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

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

Agenda

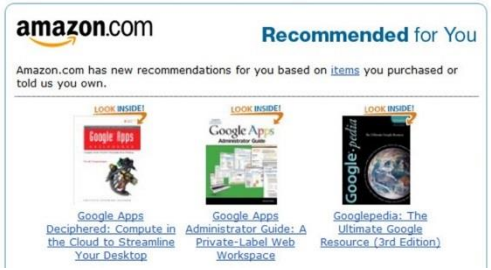
- **Why** accelerate recommendation (matrix factorization) using GPUs?
- **What** are the challenges?
- **How** cuMF tackles the challenges?
- **So what** is the result?

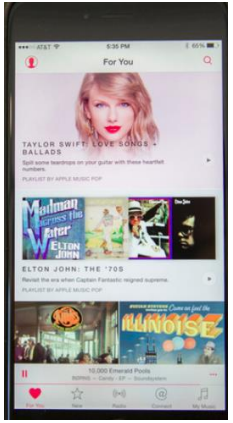
Why we need a fast and scalable recommender system?

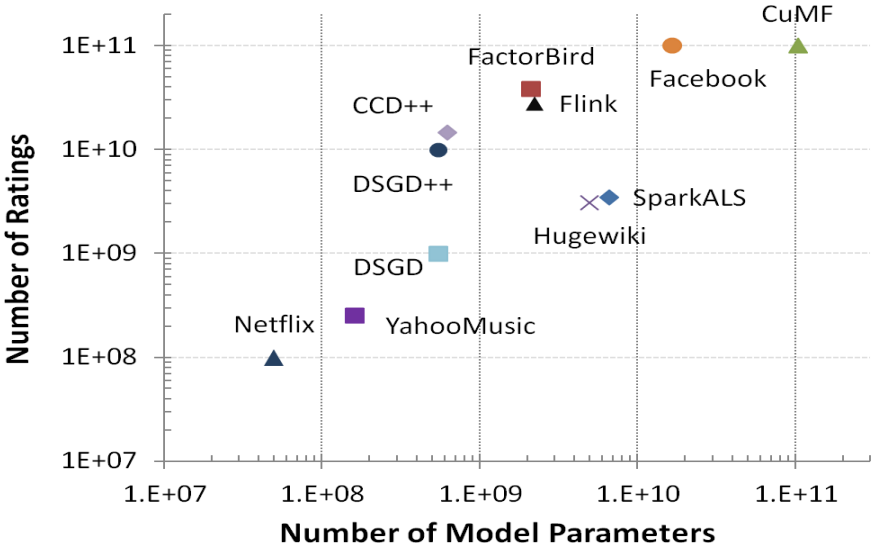
- Recommendation is pervasive
- Drive 80% of Netflix's watch hours[1]
- Digital ads market in US: 37.3 billion[2]
- Need: fast, scalable, economic

2			4	5	2.94*
5		4			1
		5		2	2.48*
	1		5		4
		4			2
4	5		1		1.12*

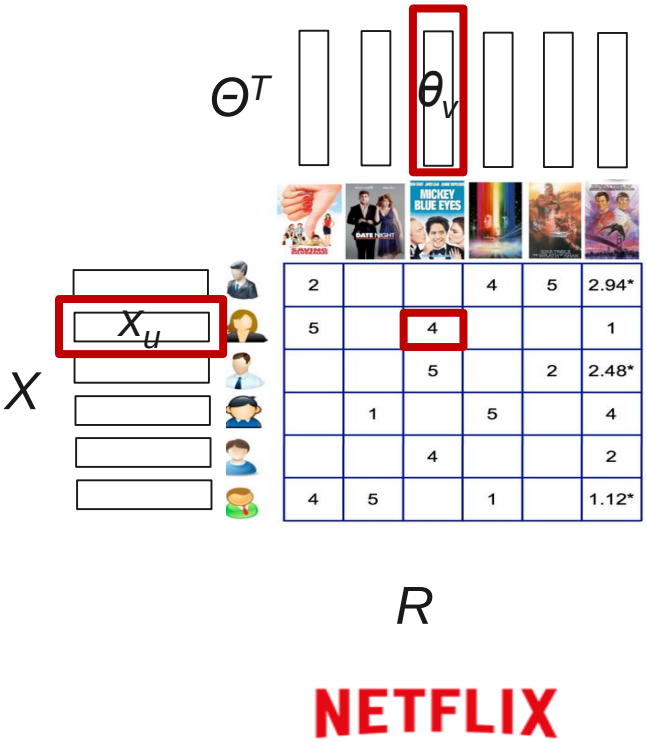
NETFLIX







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:

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

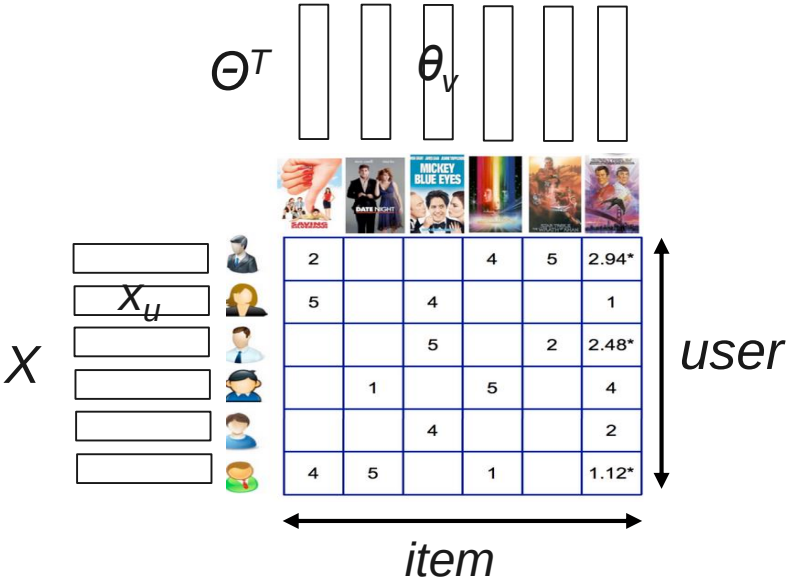
- ALS: iteratively solve

$$\sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot \mathbf{x}_u = \Theta^T \cdot R_{u*}^T$$

together with:

$$\sum_{r_{uv} \neq 0} (\mathbf{x}_u \mathbf{x}_u^T + \lambda I) \cdot \theta_v = X^T \cdot R_{*v}$$

