Large Scale Matrix Factorization: Systems and Acceleration

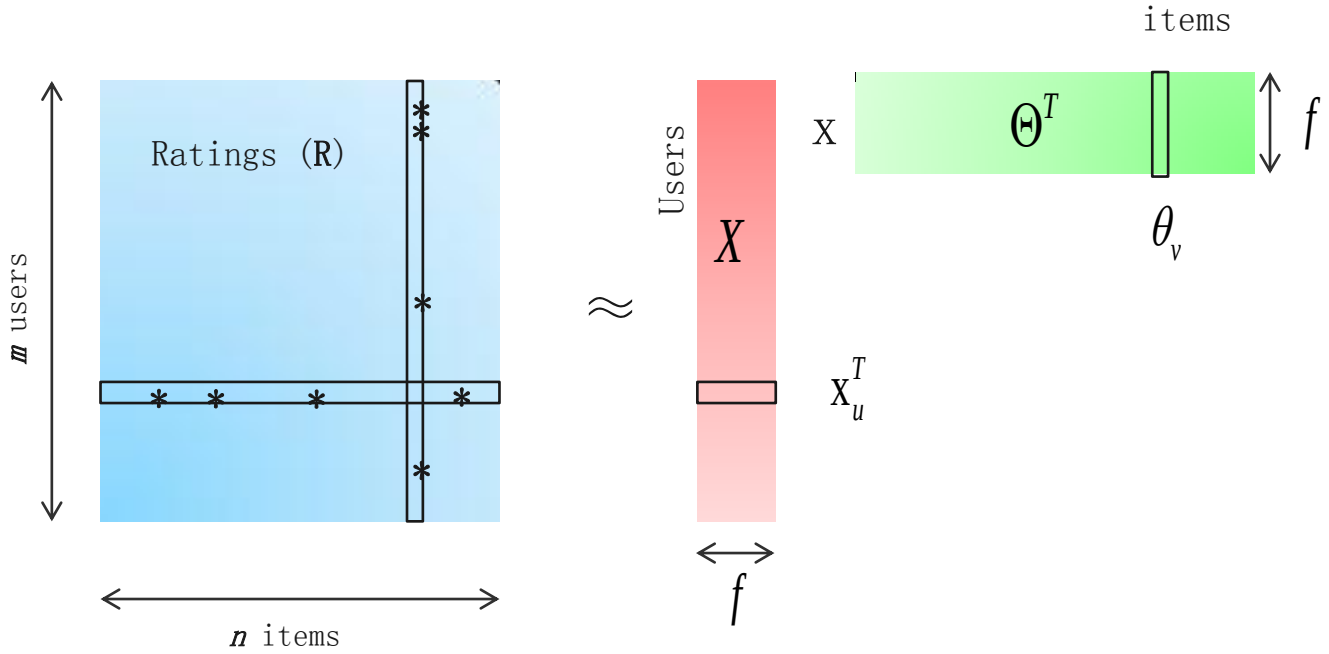
Wei Tan
IBM T. J. Watson Research Center
wtan@us.ibm.com
http://github.com/cumf
http://researcher.ibm.com/person/us-wtan

IBM

Agenda

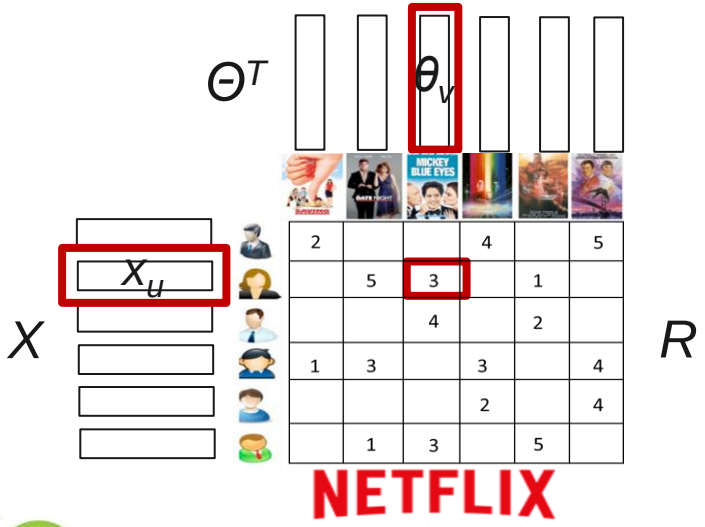
- **Fei's talk covers the formalism/theory/math of MF**
- **My talk focuses on "how to run it fast, scalable and cost-efficient"**
  - Matrix factorization, SGD and ALS (10 min)
  - Parallelize and accelerate SGD and ALS (20 min)
  - GPU accelerated SGD and ALS (20 min)
  - Conclusion and QA (10 min)

Matrix Factorization



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)



NETFLIX

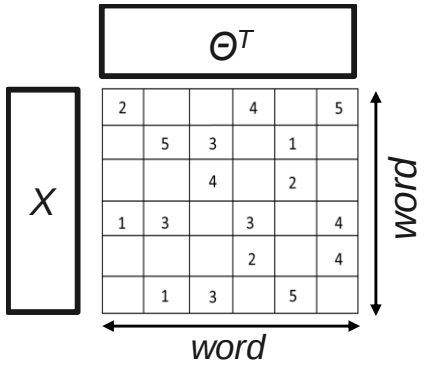
Spotify amazon.com Quora Music

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

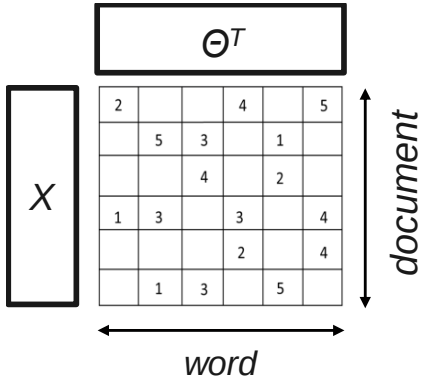
$$R \approx X \cdot \Theta^T$$

and minimize the empirical lost:

