PARALLEL COMPUTING HEURISTICS FOR LOW-RANK MATRIX COMPLETION

Charlie Hubbard and Chinmay Hegde

Electrical and Computer Engineering Department
Iowa State University, Ames, IA, USA 50010

ABSTRACT

Current algorithms for low-rank matrix completion often suffer from scalability issues — both in terms of memory as well as running time — when presented with very large datasets. In this paper, we introduce new parallel computing heuristics that can greatly accelerate matrix completion algorithms when used in GPU-based computing environments. Our heuristics enable speeding up popular algorithms for nonlinear matrix completion on standard real-world test datasets by orders of magnitude, while being highly memory-efficient.

1. INTRODUCTION

Motivation. The problem of recovering a data matrix from a small sample of its entries, also called the *matrix completion* problem, arises in several real-world applications including content recommender systems, sensor localization, and system identification.

For real-world applications, this problem poses several challenges. First, the application of standard approaches for matrix completion to the problem of content recommendation is not seamless. The matrix completion literature assumes that the entries of the matrix are *real-valued*; however, the ratings provided by users of content providers are almost always "quantized" to some finite set of integers. To remedy this, the use of *1-Bit matrix completion* [1] has been shown to outperform contemporary matrix completion methods by intrinsically modeling the ratings as non-numeric entities.

Second, standard matrix completion algorithms do not leverage any "side-information" known about the rows (users) and items (columns) of a given matrix. To remedy this, a new framework known as *inductive matrix completion* (IMC) [2] has been proposed. However, incorporating this extra information expands the memory footprint of the associated algorithms.

In both of the above situations, current state-of-the-art methods suffer from *scalability* issues. For example, in collaborative filtering applications used by content providers such as Netflix or Amazon, the number of users and/or items can both be in the order of hundreds of millions. In order to perform efficient matrix completion for very large datasets, parallelization is paramount. The question to be asked is: how best to leverage modern parallel computing environments for matrix completion methods?

Our contributions. In this paper, we introduce two new parallel computing heuristics for solving matrix completion problems.

Our heuristics, that we call GPUFISH and IMCFISH, are modular, tunable, and leverage the massive number of multiple concurrent kernel executions possible on a modern GPU. As stylized applications, we demonstrate how to adapt GPUFISH to solve the 1-bit matrix completion problem where the matrix observations are binary [1]. Our results demonstrate that we achieve a 150x speedup over existing serial algorithms, while maintaining comparable prediction accuracy. Our work demonstrates that a standard workstation equipped with a single GPU can be effectively deployed to solve very large scale matrix completion problems. We also demonstrate the use of IMCFISH to solve inductive matrix completion problems. For very large IMC problems IMCFISH is able to obtain competitive solutions while only having access to a fraction of the dataset.

Our work is open-source and a CUDA implementation of our proposed heuristics is available at https://github.com/cghubbard/gpu-fish.

Our techniques. The algorithmic core of GPUFISH is an optimized implementation of the JELLYFISH framework [3]. Similar to the approach proposed in [3], our approach also employs a randomized, incremental stochastic gradient descent approach. However, GPUFISH generalizes the previous approach in two distinct ways: (i) GPUFISH enables the user to transparently adapt to domain-dependent problems, thus extending the matrix completion framework to numeric as well as non-numeric observations. (ii) GPUFISH enables the user to leverage the full parallel processing power of a GPU, and can concurrently process hundreds of available samples in training.

For inductive matrix completion (IMC) problems, a straightforward application of the above parallelization scheme does not work. Instead, our proposed heuristic IMCFISH uses a *partial* gradient descent scheme that enables parallel updates in a manner similar to JELLYFISH. This gradient descent scheme relies on a novel data partitioning approach that we call *striping*. Together with the parallelization heuristics described above, this leads to improved performance.

Relation to prior work. The recent, large body of work in matrix completion has shown that as long as the matrix $\mathbf{M}$ possesses *sufficiently low rank*, we can recover the missing entries of $\mathbf{M}$ via a convex optimization procedure [4–7]. However, convex optimization approaches are not particularly suitable for matrices larger than a few hundred rows/columns. To resolve this, a non-convex, incremental heuristic for matrix completion was introduced in [3].