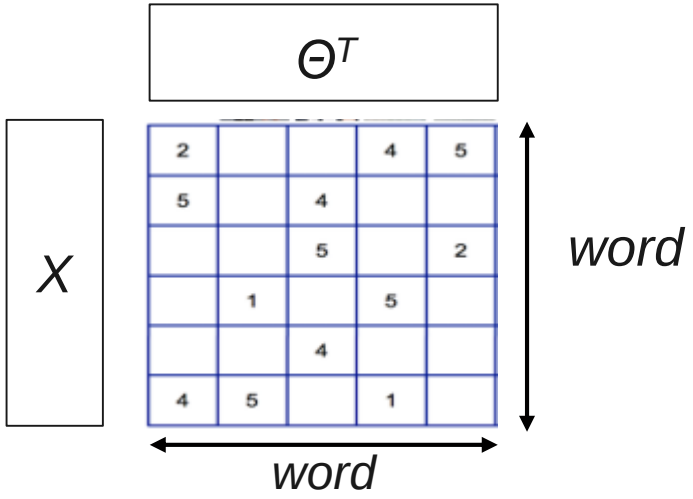
Matrix factorization/ALS is versatile



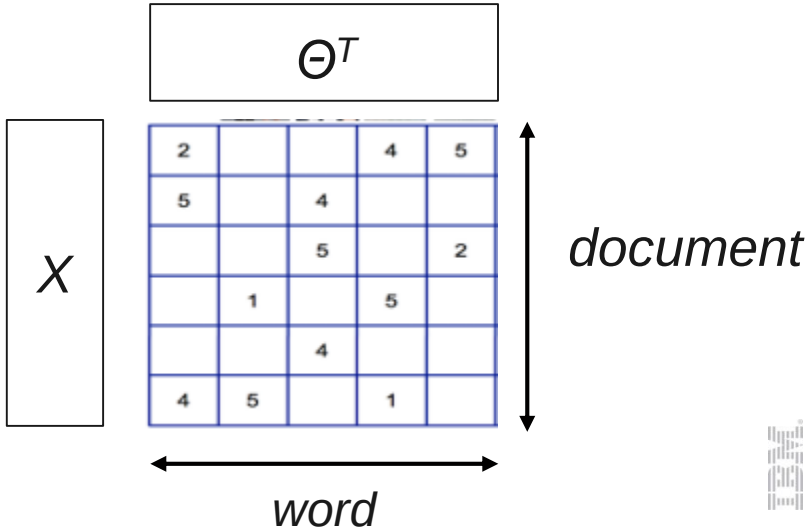
Recommendation

a very important algorithm to accelerate

Word embedding



Topic model



Challenges of fast and scalable ALS

- ALS needs to solve:

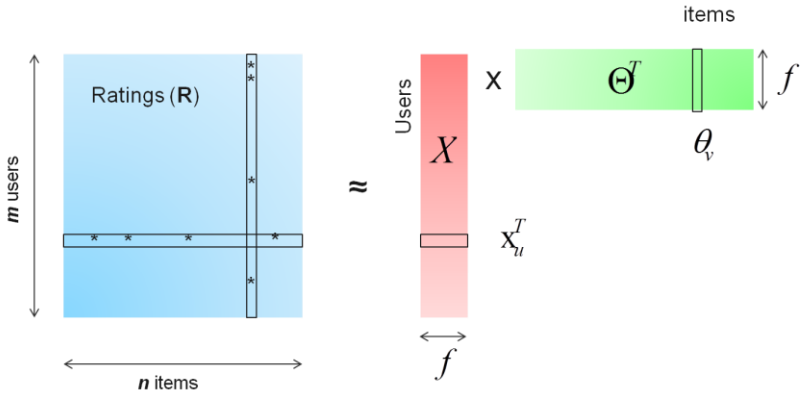
LU or Cholesky decomposition: cublas

$$\boxed{\sum_{r_{uv}\neq 0} (\theta_v \theta_v^T + \lambda I)} \cdot \mathbf{x}_u = \Theta^T \cdot R_{u*}^T$$

