \neq 0} (\theta_v \theta_u^T + \lambda I) \cdot x_{u,v} = \Theta^T \cdot R_{uv}^T \]
Address Challenge 3: scale-up ALS on multiple GPUs

- **Data parallel**: solve with a portion of the training data

\[
A_u = \sum_{r_{uv} \neq 0} \left( \theta_v \theta_u^T + \lambda I \right) = \sum_{i=1}^{p} \sum_{r_{uv} \neq 0} \left( \theta_v \theta_u^T + \lambda I \right)
\]
Recap: challenges of ALS

• ALS needs to solve many:

\[
\sum_{v \neq 0} r_{u,v} (\theta_v \Theta_v + \lambda I) \cdot x_u = \Theta^T \cdot R_{u*}^T
\]

• Challenge 1: access and aggregate many \( \theta_v \)s: memory irregular and compute intensive -- use register, smem and non-coalesced read

• Challenge 2: LU or Cholesky solver: compute intensive -- use approximate CG solver and FP16

• Challenge 3: Single GPU can NOT handle big \( m, n \) and \( Nz \) -- use model and data parallelism, and topology aware reduction
Connect cuMF to Spark MLlib

- Spark applications relying on mllib/ALS need no change
- Modified mllib/ALS detects GPU and offload maxtrix computation
- Leverage the best of Spark (scale-out) and GPU (scale-up)

\[ \sum_{r_{uv} \neq 0} (\theta_v \theta_u^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{u*}^T \]

https://github.com/IBMSparkGPU/CUDA-MLlib
Connect cuMF to Spark MLlib

- RDD on CPU: to distribute rating data and shuffle parameters
- Solver on GPU: to form and solve \[ \sum_{r_{uv} \neq 0} (\theta_v \theta_u^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{u*}^T \]
- Able to run on multiple nodes, and multiple GPUs per node

1 Power 8 node + 2 K40

shuffle

RDD RDD RDD RDD RDD ... RDD RDD RDD ...

CUDA kernel

GPU1

CUDA kernel

GPU2

shuffle

RDD RDD RDD RDD ...

CUDA kernel

GPU1

1 Power 8 node + 2 K40
Challenges of SGD: highly efficient update kernel

• Iterate over all ratings and do this in sequence:

\[ x_u = x_u - \alpha [(x_u^T \theta_v - r_{uv})\theta_v + \lambda x_u] \]
\[ \theta_v = \theta_v - \alpha [(x_u^T \theta_v - r_{uv})x_u + \lambda \theta_v] \]

• Memory bound

```
//read the sample
int index = get_index_of_sample();
float r = log(samples[index].r);
int u = ldg(samples[index].u);
int v = ldg(samples[index].v);

//read the p & q
int p_offset = u*x, q_offset = v*x;
float tmp_pl = half2float(p[p_offset + threadIdx.x]), tmp_p2 = half2float(p[p_offset + threadIdx.x + 32]);
float tmp_ql = half2float(q[q_offset + threadIdx.x]), tmp_q2 = half2float(q[q_offset + threadIdx.x + 32]);

//compute the update
float split = __half2float(p[p_offset + threadIdx.x] + q[q_offset + threadIdx.x]);
float split2 = __half2float(p[p_offset + threadIdx.x + 32] + q[q_offset + threadIdx.x + 32]);
float deriv = split - split2;
```
Challenges of SGD: how to parallelize

(a) Hogwild!

(b) Matrix Blocking
Experiment 1: the performance of MF systems

- cuMF_ALS w/ FP16 on Maxwell and Pascal
  - 1 GPU for Netflix and Yahoo
- LIBMF: 1 CPU w/ 40 threads
- NOMAD
  - 32 nodes for Netflix and Yahoo
  - 64 HPC nodes for Hugewiki
- 2-10x as fast
Experiment 2: ALS vs. SGD in cuMF (on Maxwell)

- ALS slightly slower than SGD on single GPU
- On big data set Hugewiki, ALS@4 GPU performs best -- SGD harder to parallel to multiple GPUs!
Experiment 3: is GPU (cuMF) cost efficient?

- cuMF_ALS @4 Maxwell
  ≈ 1/10 SparkALS @50 nodes
  ≈ $2.5/\text{hr} \rightarrow 1/10 \text{ of 50 nodes}
  ≈ 1\% \text{ of SparkALS’s cost}

<table>
<thead>
<tr>
<th>Baseline</th>
<th>baseline config</th>
<th>#nodes</th>
<th>price/node/hr</th>
<th>cuMF speed</th>
<th>cuMF cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOMAD</td>
<td>m3.xlarge</td>
<td>32</td>
<td>$0.27</td>
<td>10x</td>
<td>3%</td>
</tr>
<tr>
<td>SparkALS</td>
<td>m3.2xlarge</td>
<td>50</td>
<td>$0.53</td>
<td>10x</td>
<td>1%</td>
</tr>
<tr>
<td>Factorbird</td>
<td>c3.2xlarge</td>
<td>50</td>
<td>$0.42</td>
<td>6x</td>
<td>2%</td>
</tr>
</tbody>
</table>
Recap

- Fei’s talk covers the formalism/theory/applications of MF
- My talk focuses on “how to make it fast, scalable and cost-efficient”
  - About computer architecture, systems, performance (not accuracy)
- Agenda
  - Matrix factorization, SGD and ALS (10 min)
  - Parallelize and accelerate SGD and ALS (20 min)
  - GPU accelerated SGD and ALS (20 min)
  - Conclusion and QA (10 min)
Thank you, questions?

- Code: http://github.com/cuMF/
- Blog: http://ibm.biz/cumf-blog
- Contact: Wei Tan, wtan@us.ibm.com