The problem of completing a matrix whilst taking into account the available rows/column features is known as *inductive matrix completion* (IMC). Theory, applications, and expansions of IMC are explored in [2, 8–10]. However, to our knowledge there have been no works that explore the effect of parallelization for IMC algorithms.

Finally, a very recent work [11] advocates a new algorithm for GPU-based matrix completion based on *cyclic coordinate descent* (CCD). Our method differs in two respects: we consider the more complicated problems of 1-bit matrix completion as well as inductive matrix completion, and our update rules involve large portions of the matrix variables (as opposed to individual coordinates). Due to space (and time) constraints, we defer a thorough comparison to future work.

2. PARALLEL 1-BIT MATRIX COMPLETION

As our first application, we describe an instantiation of GPUFISH for solving large scale instances of the matrix completion problem where the user ratings are available in the form of binary (like/dislike) observations. We adopt the 1-bit matrix completion model of [1]. The goal is to fill in any missing entries of a rank-r matrix $\mathbf{M}$ with n_r rows and n_c columns. However, in a departure from classical matrix completion, we do not get to directly observe the entries of $\mathbf{M}$. Instead, consider any twice-differentiable function $p : \mathbb{R} \to [0, 1]$. We record observations $\mathbf{Y}$ such that:

$$Y_{i,j} = \begin{cases} +1 & \text{with probability } p(M_{i,j}), \\ -1 & \text{with probability } 1 - p(M_{i,j}), \end{cases} \quad \text{for} \quad (i,j) \in \Omega. \tag{2.1}$$

As with previous work in matrix completion, it is important that Ω is chosen *uniformly at random*. In [1], the *Probit* and *Logit* functions are explored as natural functions to model the underlying distribution $p(\cdot)$ of the entries of $\mathbf{Y}$. In this work, we focus on the Logit function $p(x) = \frac{e^x}{1+e^x}$. To recover an estimate of $\mathbf{M}$ we can maximize the log-likelihood function of the optimization variable $\mathbf{X}$ over the set of observations Ω. Denote $\mathbb{1}_A$ as the indicator function over a Boolean condition A. Then the log-likelihood function corresponding to the Logit model is given by:

$$\mathcal{L}(\mathbf{X}) := \sum_{(i,j)\in\Omega} \big(\mathbb{1}_{Y_{i,j}=1} \log(p(X_{i,j})) + \mathbb{1}_{Y_{i,j}=-1} \log(1 - p(X_{i,j}))\big) \tag{2.2}$$

The estimate of $\mathbf{M}$, therefore, is given by the solution to the constrained optimization problem[1]:

$$\widehat{\mathbf{M}} = \underset{\mathbf{X}}{\text{argmax}}\ \mathcal{L}(\mathbf{X}), \ \ \text{rank}(\mathbf{X}) \le r\,. \tag{2.3}$$

The optimization problem (2.3) is non-convex, due to the presence of the rank constraint on $\mathbf{X}$. The standard method adopted in matrix completion approaches is to perform a *nuclear norm relaxation* of the rank constraint. However, nuclear norm-regularized matrix recovery formulations can incur a high running time [3,4,6]. In order to resolve this issue, we adopt the JELLYFISH approach of [3]. We factorize the variable $\mathbf{X}$ into two variables $\mathbf{L}$ and $\mathbf{R}$ and obtain the *bilinear* optimization problem:

$$\min\ \mathcal{L}(\mathbf{L}\mathbf{R}^*) \text{ s.t. } \|\mathbf{L}\|_{2,\infty}^2 \le \text{B},\ \|\mathbf{R}\|_{2,\infty}^2 \le \text{B}. \tag{2.4}$$

To solve (2.4), we adopt an *incremental projected gradient descent* approach. We alternately update $\mathbf{L}$ (resp., $\mathbf{R}$) while keeping $\mathbf{R}$ (resp., $\mathbf{L}$) fixed. In each iteration, the updates to the i^{th} row of $\mathbf{L}$ and the j^{th} row of $\mathbf{R}$ are given by:

$$\begin{aligned} \mathbf{L}_{i_k}^{(k+1)} &= \Pi_{\text{B}}\left(\mathbf{L}_{i_k} - \alpha_k \mathcal{L}'(\mathbf{L}_{i_k}^{(k)}\mathbf{R}_{j_k}^{(k)*})\mathbf{R}_{j_k}^{(k)}\right) \\ \mathbf{R}_{i_k}^{(k+1)} &= \Pi_{\text{B}}\left(\mathbf{R}_{i_k} - \alpha_k \mathcal{L}'(\mathbf{L}_{i_k}^{(k)}\mathbf{R}_{j_k}^{(k)*})\mathbf{L}_{j_k}^{(k)}\right) \end{aligned} \tag{2.5}$$

where the projection operator Π onto the constraint set in (2.4) admits the closed form expression $\Pi_B(v) = \frac{\sqrt{B}v}{\|v\|}$ if $\|v\|^2 \ge$ B and v otherwise. The step size parameter α_k decreases by a constant amount at every iteration.

The above gradient descent method has several computational advantages. We observe that the updates performed in (2.5) operate on highly *local* portions of the matrices $\mathbf{L}$ and $\mathbf{R}$. Therefore, if the observed data points in Ω were suitably partitioned into non-overlapping blocks, then each block can be *independently* processed. We will refer to these blocks as *chunks* and index any chunk C as $C_{a,b}$ where a and b are row and column indices of the chunk in the partitioned matrix. Moreover, if we have the means to process p chunks in parallel, then we can divide $\mathbf{M}$ into p^2 chunks, and further group the chunks into p rounds. Now, we can process each round sequentially, such that no two parallel processes will ever be manipulating the same rows of $\mathbf{R}$ or $\mathbf{L}$ at the same time, thus eschewing any *locking* delays[2]. This is the intuition exploited in [3]; however, they assume a standard multi-core CPU-based computing model, and also do not consider the problem of nonlinear (one-bit) matrix completion.

GPUFish: Parallel matrix completion on GPU. We provide a high level description of the organization of a GPU. Each process instantiated on the GPU is known as a *kernel*. A kernel can be executed in parallel across several threads of the GPU. A compiler hierarchically groups the parallel threads into *blocks*, and blocks into *a grid* of blocks. When launching a kernel on the GPU, the user controls the number of blocks to launch as well as the number of threads per block to launch. Each thread launched by the kernel executes an instance of that kernel. Threads in a block execute concurrently.

We now leverage this special architecture for our incremental gradient descent algorithm. Suppose that we have divided Ω into p^2 chunks. We launch a single kernel for each of the p rounds we have created. As noted above, the kernels must be launched sequentially to perform the parallel updates without fine-grained locking. Each round will contain p chunks so we will instantiate our kernel with p blocks. Each block is responsible for performing gradient updates (2.5) for all data points (samples) in the corresponding chunk. Each block of the kernel contains r worker threads; each thread, t_k, in a given block is responsible for updating $\mathbf{L}_{ik}$ and $\mathbf{R}_{jk}$, that is, the k^{th} entry in the rows of $\mathbf{L}$ and $\mathbf{R}$ being updated by (2.5) according to the data point $(i, j, Y_{i,j})$. In this way, we not only perform the gradient updates for a large number of data points, but also update *in parallel* the r entries of any row of $\mathbf{L}$ or $\mathbf{R}$. Therefore, using this procedure we get an r-fold speedup *per round* over the JELLYFISH algorithm.

We can further optimize running time as follows. While a given kernel (corresponding to one of the rounds) is being processed by the GPU, we simultaneous loading the data required to execute the next kernel onto the GPU. At the completion of the given we remove its data from the GPU and continue to the next round. The process of chunking our data and then performing parallel gradient updates over p kernels is known as an *epoch*. Because each epoch requires a new shuffle of our dataset, we begin each epoch by launching a separate CPU thread to compute the shuffle required for the next epoch; this extra CPU thread is executed in parallel with the GPU kernel launches being handled by the main CPU.

Each of the above optimization heuristics are specific to GPU-based computing and enable considerable improvements in running time over the standard JellyFish algorithm. Therefore, we call our modified approach GPUFISH, summarized as Algorithm 1.

We discuss some specific schemes for managing the various observations and variables in practical implementations of GPUFish. Before the first epoch, the matrices $\mathbf{L}$ and $\mathbf{R}$ are loaded into the *global memory* of the GPU where they can be accessed by all threads of the GPU. Thread access to global memory is generally slow, so rather

[1]To be precise, the problem formulation in [1] also included a boundedness constraint on $\|\mathbf{M}\|_\infty$, but we omit that constraint here.