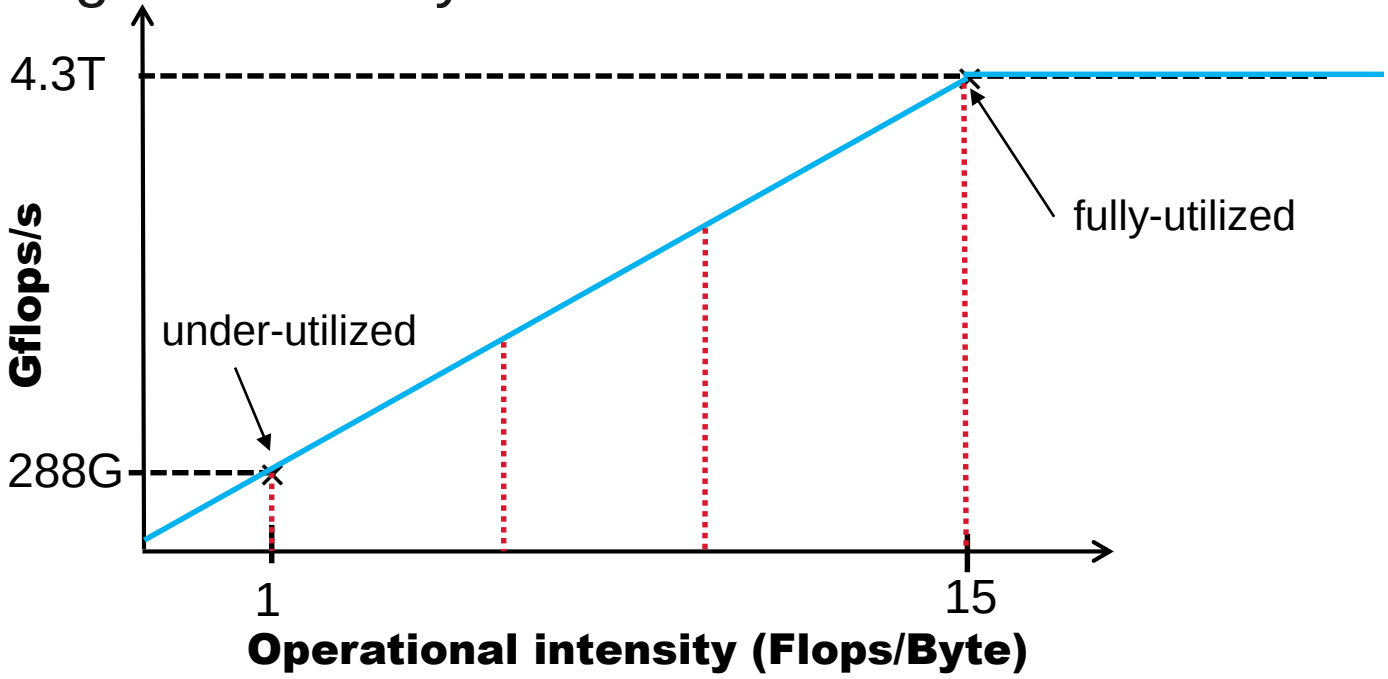
spmm: cublas

- Challenge 1: access and aggregate many θ_vs: **irregular** (*R* is sparse) and memory **intensive**