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

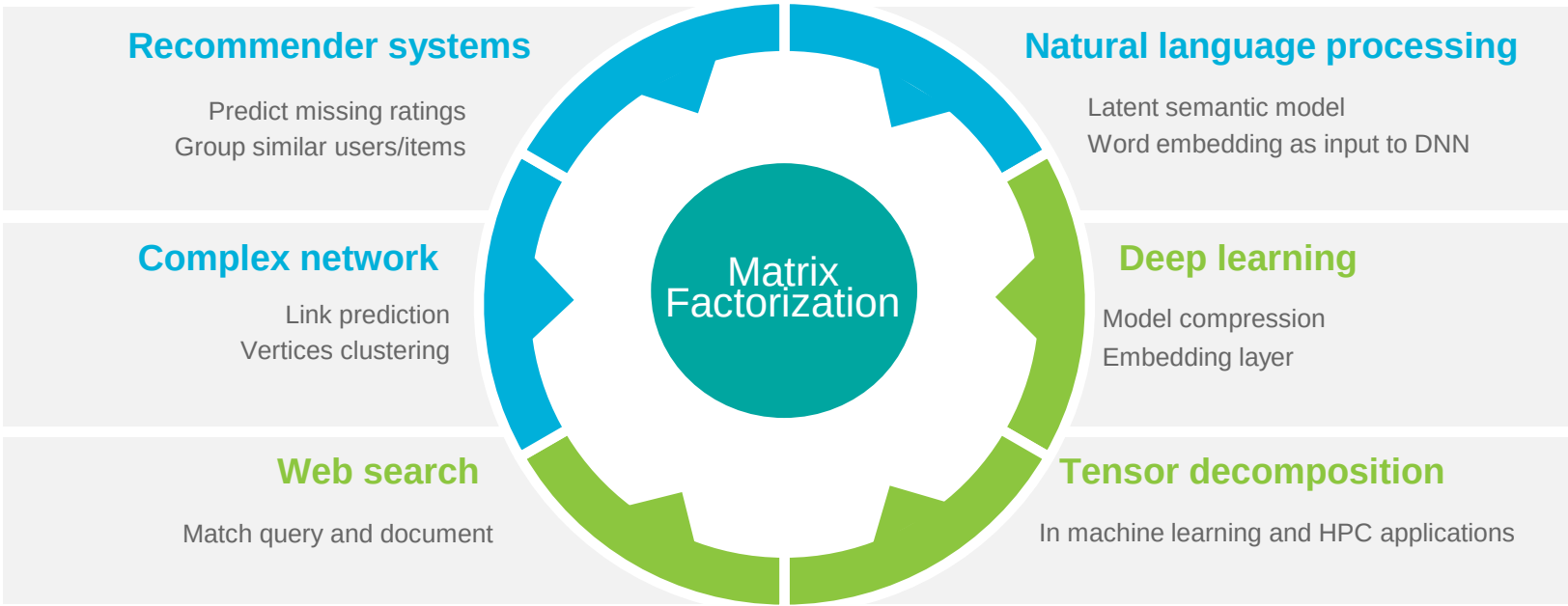


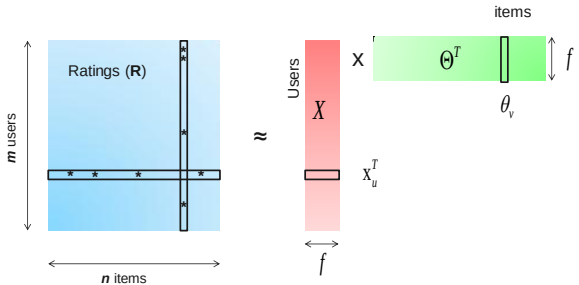
Word embedding

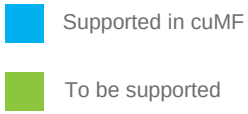


Topic model

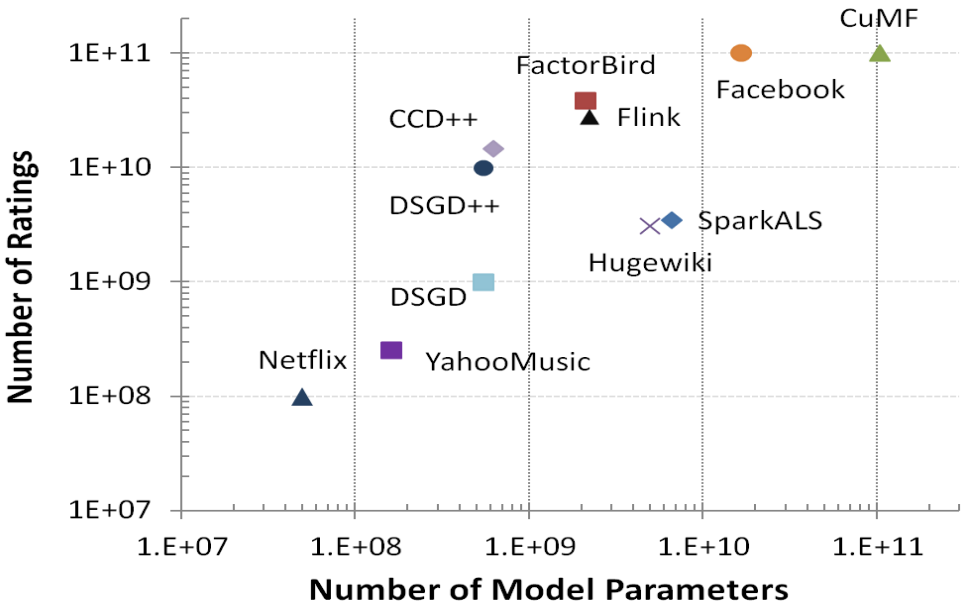
Matrix Factorization is a Key







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

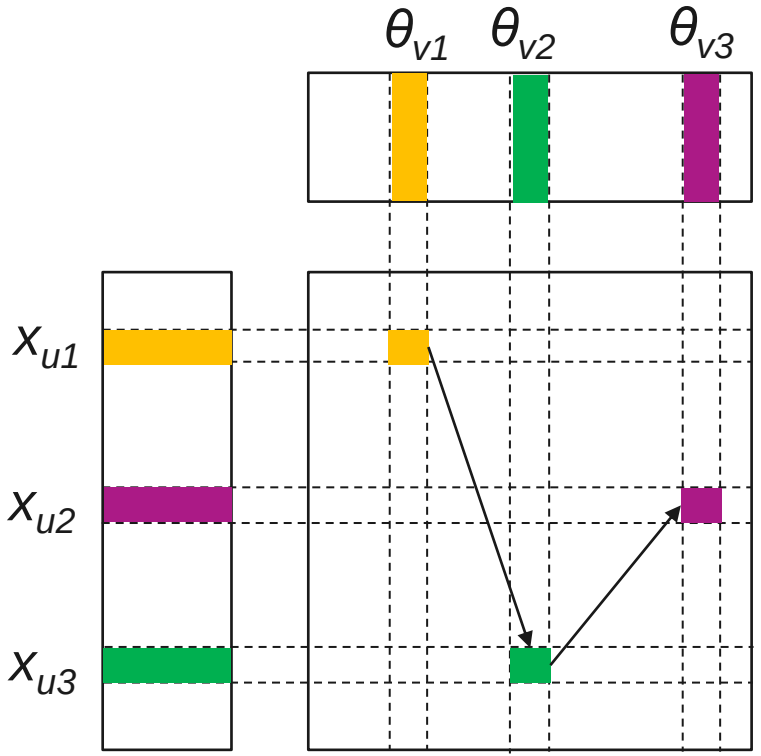
$$R \approx X \cdot \Theta^T$$

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

Stochastic gradient descent (**SGD**)

$$\boldsymbol{x}_u = \boldsymbol{x}_u - \alpha[(\boldsymbol{x}_u^T \boldsymbol{\theta}_v - r_{uv})\boldsymbol{\theta}_v + \lambda \boldsymbol{x}_u]$$
$$\boldsymbol{\theta}_v = \boldsymbol{\theta}_v - \alpha[(\boldsymbol{x}_u^T \boldsymbol{\theta}_v - r_{uv})\boldsymbol{x}_u + \lambda \boldsymbol{\theta}_v]$$

- 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

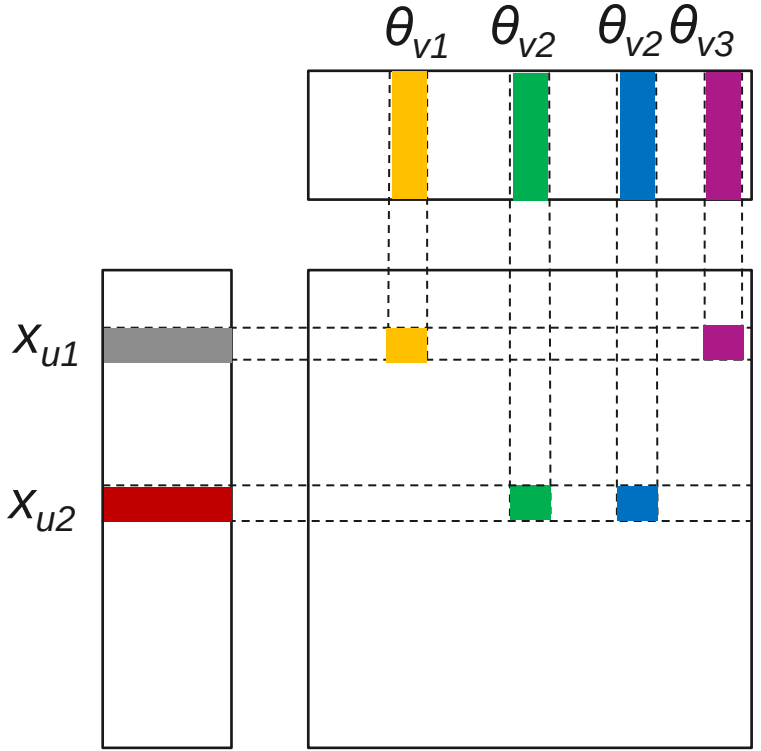
$$R \approx X \cdot \Theta^T \qquad J = \sum_{u,v}(r_{uv} - \boldsymbol{x}_u^T \boldsymbol{\theta}_v)^2 + \lambda(\sum_u n_{x_u}||\boldsymbol{x}_u||^2 + \sum_v n_{\theta_v}||\boldsymbol{\theta}_v||^2)$$

Alternating Least Square (**ALS**)

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

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

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



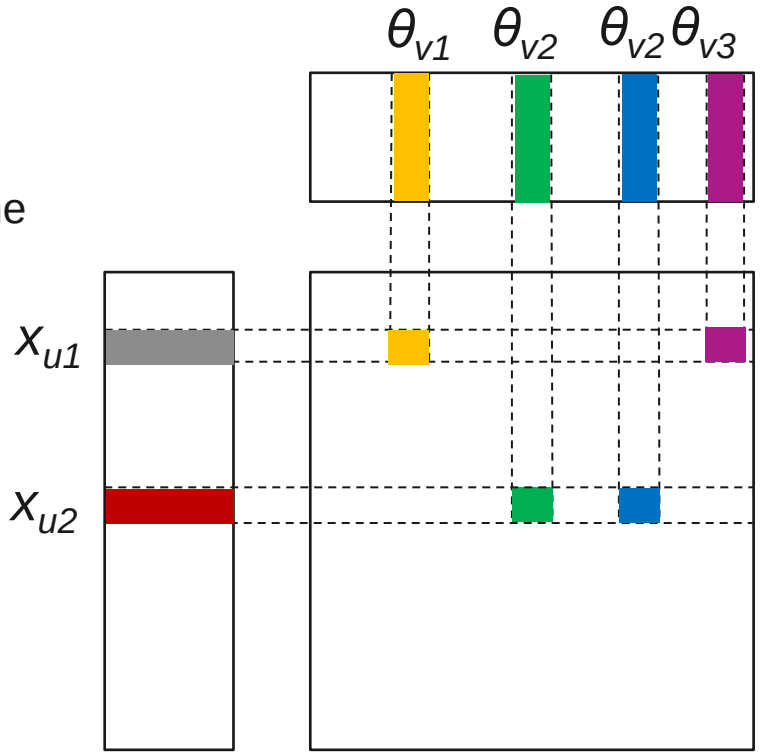
To Solve MF: CD

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

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

Coordinate descent (CD)

