CuMF: Large-Scale Matrix Factorization on Just One Machine with GPUs

Wei Tan, IBM T. J. Watson Research Center
wtan@us.ibm.com
Agenda

- Why accelerate recommendation (matrix factorization) using GPUs?
- What are the challenges?
- How cuMF tackles the challenges?
- So what is the result?
Why we need a fast and scalable recommender system?

- Recommendation is pervasive
- Drive 80% of Netflix’s watch hours
- Digital ads market in US: 37.3 billion
- Need: fast, scalable, economic
ALS (alternating least square) for collaborative filtering

- **Input**: users ratings on some items
- **Output**: all missing ratings
- **How**: factorize the rating matrix $R$ into

$$R \approx X \cdot \Theta^T$$

and minimize the lost on observed ratings:

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

- **ALS**: iteratively solve

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

together with:

$$\sum_{r_{uv} \neq 0} (x_u x_u^T + \lambda I) \cdot \theta_v = X^T \cdot R_{uv}$$
Matrix factorization/ALS is versatile

\[ X \]

\[ \Theta^T \]

\[ \theta_y \]

**Recommendation**

**Word embedding**

**Topic model**

*a very important algorithm to accelerate*
Challenges of fast and scalable ALS

• ALS needs to solve:

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

LU or Cholesky decomposition: cublas

spmm: cublas

• Challenge 1: access and aggregate many \( \theta_v \)s: irregular (\( R \) is sparse) and memory intensive

• Challenge 2: Single GPU can NOT handle **big** \( m \), \( n \) and \( nnz \)
Challenge 1: memory access

- Nvidia K40: Memory BW: 288 GB/sec, compute: 4.3 Tflops/sec
- Higher flops → higher op intensity (more flops per byte) → caching!
Address challenge 1: memory-optimized ALS

- To obtain
  \[ \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \]

1. Reuse \( \theta_v \)'s for many users
2. Load a portion into smem
3. Tile and aggregate in register
Address challenge 1: memory-optimized ALS

\[ \sum_{t_{uv} \neq 0} (\theta_v T_{uv}^T + \lambda I) \]
Address challenge 2: scale-up ALS on multiple GPUs

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

\[
\sum_{r_{uv} \neq 0} (\theta_v \theta_u^T + \lambda I) \cdot x_{uv} = \Theta^T \cdot R_{uv}^T
\]
Address challenge 2: scale-up ALS on multiple GPUs

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

\[
A_u = \sum_{r_{uv} \neq 0} (\theta_v \theta_u^T + \lambda I) = \sum_{i=1}^{p} \sum_{r_{uv} \neq 0} (\theta_v \theta_u^T + \lambda I)
\]
Address challenge 2: parallel reduction

- Data parallel needs cross-GPU reduction

\[ A_u = \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) = \sum_{i=1}^{p} \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \]

One-phase parallel reduction.

Two-phase parallel reduction.

Intra-socket -- -- -- Inter-socket
Recap: cuMF tackled two challenges

- ALS needs to solve:

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

- Challenge 1: access and aggregate many \( \theta_v \)'s: irregular (\( R \) is sparse) and memory intensive

- Challenge 2: Single GPU can NOT handle big \( m \), \( n \) and \( nnz \)

LU or Cholesky decomposition: cublas

spmm: cublas
Connect cuMF to Spark MLlib

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

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

- In C (circa. 10k LOC)
- CUDA 7.0/7.5, GCC OpenMP v3.0

Baselines:
- Libmf: SGD on 1 node [RecSys14]
- NOMAD: SGD on >1 nodes [VLDB 14]
- SparkALS: ALS on Spark
- FactorBird: SGD + parameter server for
- Facebook: enhanced Giraph

<table>
<thead>
<tr>
<th>Data Set</th>
<th>(m)</th>
<th>(n)</th>
<th>(N_z)</th>
<th>(f)</th>
<th>(\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netflix</td>
<td>480,189</td>
<td>17,770</td>
<td>99M</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>YahooMusic</td>
<td>1,000,990</td>
<td>624,961</td>
<td>252.8M</td>
<td>100</td>
<td>1.4</td>
</tr>
<tr>
<td>Hugewiki</td>
<td>50,082,603</td>
<td>39,780</td>
<td>3.1B</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>SparkALS</td>
<td>660M</td>
<td>2.4M</td>
<td>3.5B</td>
<td>10</td>
<td>0.05</td>
</tr>
<tr>
<td>Factorbird</td>
<td>229M</td>
<td>195M</td>
<td>38.5B</td>
<td>5</td>
<td>0.05</td>
</tr>
<tr>
<td>Facebook</td>
<td>1B</td>
<td>48M</td>
<td>112B</td>
<td>16</td>
<td>0.05</td>
</tr>
<tr>
<td>cuMF</td>
<td>1B</td>
<td>48M</td>
<td>112B</td>
<td>100</td>
<td>0.05</td>
</tr>
</tbody>
</table>
CuMF performance

- 1 GPU vs. 30 cores, CuMF slightly faster than libmf and NOMAD
- CuMF scales well on 1, 2, 4 GPUs

**X-axis:** time in seconds; **Y-axis:** Root Mean Square Error (RMSE) on test set
Effectiveness of memory optimization

- Aggressively using registers $\rightarrow 2x$
- Using texture $\rightarrow 25\%-35\%$ faster

**X-axis**: time in seconds; **Y-axis**: Root Mean Square Error (RMSE) on test set
CuMF performance and cost

- CuMF @4 GPUs ≈ NOMAD @64 HPC nodes ≈ 10x NOMAD @32 AWS nodes

- CuMF @4 GPUs ≈ 10x SparkALS @50 nodes ≈ 1% of its 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>
CuMF accelerated Spark on Power 8

- CuMF @2 K40s achieves $6+x$ speedup in training (193 sec $\rightarrow$ 1.3k sec)

*GUI designed by Amir Sanjar
Conclusion

- **Why** accelerate recommendation (matrix factorization) using GPU?
  - Need to be fast, scalable and economic

- **What** are the challenges?
  - Memory access, scale to multiple GPUs

- **How** cuMF tackles the challenges?
  - Optimize memory access, parallelism and communication

- **So what** is the result?
  - Up to 10x as fast, 100x as cost-efficient
  - Use cuMF standalone or with Spark
  - GPU can tackle ML problems beyond deep learning!
Thank you, questions?


Source code available soon.

Wei Tan, IBM T. J. Watson Research Center wtan@us.ibm.com http://ibm.biz/wei_tan