- Challenge 2:
  Single GPU can NOT handle **big** *m*, *n* and *nnz*

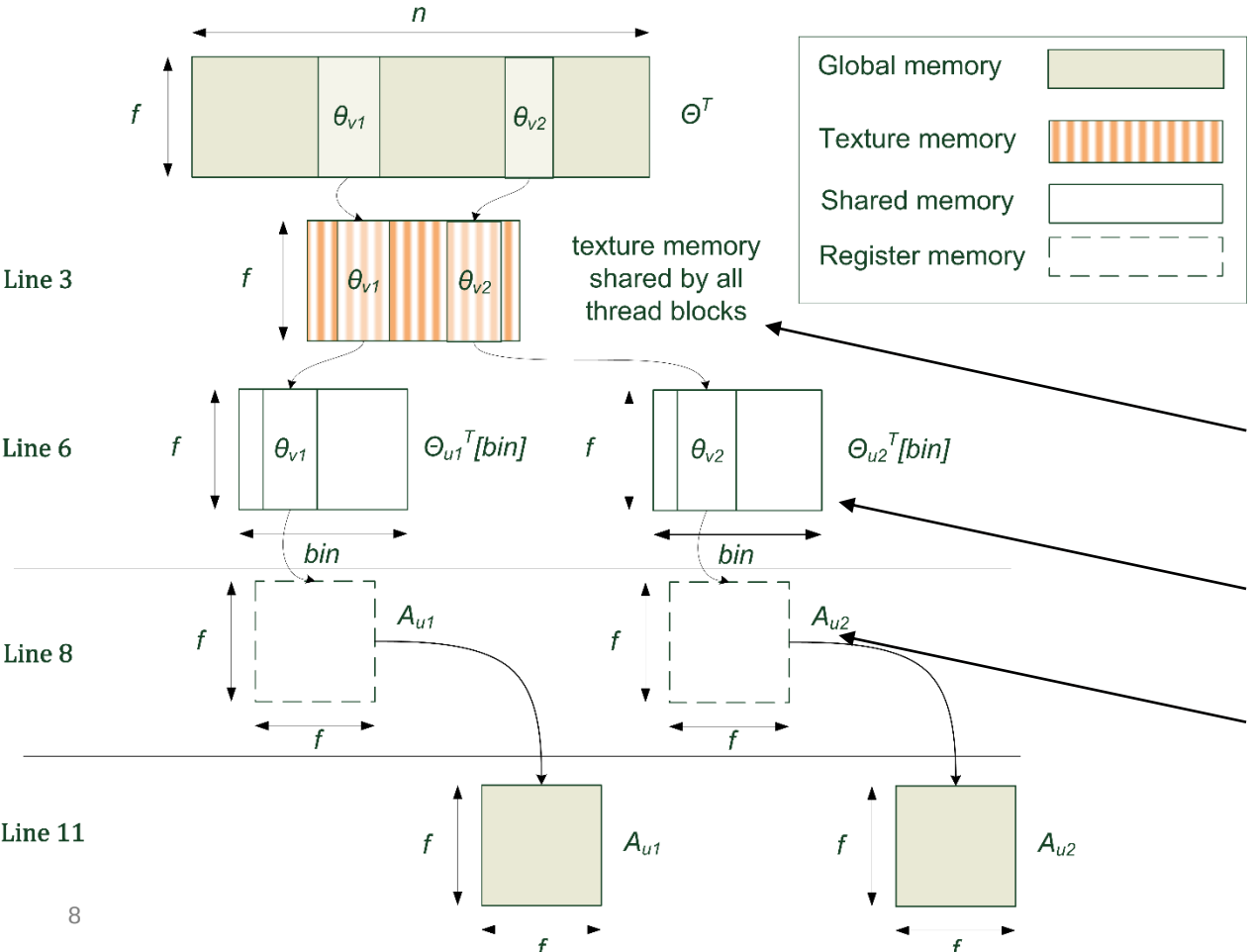


Challenge 1: memory access



- 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



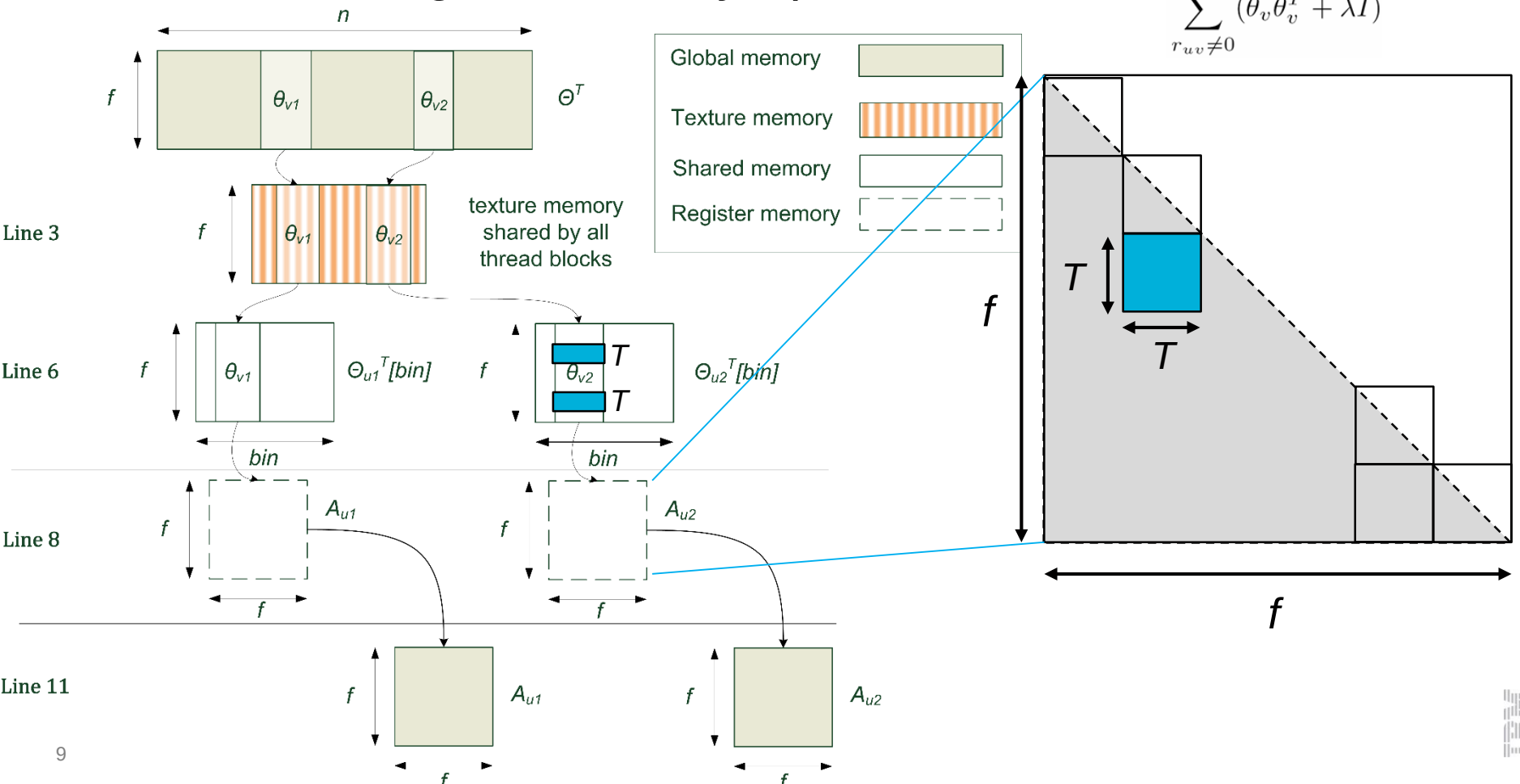
- To obtain

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

1. Reuse θ_vs for many users
2. Load a portion into smem
3. Tile and aggregate in register