- Similar to ALS
- But update one coordinate of x_u and θ_u at a time



To Parallelize ALS

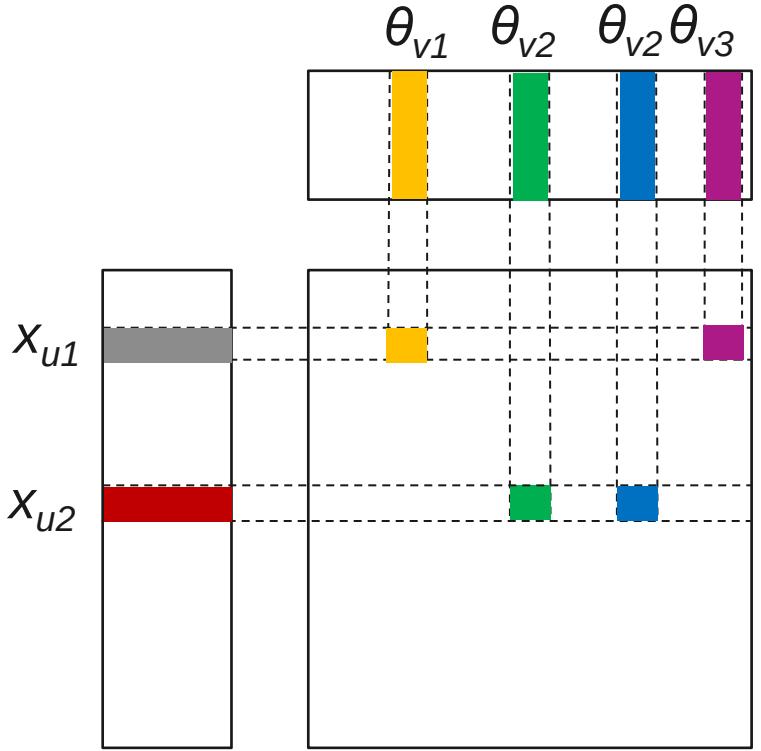
$$R \approx X \cdot \Theta^T \qquad J = \sum_{u,v}(r_{uv} - \boldsymbol{x}_u^T\boldsymbol{\theta}_v)^2 + \lambda(\sum_u n_{x_u}||\boldsymbol{x}_u||^2 + \sum_v n_{\theta_v}||\boldsymbol{\theta}_v||^2)$$

Alternating Least Square (**ALS**)

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

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

- Solve x_us independently (θ_vs thereafter)
- Parallelize the solve of x_us on multiple nodes
  - Replicate Θ
  - Partially replicate Θ
  - Split Θ on multiple nodes

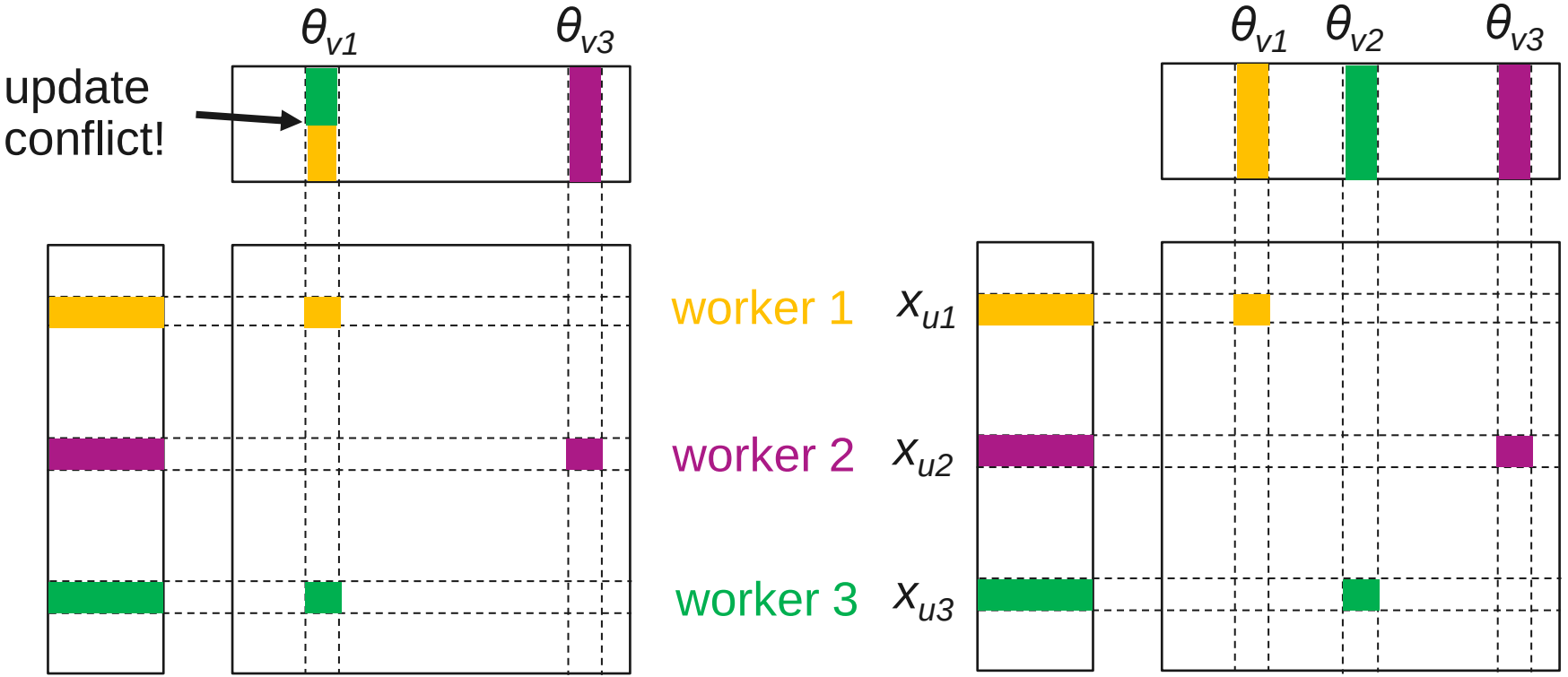


Parallelize SGD: Hogwild!

$$\boldsymbol{x}_u = \boldsymbol{x}_u - \alpha[(\boldsymbol{x}_u^T\boldsymbol{\theta}_v - r_{uv})\boldsymbol{\theta}_v + \lambda\boldsymbol{x}_u]$$

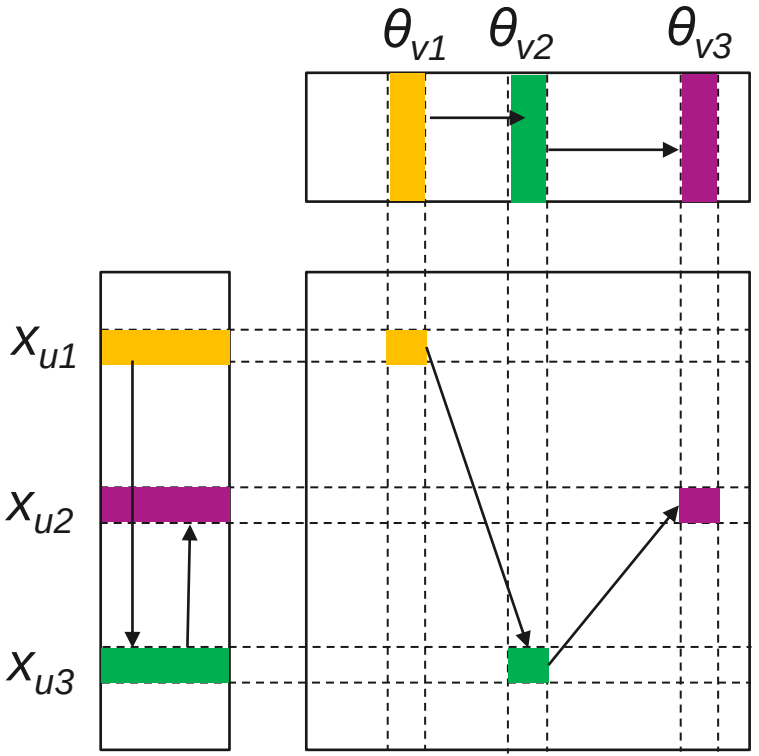
$$\boldsymbol{\theta}_v = \boldsymbol{\theta}_v - \alpha[(\boldsymbol{x}_u^T\boldsymbol{\theta}_v - r_{uv})\boldsymbol{x}_u + \lambda\boldsymbol{\theta}_v]$$

Hogwild! [Niu et al. 2011]
-- parallel SGD converges despite of (occasional) update conflict

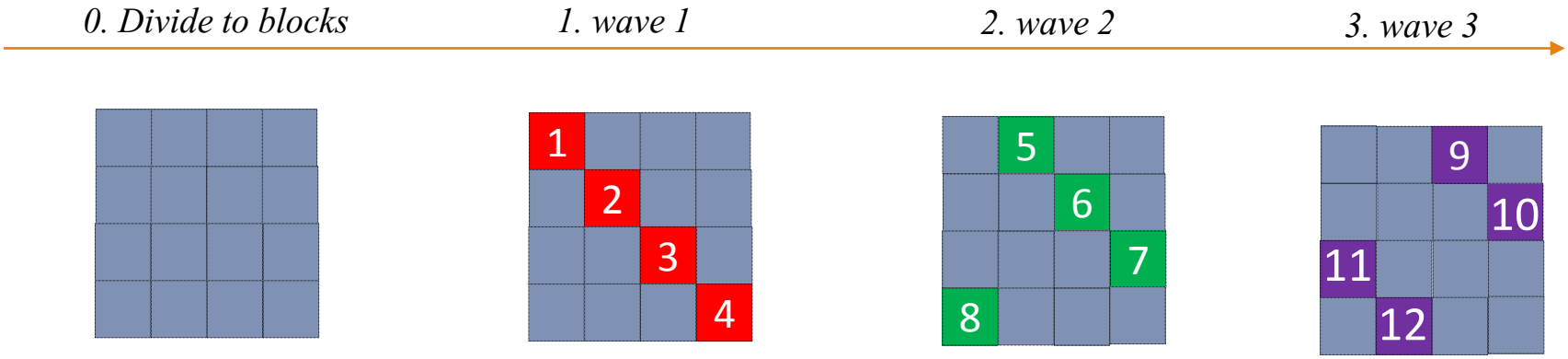


Hogwild! Not Good Enough?