[2]The JELLYFISH algorithm further generate random permutations of the row and column indices of our matrix $\mathbf{M}$, π_{row} and π_{col} to ensure that the data points in any chunk differ between subsequent passes over that data set.

Algorithm 1 GPUFish

```
1: Permute rows and columns of $\mathbf{M}$, shuffle $\Omega$
2: Separate $\Omega$ into $p^2$ chunks
3: Round$[i] = p$ chunks s.t. all chunks are non-overlapping
4: Transfer data for Round[1] to GPU
5: for i = 1 to $p$ in parallel do
6:     GPU_Gradient_Updates($p$ blocks, $r$ threads per block)
7:     Transfer data for Round[i+1] to GPU overwriting Round[i-1]
8: end for
```

Algorithm 2 GPU_Gradient_Updates

```
1: for each of $p$ chunks in parallel do
2:     for each data point $(i, j, rating)$ in the chunk do
3:         apply (2.5) to $\mathbf{L}$ and $\mathbf{R}$
4:     end for
5: end for
```

than make repeated calls to global memory we begin by loading the relevant rows of $\mathbf{L}$ and $\mathbf{R}$ into *shared memory* on the GPU. This memory is shared only between the threads of each block and access to it is significantly faster than global memory. After completing our computation of (2.4) from our copies of $\mathbf{L}$ and $\mathbf{R}$ in shared memory we write the our updates to $\mathbf{L}$ and $\mathbf{R}$ back to global memory.

In addition to making use of the GPU's faster shared memory, we also make use of the ability of the GPU's ability to transfer data while processing a kernel(s). At the beginning of each epoch we transfer the data needed for the first round of gradient updates and launch the kernel responsible for performing updates on the data. As this kernel processes the first round we gather the data required to process round two and load it onto the GPU. This procedure is repeated until the epoch has finished. While providing an obvious speedup over performing all of the data transfers at once, this data management scheme also enables us to only have two rounds worth of data (approximately $\frac{2\times|\Omega|}{p}$ data points) on the GPU at any given time. This enables GPUFish to process *ultra-large data sets* even on memory-limited GPUs.

3. PARALLEL INDUCTIVE MATRIX COMPLETION

We now extend the above ideas for a different matrix recovery problem known as *inductive matrix completion* (IMC). In IMC, we define two new *feature matrices*: $\mathbf{A} \in \mathbb{R}^{n_r \times n_{d_1}}$ that contains side information about the rows of $\mathbf{M}$, and $\mathbf{B} \in \mathbb{R}^{n_c \times n_{d_2}}$ that contains side informations about the columns of $\mathbf{M}$. Under this model, any observation can be written as

$$M_{i,j} = A_i \mathbf{Z} B_j{}^*$$

where $\mathbf{Z} \in \mathbb{R}^{d_1 \times d_2}$ describes a latent low-rank matrix variable[3]. In the same manner as our previous formulation we will factorize our decision variable $\mathbf{Z}$ as $\mathbf{X} = \mathbf{L}\mathbf{R}^*$ where $\mathbf{L} \in \mathbb{R}^{d_1 \times k}$ and $\mathbf{R} \in \mathbb{R}^{d_2 \times k}$.

As above, in order to estimate Z, we obtain a bilinear optimization problem:

$$\min\ \mathcal{L}(\mathbf{A}\mathbf{L}(\mathbf{B}\mathbf{R})^*) \text{ s.t. } \|\mathbf{L}\|_{2,\infty}^2 \le \mathrm{B},\ \|\mathbf{R}\|_{2,\infty}^2 \le \mathrm{B}. \qquad (3.1)$$

[3]The feature vectors for the rows and columns $\mathbf{M}$ are often constructed from meta-information: e.g., "likes" and "reblogs" information in [12] and phenotype relationships in [9]. From this meta-information, we can create adjacency matrices that describe user-user and item-item relationships, and take the leading eigenvectors of these adjacency matrices as feature vectors [9].