Address challenge 1: memory-optimized ALS

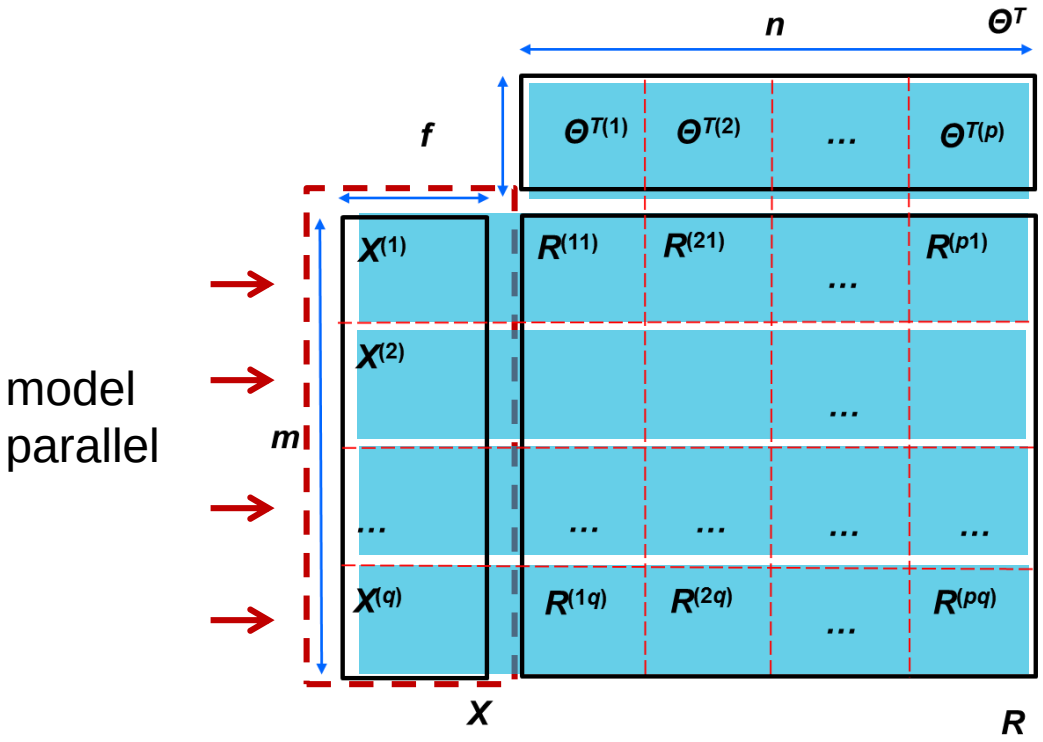
$$\sum_{r_{uv}\neq 0} (\theta_v \theta_v^T + \lambda I)$$



Address challenge 2: scale-up ALS on multiple GPUs

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

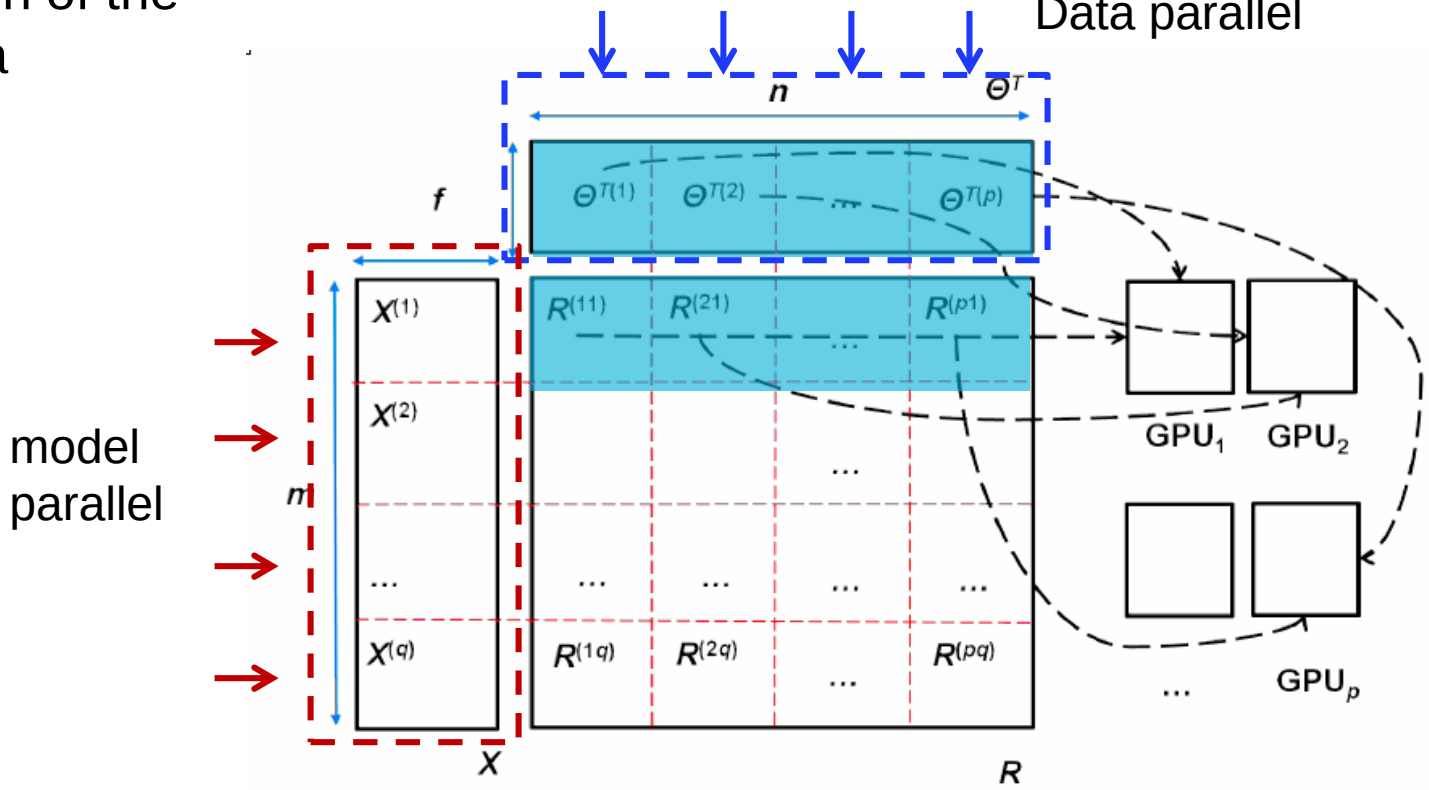
$$\sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot \mathbf{x}_u = \Theta^T \cdot R_{u*}^T$$