Random sampling hurts cache performance (on GPUs)
-- hardware cannot prefetch

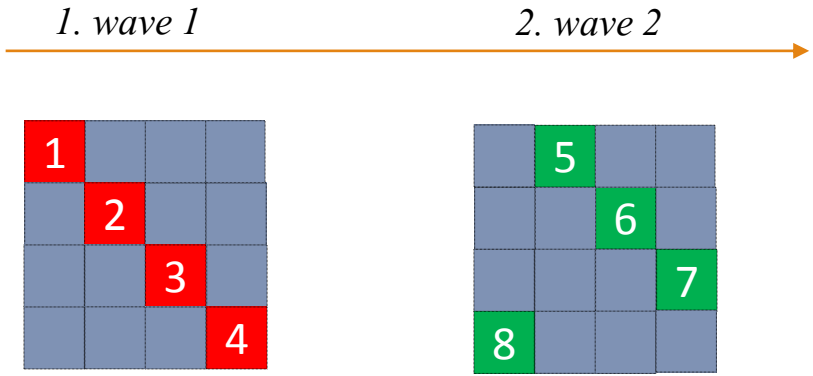


Parallelize SGD: Matrix blocking

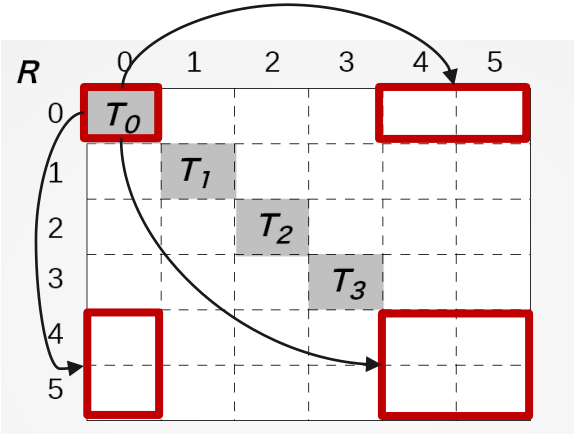


- Divide *R* into blocks, say 4*4
- 4 workers update 4 “non-overlapping” blocks concurrently
  - Workers do not need to communicate

Parallelize SGD: Matrix blocking



- Cons: all 4 workers need to complete before the next wave



- 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

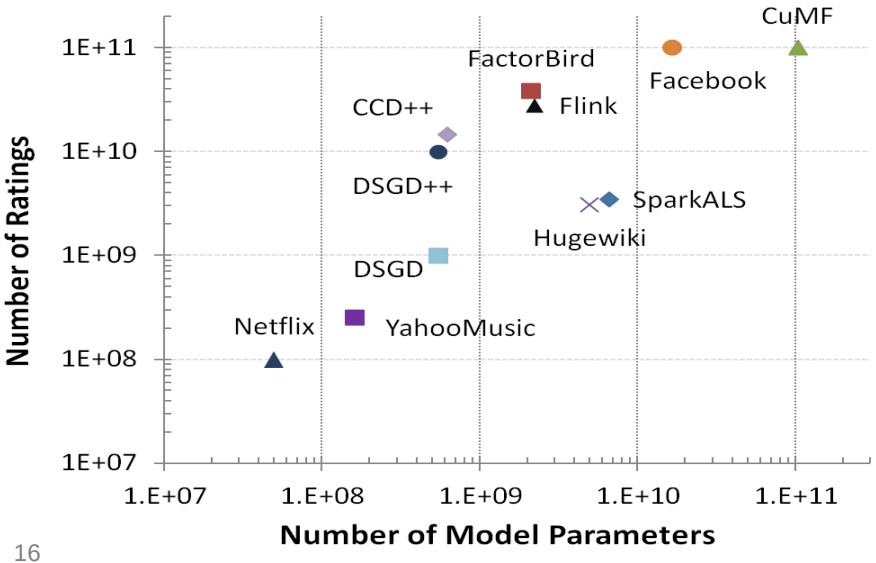
MF methods with SGD, ALS and CCD

	CPU	GPU
SGD	**lock-free**: workers independently sample & update single-node: HogWild! [15] multi-nodes: FactorBird [8], Petuum [28] **blocking**: workers on non-overlapping blocks blockDim=#workers: DSGD [13] blockDim>#workers: LIBMF [9], NOMAD [12], DSGD++ [18] nested blocking: dcMF [29], MLGF-MF [16]	**single and multiple GPUs**: cuMF_SGD [27] implements lock-free and blocking, and is memory-optimized with reduced precision.
ALS	**replicate features**: PALS [30], DALS [18] **partial replicate**: SparkALS [31], GraphLab [32], Sparkler [11] **rotate**: Facebook [5] **approximate ALS**: [33]	**single GPU**: BIDMach [24], ALS-HPC [25] **single and multiple GPUs**: cuMF [23]
CCD	**multi-core and multi node**: CCD++ [14]	

Challenge: compute and memory capacity of CPU

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

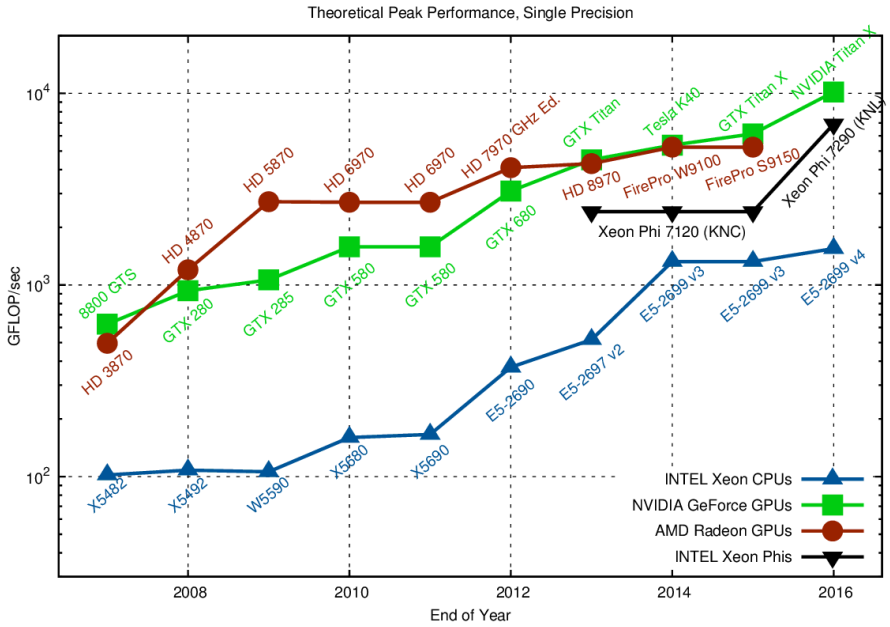
		Compute (C)	Memory (M)	C/M
ALS	get_hermitian	$\mathcal{O}(N_z f^2)$	$\mathcal{O}(N_z f + (m+n)f^2)$	f
	direct solver	$\mathcal{O}((m+n)f^3)$	$\mathcal{O}((m+n)f^2)$	f
SGD		$\mathcal{O}(N_z f)$	$\mathcal{O}(N_z f)$	1