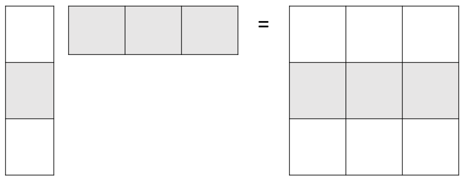

Fig. 1. *Multiplication by a striped vector gives a striped product.*

Proceeding in an identical fashion as (2.5), we can use incremental projected gradient descent to solve (3.1); our updates for each iteration are:

$$\begin{aligned} \mathbf{L}_k^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{L}_k - \alpha_k \mathcal{L}'(\mathbf{A}_i \mathbf{L}_k^{(k)} \mathbf{R}_k^{(k)*} \mathbf{B}_j^*) \mathbf{A}_i^* \mathbf{B}_j \mathbf{R}_k^{(k)}\right) \\ \mathbf{R}_k^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{R}_k - \alpha_k \mathcal{L}'(\mathbf{A}_i \mathbf{L}_k^{(k)} \mathbf{R}_k^{(k)*} \mathbf{B}_j^*) \mathbf{B}_j^* \mathbf{A}_i \mathbf{L}_k^{(k)}\right) \end{aligned} \qquad (3.2)$$

The updates in (3.2) differ from the standard one-bit matrix completion problem in a rather fundamental fashion. Crucially, the gradient updates are *no longer local*; every incremental update step affects all of $\mathbf{L}$ and all of $\mathbf{R}$, in sharp contrast with our earlier case. Therefore, existing "block"-based data partitioning techniques (such as that proposed in [3]) are no longer applicable. We resolve this issue using a novel algorithm described as follows.

IMCFish: Parallel inductive matrix completion. We first introduce the notion of *striping*. Suppose we choose an integer parameter s (for simplicity, a divisor of both m and n); then, the q^{th} stripe of a vector $v \in \mathbb{R}^{1\times n}$, denoted v^q, is given by: $v^q[k] = 0$ for $k < q * (m/s)$ and $k \ge (q+1) * (m/s)$, and $v[k]$ otherwise. If we multiply the transpose of our striped vector, $v^* \in \mathbb{R}^{n\times 1}$ with some other vector $t \in \mathbb{R}^{1\times m}$ we observe that the product of these two vectors, $\mathbf{Y}$, is a striped matrix (see Fig. 1): a row of $\mathbf{Y}$ is zero if the corresponding row of v^* is zero. Using this knowledge we can rewrite out updates (3.3) in terms of stripes of $\mathbf{A}$ and $\mathbf{B}$:

$$\begin{aligned} \mathbf{L}_k^{q,(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{L}_k - \alpha_k f'(\mathbf{A}_i \mathbf{L}_k^{(k)} \mathbf{R}_k^{(k)*} \mathbf{B}_j^*) \mathbf{A}_i^{*q} \mathbf{B}_j^{q'} \mathbf{R}_k^{(k)}\right) \\ \mathbf{R}_k^{q',(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{R}_k - \alpha_k f'(\mathbf{A}_i \mathbf{L}_k^{(k)} \mathbf{R}_k^{(k)*} \mathbf{B}_j^*) \mathbf{B}_j^{*q'} \mathbf{A}_i^q \mathbf{L}_k^{(k)}\right) \end{aligned} \qquad (3.3)$$

In this way, we are able to confine our gradient updates to *local* portions of $\mathbf{L}$ and $\mathbf{R}$ allowing us to do many updates in parallel. We call this algorithm IMCFish, and omit its pseudocode description due to space constraints.

We now describe an efficient GPU-based implementation of IMCFish. Examining (3.3), we find that while our updates to are local, individual rows of $\mathbf{L}$ and $\mathbf{R}$ cannot be updated in parallel; rather, *stripes* of $\mathbf{L}$ and $\mathbf{R}$ need to be processed. We retain the chunking strategy that we have described above for GPUFish and divide Ω into q^2 chunks divided across q rounds. Given q stripes of $\mathbf{A}$ and $\mathbf{B}$ there are q^2 combinations of the stripes of $\mathbf{A}$ and $\mathbf{B}$; each chunk of Ω will be responsible for performing gradient updates with a unique combination of stripes. As with GPUFish, we launch q GPU kernels each containing q blocks. The q^{th} stripe of $\mathbf{A}$ and the q'^{th} stripe of $\mathbf{B}$ used by a single chunk is determined by the index of that block and the current round.