Address challenge 2: scale-up ALS on multiple GPUs

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

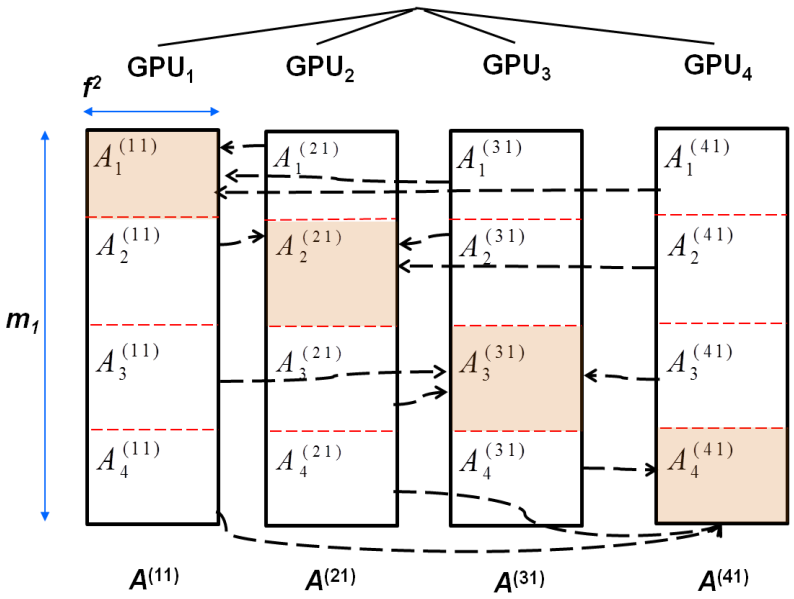
$$A_u = \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) = \sum_{i=1}^{p} \sum_{r_{uv} \neq 0}^{GPU_i} (\theta_v \theta_v^T + \lambda I)$$



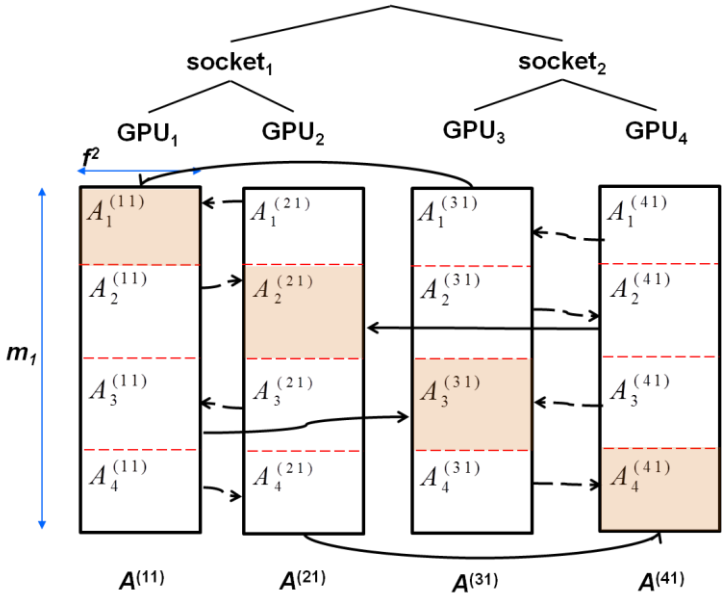
Address challenge 2: parallel reduction

- Data parallel needs cross-GPU reduction

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



One-phase parallel reduction.



Two-phase parallel reduction.

Intra-socket - - -> Inter-socket ——>

Recap: cuMF tackled two challenges

- ALS needs to solve:

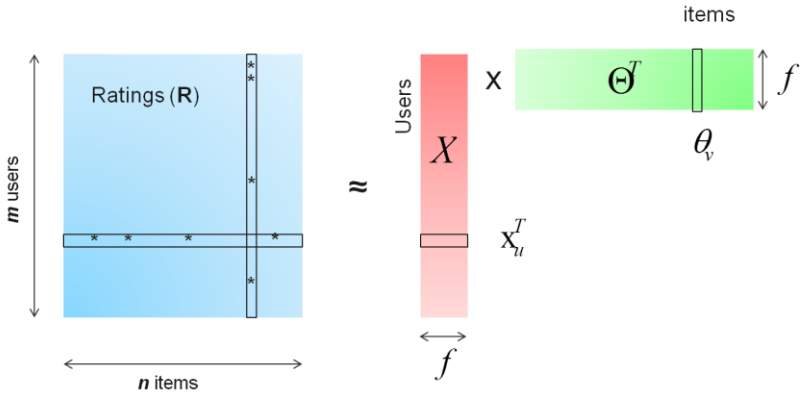
$$\boxed{\sum_{r_{uv}\neq 0} (\theta_v \theta_v^T + \lambda I)} \cdot \mathbf{x}_u = \Theta^T \cdot R_{u*}^T$$

LU or Cholesky decomposition: cublas

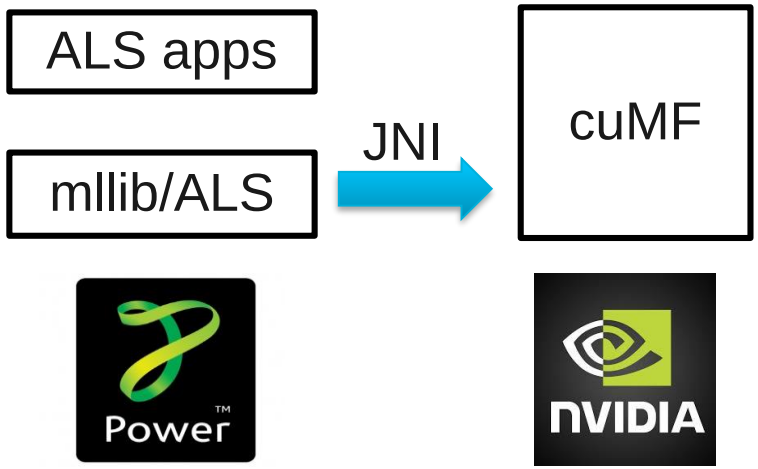
spmm: cublas

- Challenge 1: access and aggregate many θ_vs: **irregular** (*R* is sparse) and memory **intensive**