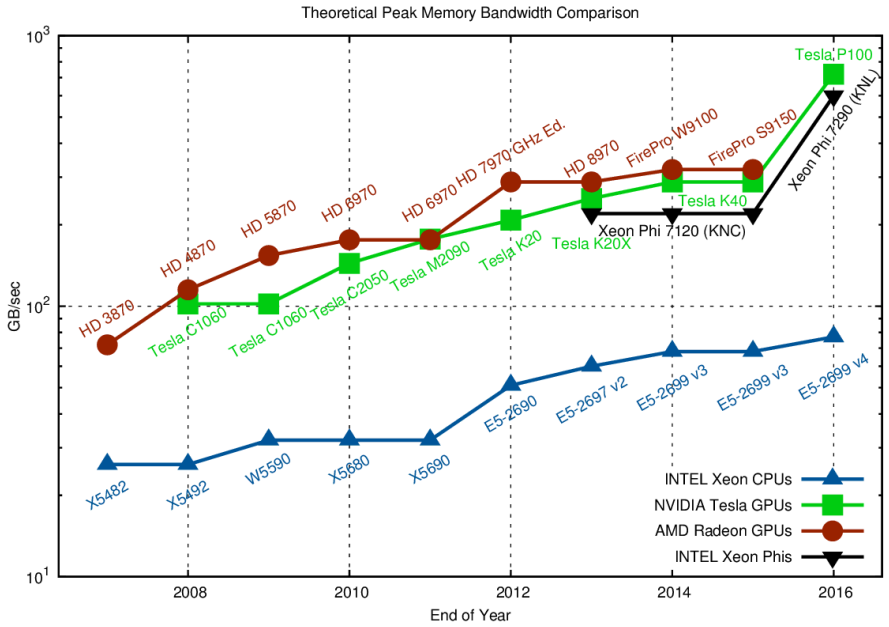


CPU offers: 1 T flops, 80 GB/s
f=100, per epoch

- 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





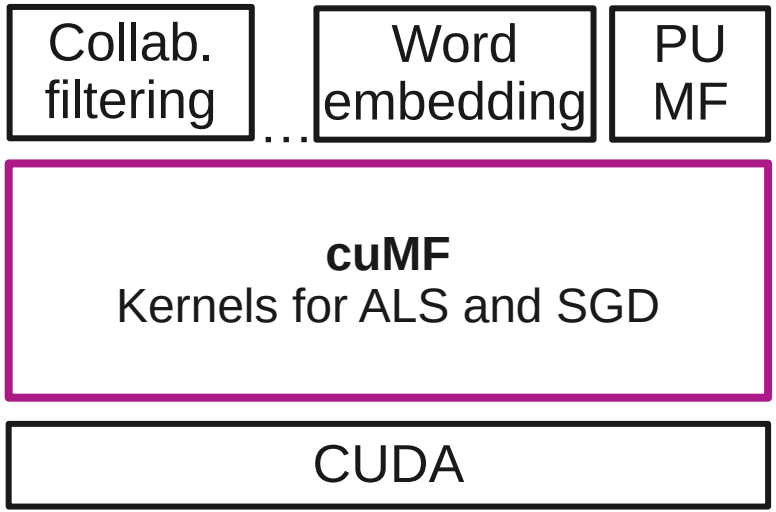
- Raw performance: **1 GPU ≈ 10 x CPU**
- Practical performance due to slow interconnection

1 GPU > 10 x CPU

4 GPU >> 40 x CPU

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Goal: a CUDA library for MF



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

cuMF
cuda Matrix Factorization

CUDA-based matrix factorization libraries.

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

http://researcher.ibm.com/person/us-wtan weitan@ieee.org

Challenges of ALS

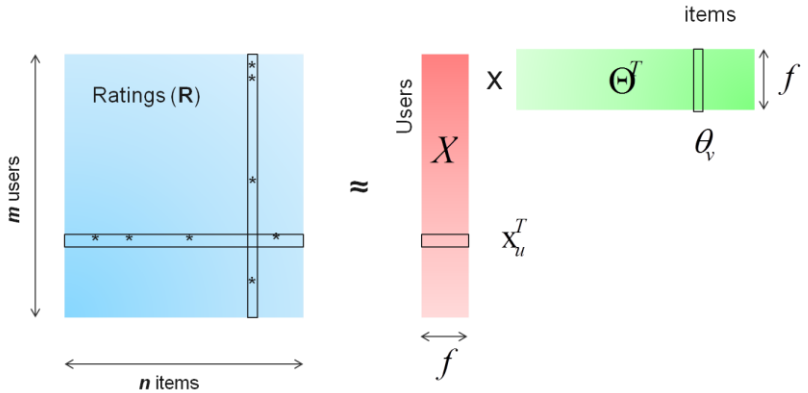
- ALS needs to solve many:

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

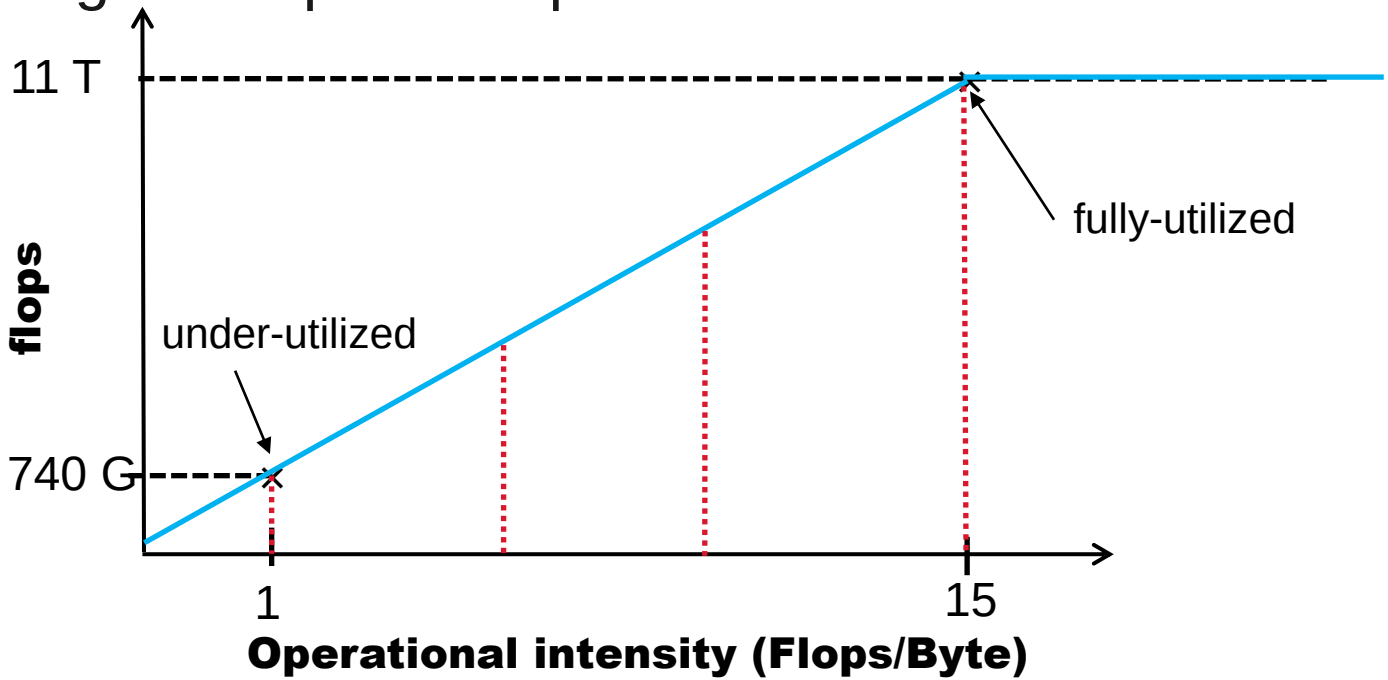
Challenge 2: LU or Cholesky solver: compute intensive

- Challenge 1: access and aggregate many θ_vs: memory **irregular** and compute **intensive**

- Challenge 3:
  Single GPU can NOT handle **big** *m*, *n* and *Nz*



Challenge 1: improve flops



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

S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (2009)

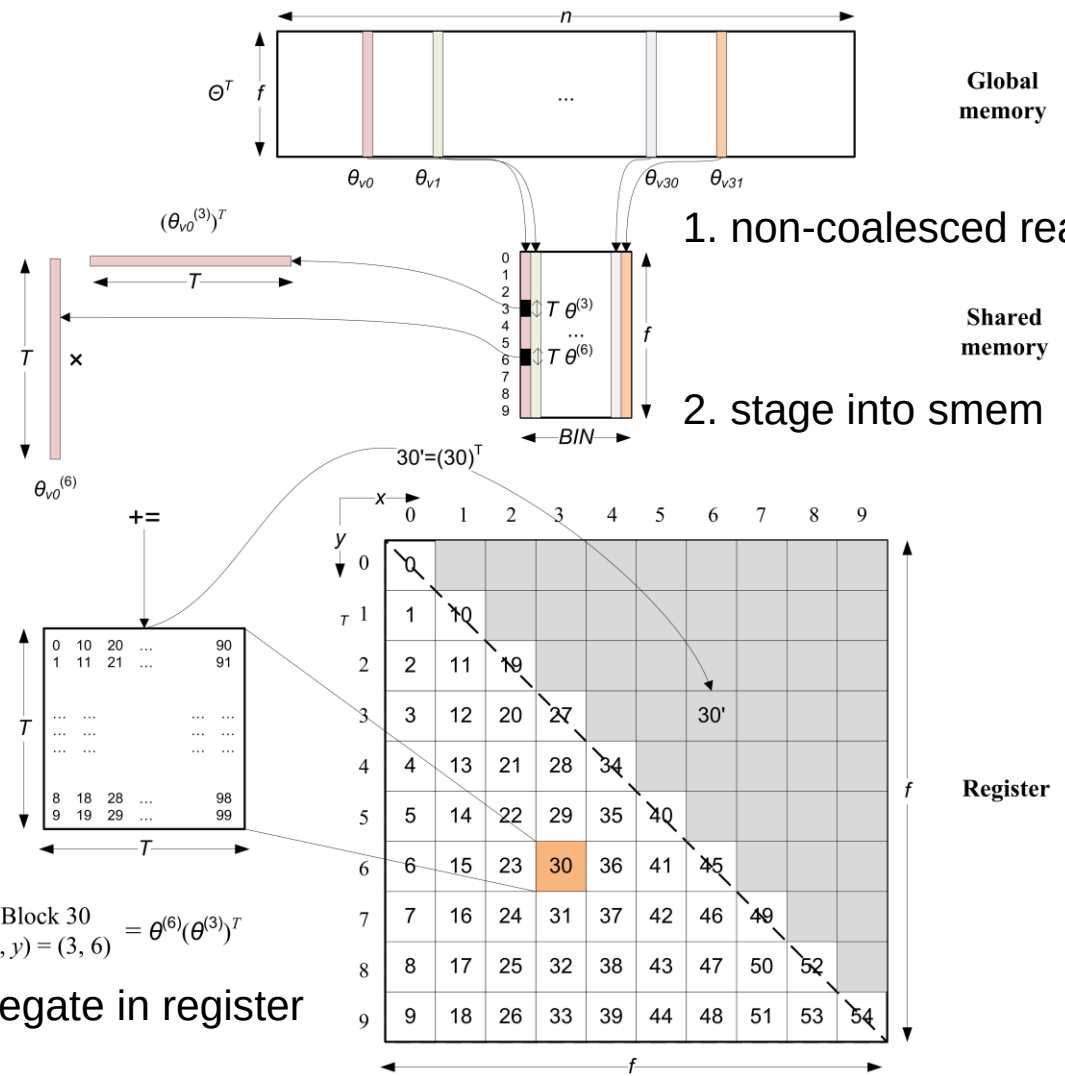
Address challenge 1: memory-optimized ALS