- Challenge 2:
  Single GPU can NOT handle **big** *m*, *n* and *nnz*



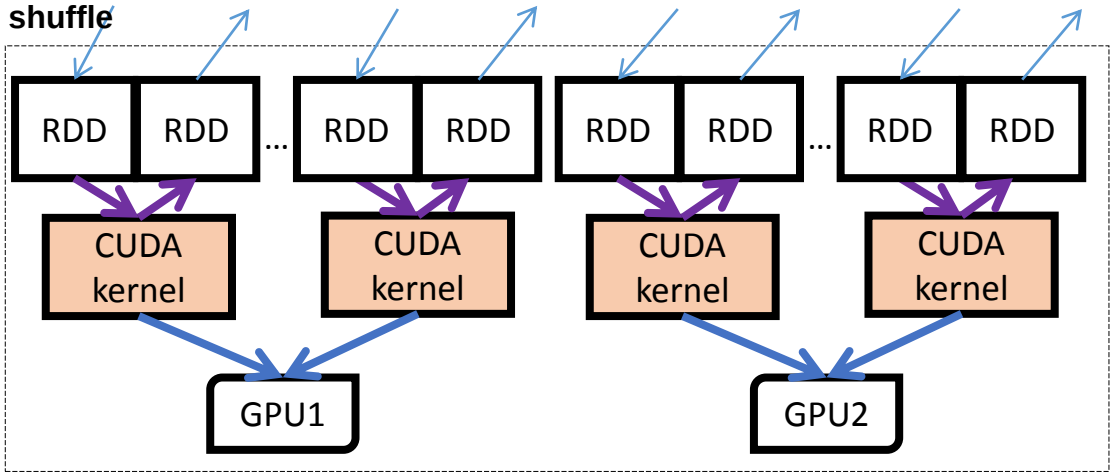
Connect cuMF to Spark MLlib



- 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)

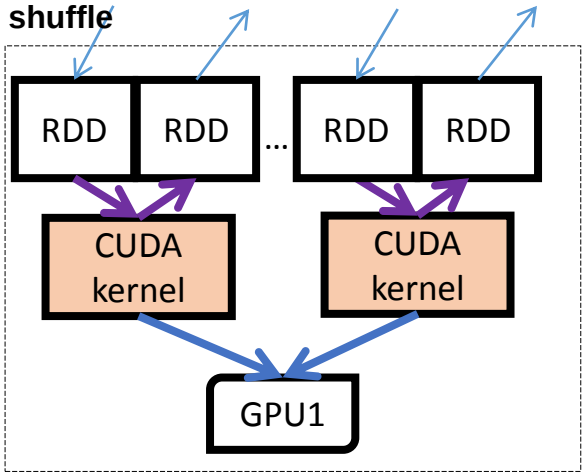
Connect cuMF to Spark MLlib

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



1 Power 8 node + 2 K40

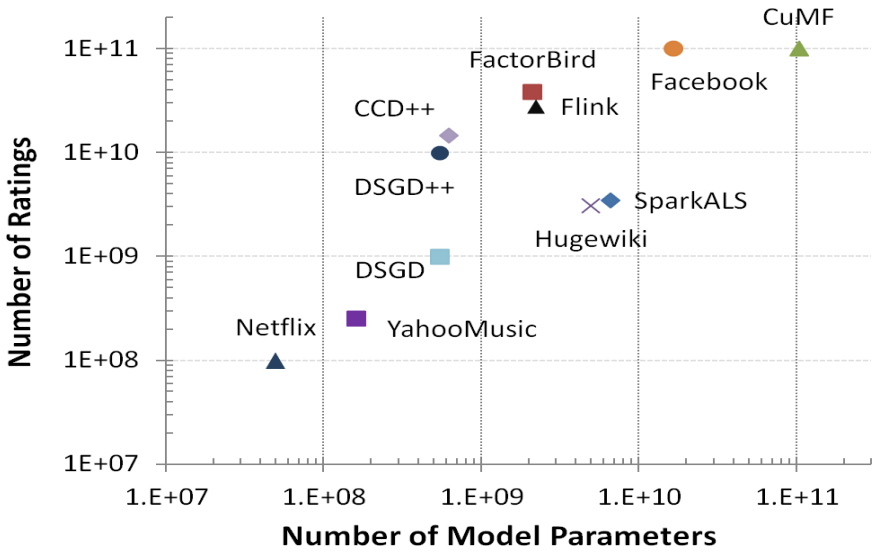
...