- To obtain

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

Register

Shared memory



Address Challenge 2: exact solver is compute intensive

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

ALS iter 1
- Exactly solve X
- Exactly solve Θ

ALS iter 2
- Exactly solve X
- Exactly solve Θ

…

ALS iter n
- Exactly solve X
- Exactly solve Θ

O(f^3)

ALS iter 1
- Approx. solve X
- Approx. solve Θ

ALS iter 2
- Approx. solve X
- Approx. solve Θ

…

ALS iter n
- Approx. solve X
- Approx. solve Θ

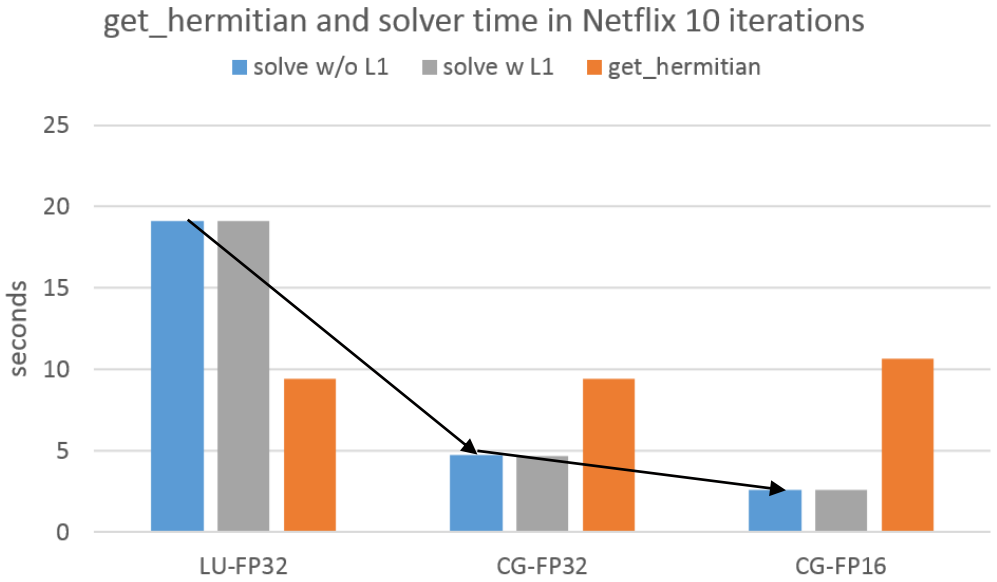
O(f^2)

Algorithm 1 The CG Solver.

```
1:  procedure CGSolve(A, b, x, f_s, ε)
2:      r = b − A · x
3:      p = r
4:      rs_old = r^T · r
5:      for j = 1 : f_s do
6:          a_p = A · p
7:          α = rs_old/(p^T · a_p)
8:          x = x + αp
9:          r = r − αp
10:         rs_new = r^T · r
11:         if √rs_new < ε 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
```

use *fs* << *f*

Address Challenge 2: use CG solver



Algorithm 1 The CG Solver.

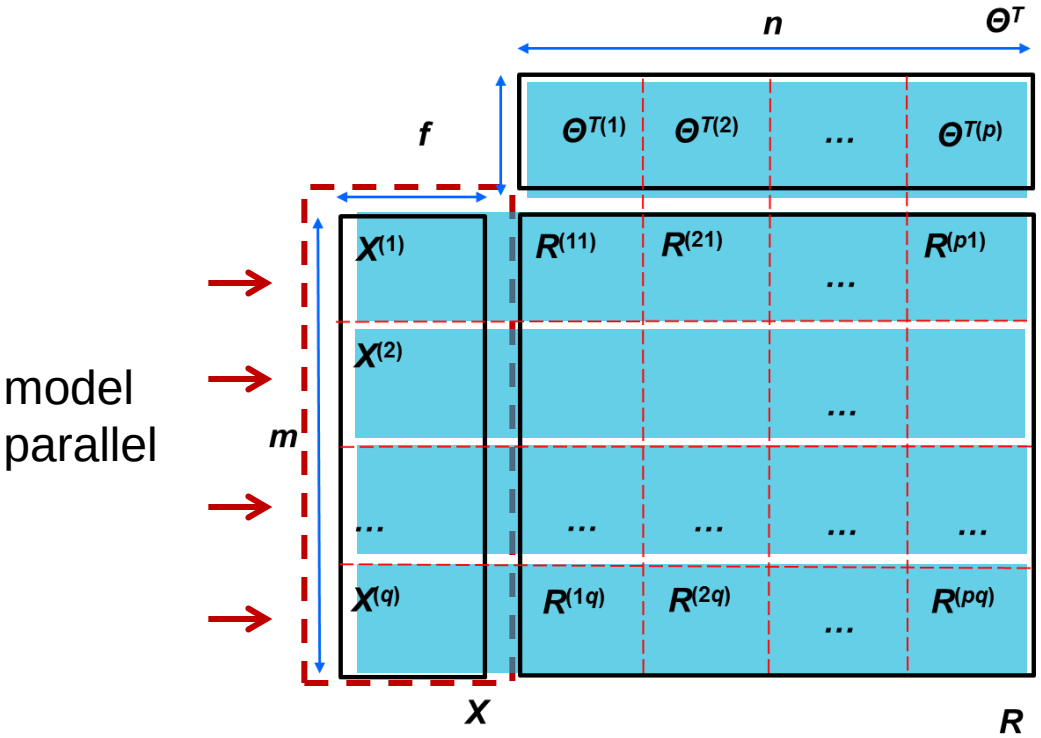
1: **procedure** CGSOLVE($A, \boldsymbol{b}, \boldsymbol{x}, f_s, \epsilon$)
2: $\boldsymbol{r} = \boldsymbol{b} - A \cdot \boldsymbol{x}$
3: $\boldsymbol{p} = \boldsymbol{r}$
4: $rs_{old} = \boldsymbol{r}^T \cdot \boldsymbol{r}$
5: **for** $j = 1 : f_s$ **do**
6: $\boldsymbol{a}_p = A \cdot \boldsymbol{p}$
7: $\alpha = rs_{old}/(\boldsymbol{p}^T \cdot \boldsymbol{a}_p)$
8: $\boldsymbol{x} = \boldsymbol{x} + \alpha \boldsymbol{p}$
9: $\boldsymbol{r} = \boldsymbol{r} - \alpha \boldsymbol{p}$
10: $rs_{new} = \boldsymbol{r}^T \cdot \boldsymbol{r}$
11: **if** $\sqrt{rs_{new}} < \epsilon$ **then**
12: **break**
13: **end if**
14: $\boldsymbol{p} = \boldsymbol{r} + (rs_{new}/rs_{old})\boldsymbol{p}$
15: $rs_{old} = rs_{new}$
16: **end for**
17: **return** $\boldsymbol{x}$
18: **end procedure**

- Solver time: CG = ¼ LU
- CG solver is memory- (instead of compute-) bound
  - CG w/ FP16 = ½ CG w/ FP32

Address Challenge 3: scale-up ALS on multiple GPUs

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

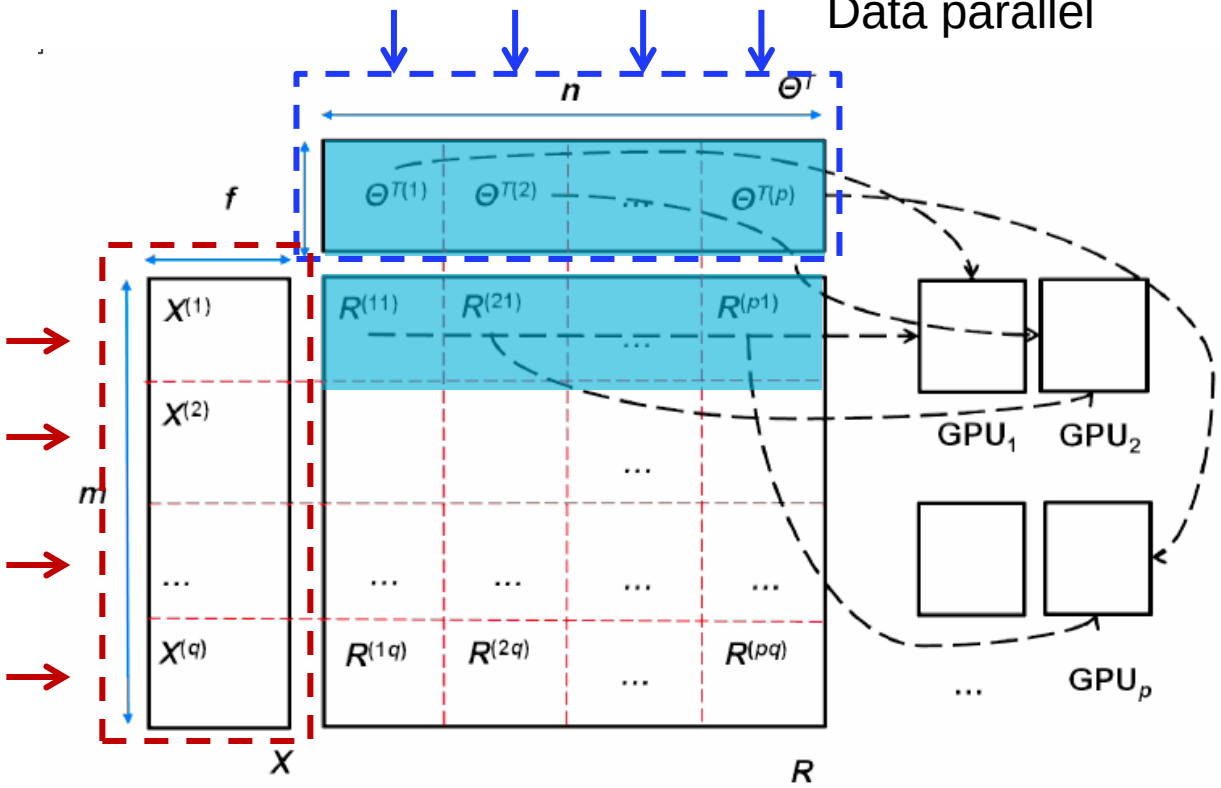
$$\sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) \cdot x_u = \Theta^T \cdot R_{u*}^T$$



Address Challenge 3: scale-up ALS on multiple GPUs

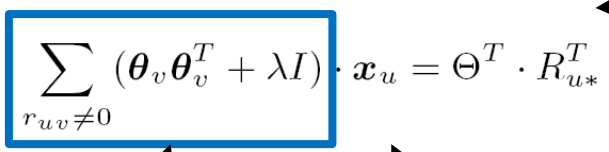
- **Data** parallel: solve with a portion of the training data

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



Recap: challenges of ALS

- ALS needs to solve many:

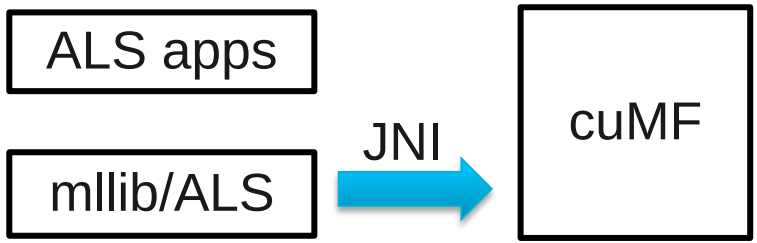


Challenge 2: LU or Cholesky solver: compute intensive
-- use approximate CG solver and FP16

- Challenge 1: access and aggregate many θ_vs: memory **irregular** and compute **intensive**
  -- use register, smem and non-coalesced read

- 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



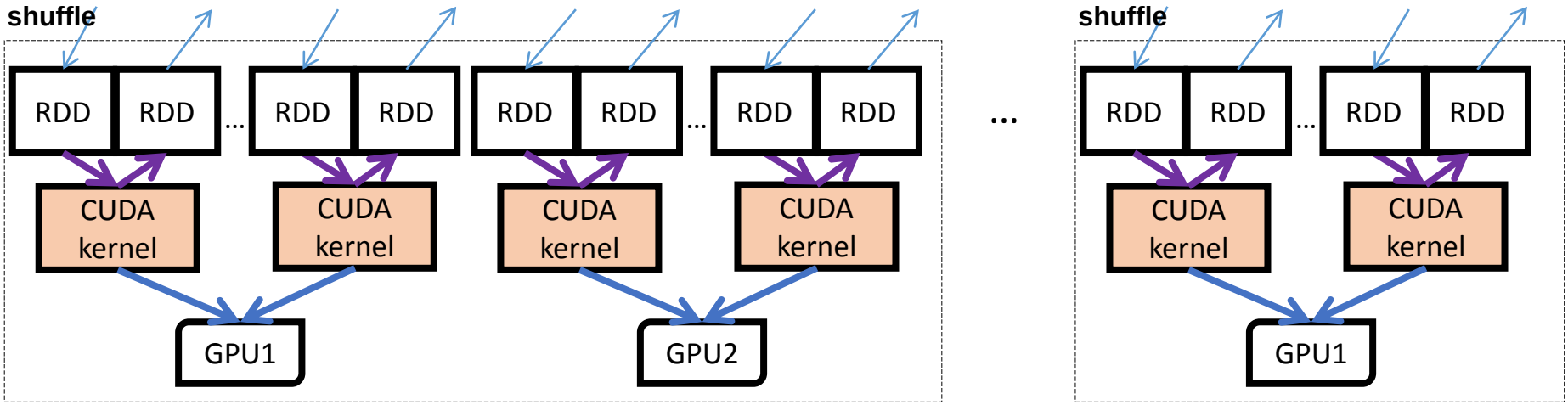
$$\sum_{r_{uv} \neq 0} (\boldsymbol{\theta}_v \boldsymbol{\theta}_v^T + \lambda I) \cdot \boldsymbol{x}_u = \Theta^T \cdot R_{u*}^T$$

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

https://github.com/IBMSparkGPU/CUDA-MLlib
http://www-01.ibm.com/support/docview.wss?uid=swg21983421

Connect cuMF to Spark MLlib

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



1 Power 8 node + 2 K40

1 Power 8 node + 2 K40

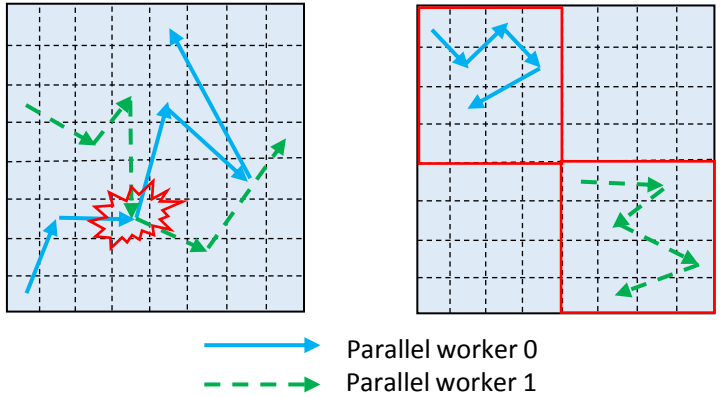
Challenges of SGD

- Iterate over all ratings and do this in sequence:

$$\boldsymbol{x}_u = \boldsymbol{x}_u - \alpha[(\boldsymbol{x}_u^T\boldsymbol{\theta}_v - r_{uv})\boldsymbol{\theta}_v + \lambda\boldsymbol{x}_u]$$
$$\boldsymbol{\theta}_v = \boldsymbol{\theta}_v - \alpha[(\boldsymbol{x}_u^T\boldsymbol{\theta}_v - r_{uv})\boldsymbol{x}_u + \lambda\boldsymbol{\theta}_v]$$