1 Power 8 node + 2 K40

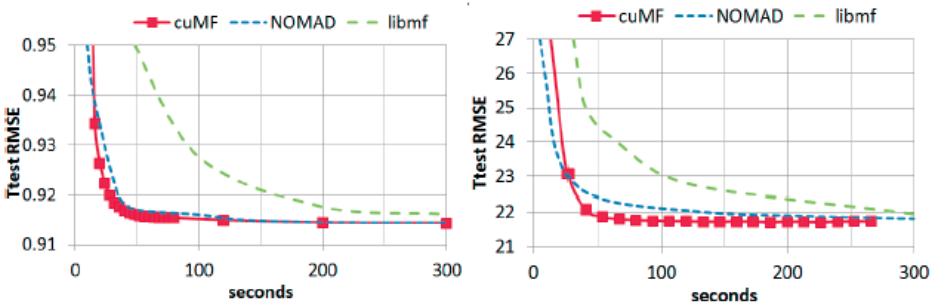
Implementation

- 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



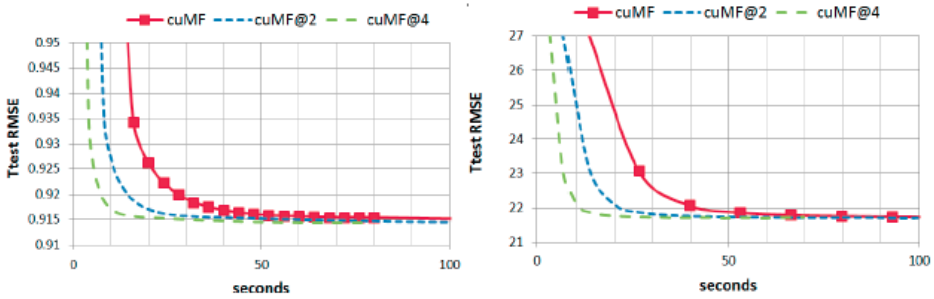
Data Set	m	n	N_z	f	λ
Netflix	480,189	17,770	99M	100	0.05
YahooMusic	1,000,990	624,961	252.8M	100	1.4
Hugewiki	50,082,603	39,780	3.1B	100	0.05
SparkALS	660M	2.4M	3.5B	10	0.05
Factorbird	229M	195M	38.5B	5	0.05
Facebook	1B	48M	112B	16	0.05
cuMF	1B	48M	112B	100	0.05

CuMF performance



(a) Netflix (b) YahooMusic

- 1 GPU vs. 30 cores, CuMF slightly faster than libmf and NOMAD

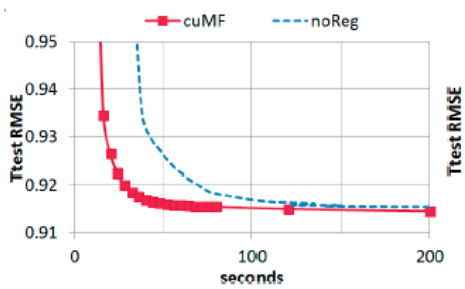


(a) Netflix (b) YahooMusic

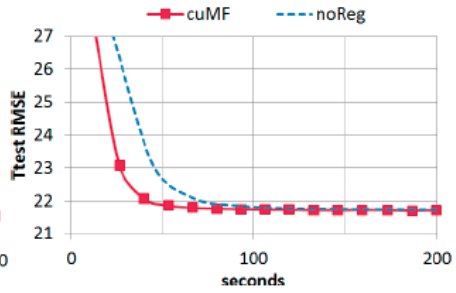
- CuMF scales well on 1, 2, 4 GPUs

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

Effectiveness of memory optimization

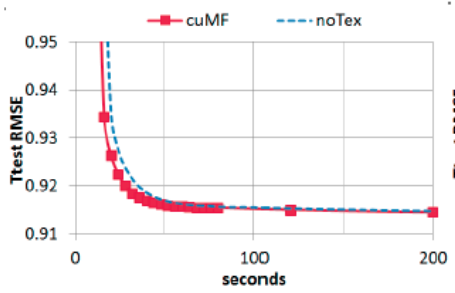


(a) Netflix

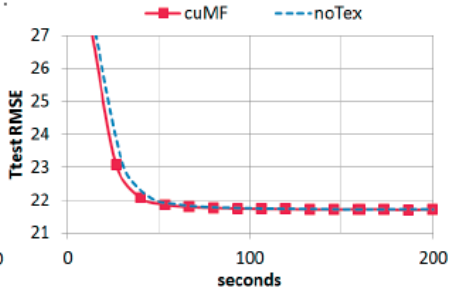


(b) YahooMusic

- Aggressively using registers → 2x



(a) Netflix

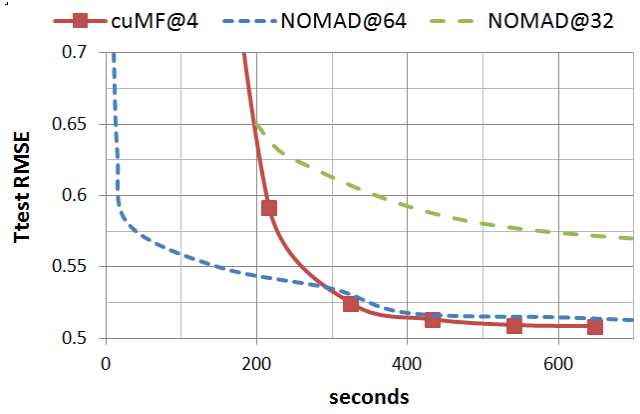


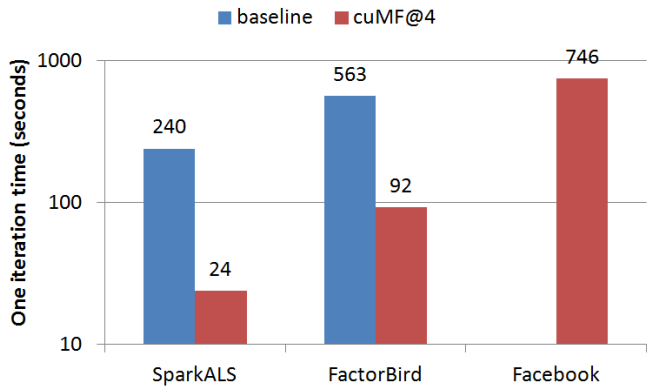
(b) YahooMusic

- Using texture → 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

Baseline	baseline config	#nodes	price /node/hr	cuMF speed	cuMF cost
NOMAD	m3.xlarge	32	$0.27	**10x**	**3%**
SparkALS	m3.2xlarge	50	$0.53	**10x**	**1%**
Factorbird	c3.2xlarge	50	$0.42	**6x**	**2%**

CuMF accelerated Spark on Power 8

- CuMF @2 K40s achieves **6+x** speedup in training (193 sec → 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?

Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs.
Wei Tan, Liangliang Cao, Liana Fong. **HPDC 2016**,
http://arxiv.org/abs/1603.03820

Source code available soon.

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