- Memory bound

2. how to parallelize



(a) Hogwild *(b) Matrix Blocking*

1. update kernel

Cache

Half precision

Memory Coalescing

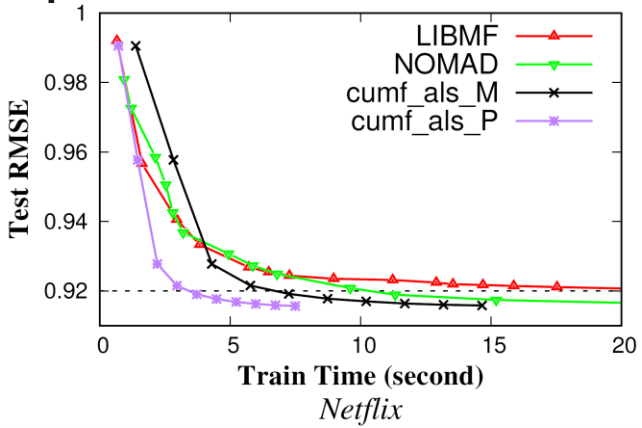
ILP

Warp shuffle

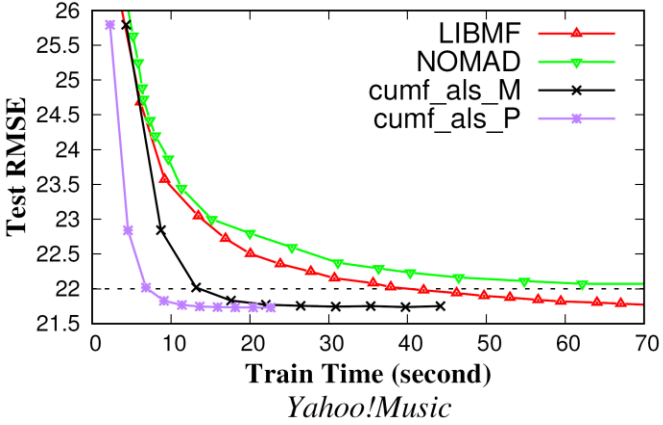
Register Reuse

```
1   //read the sample
2   int index = get_index_of_sample();
3   float r =  __ldg(&samples[index].r);
4   int   u = __ldg(&samples[index].u);
5   int   v = __ldg(&samples[index].v);
6
7   //read the p & q
8   int p_offset = u*k, q_offset = v*k;
9   float tmp_p1 = __half2float(p[p_offset + threadIdx.x]), tmp_p2 = __half2float(p[p_offset + threadIdx.x + 32]);
10  float tmp_q1 = __half2float(q[q_offset + threadIdx.x]), tmp_q2 = __half2float(q[q_offset + threadIdx.x + 32]);
11
12  //get dot product.
13  float tmp_product = tmp_p1*tmp_q1 + tmp_p2*tmp_q2;
14  tmp_product += __shfl_down(tmp_product, 16);
15  tmp_product += __shfl_down(tmp_product, 8);
16  tmp_product += __shfl_down(tmp_product, 4);
17  tmp_product += __shfl_down(tmp_product, 2);
18  tmp_product += __shfl_down(tmp_product, 1);
19  tmp_product = __shfl(tmp_product,0);
20  float der = e - tmp_product;
21
22  //compute the derivative and update
23  p[p_offset + threadIdx.x +  0] = __float2half(tmp_p1 + tmp_lrate*(der*tmp_q1 - lambda_p*tmp_p1));
24  q[q_offset + threadIdx.x +  0] = __float2half(tmp_q1 + tmp_lrate*(der*tmp_p1 - lambda_q*tmp_q1));
25  p[p_offset + threadIdx.x + 32] = __float2half(tmp_p2 + tmp_lrate*(der*tmp_q2 - lambda_p*tmp_p2));
26  q[q_offset + threadIdx.x + 32] = __float2half(tmp_q2 + tmp_lrate*(der*tmp_p2 - lambda_q*tmp_q2));
```

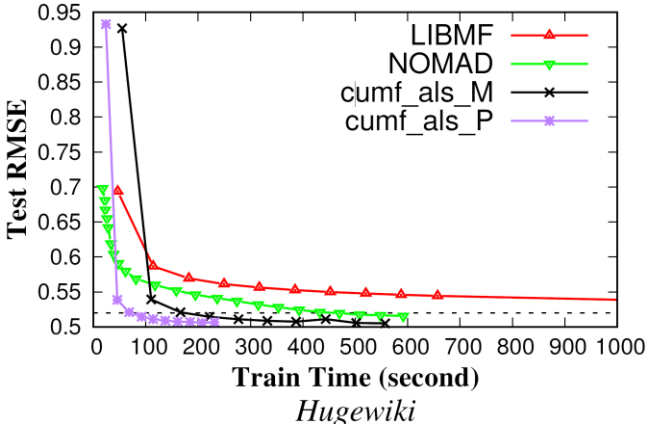
Experiment 1: is cuMF fast and scalable?



Netflix



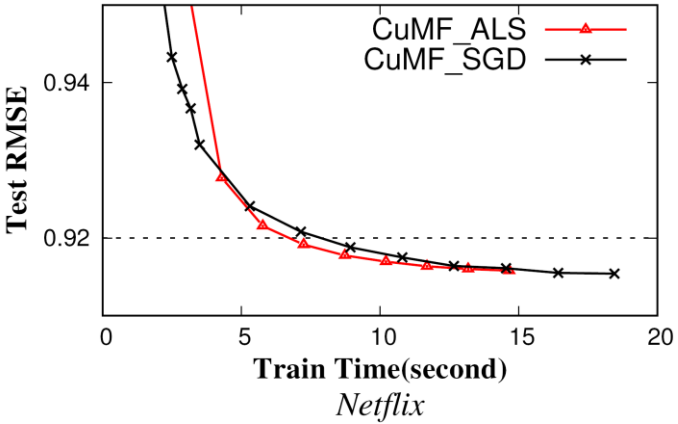
Yahoo!Music

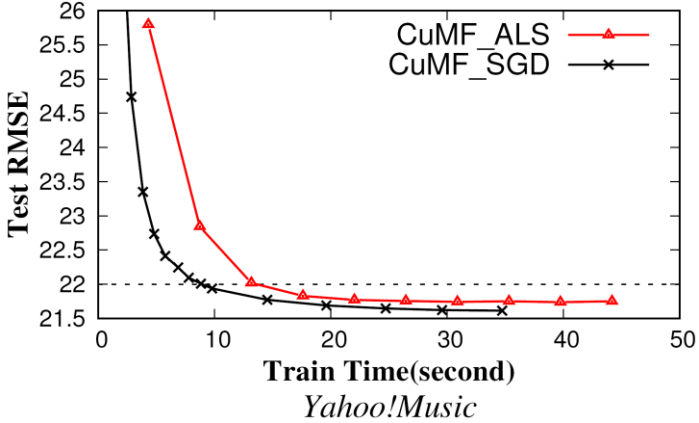


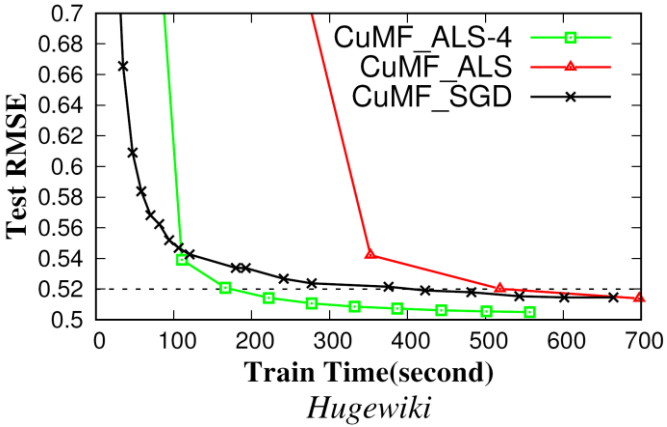
Hugewiki

- cuMF_ALS w/ FP16 on **M**axwell and **P**ascal
  - 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: cuMF_ALS and cuMF_SGD (on Maxwell)

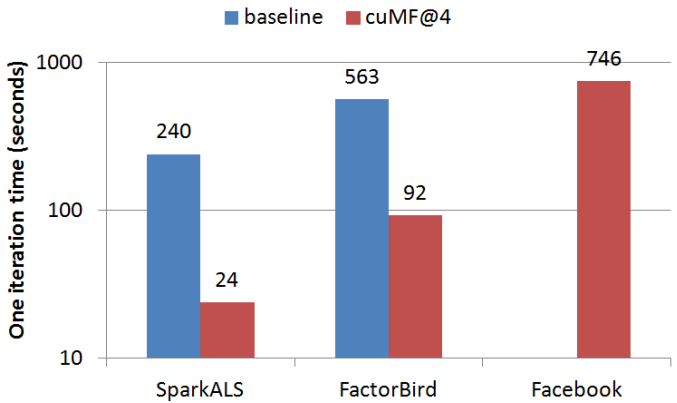






- ALS slightly slower than SGD on single GPU
- On big data set Hugewiki, ALS@4 GPU performs best -- SGD harder to parallel to multiple GPUs!

Experiment 3: is cuMF cost efficient?



- cuMF_ALS @4 Maxwell
  - ≈ **1/10** SparkALS @50 nodes
  - ≈ $2.5/hr → **1/10** of 50 nodes
  - ≈ **1%** of SparkALS's 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%**

Conclusion

- **Why** accelerate matrix factorization using GPU?
  - MF need to be fast, scalable, and economic
  - GPU offers ~10x flops, memory BW, and fast interconnect
- **How** cuMF tackles the challenges?
  - Optimize memory access, parallelism and communication
  - Approximate computing
  - Reduced precision
- **What** is the result?
  - Implemented ALS and SGD
  - Up to 10x as fast, 100x as cost-efficient
  - Use cuMF standalone, with Spark or Tensorflow

Thank you, questions?

- Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs. HPDC 2016
- CuMF_SGD: Fast and Scalable Matrix Factorization. CoRR abs/1610.05838, 2016
- Code: http://github.com/cuMF/
- Blog: http://ibm.biz/cumf-blog
- Contact: Wei Tan, wtan@us.ibm.com