Note that in each gradient update of IMCFish, we only require a small portion of the feature matrices $\mathbf{A}$ and $\mathbf{B}$. In particular, a single gradient update only requires $\frac{d_2}{q}$ feature values from $\mathbf{A}$ and $\frac{d_2}{q}$. Given n_r rows and n_c columns we require $\frac{n_r * d_1}{q}$ features from $\mathbf{A}$ and $\frac{n_c * d_2}{q}$ features from $\mathbf{B}$ to perform all the gradient updates in a

Original Rating	1	2	3	4	5	Overall	Runtime(s)
GPUFISH: ML 100k	80%	77%	58%	71%	87%	**72%**	**0.30**
1-Bit: ML 100k	79%	73%	58%	75%	89%	**73%**	47
GPUFISH: ML 1m	86%	74%	55%	75%	92%	**74%**	**1.1**
1-Bit: ML 1m	84%	76%	53%	77%	94%	**75%**	3130

Table 1. *A comparison between 1-bit matrix completion from [1] and the 1-bit matrix completion implemented in* GPUFISH. GPUFISH *produces results on par with the traditional 1-bit approach and is able to do so in a fraction of the runtime.*

given round. Therefore, this scheme reduces the on-GPU memory requirement by a factor of q for each feature matrix; if q is chosen large enough, this can be a significant space complexity improvement over standard stochastic gradient descent algorithms, and enables inductive matrix completion even on memory-limited GPUs. Moreover, at the beginning of each epoch, we compute an estimate for each point in Ω using the GPU-enabled linear algebra library, cuBLAS. The computation of the gradient for a single data sample in Ω requires only a single row of $\mathbf{A}$ and a single row of $\mathbf{B}$.

4. PERFORMANCE EVALUATION

All experiments were performed on a Dell workstation equipped with: a 6-core Xeon E5-2620 v3 CPU, 64GB of RAM, a 256GB Class 30 SSD, and an NVIDIA GeForce GTX 1080 GPU. For our experiments, we use Linux 3.10.0-327 along with NVCC V8.0.26.

GPUFish. We test the ability of GPUFISH to make predictions in the collaborative filtering environment on real-world data. Specifically we make predictions about user interest in movies for the Movielens (100k , 1m and 20m) data set [13]. Where possible we compare our results to those produced from the code released with [1].

We transform the user-movie ratings from the Movielens data set (integers in $[1, 5]$) to one-bit observations by subtracting the average over all ratings (approximately 3.5) from each rating and recording the sign. Our input parameters, including rank, are again determined by a grid search. Each instance of GPUFISH was terminated after 20 epochs. For each Movielens data set (100k, 1M and 20M) we remove 5,000 ratings for testing purposes, and train the model with the remaining ratings. In Table 1 we present the percentage of one-bit ratings correctly recovered by GPUFISH as a function of the original rating. We also display the overall percentage of ratings correctly recovered as well as the runtime of the algorithm.

We empirically determine the number of blocks per kernel (the number of chunks in a round) that results in the smallest run time. The results are presented in Figure 2. For each epoch we perform two processes in parallel: gradient updates on the GPU, and the permuting and chunking Ω; the run time of each epoch is the maximum of the time taken by either of these two processes. Examining Figure 2, we see that executing GPUFISH with a larger number of blocks per kernel can only decrease our runtime to the extent that it is no longer determined by the execution of gradient updates. At approximately 30 blocks per kernel our GPU gradient updates can be performed faster than our permutations and chunking of Ω, and therefore we no longer see a decrease in runtime beyond this level of block granularity.

IMCFish. We now test IMCFISH on a synthetic matrix dataset. Where possible we compare our results to those produced from the code released with [9].

We present phase transitions for the recovery of a random ma-

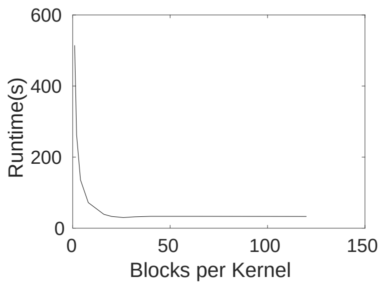

Fig. 2. *The runtime of 20 epochs of* GPUFISH *vs the number of blocks per kernel on MovieLens (20M)*

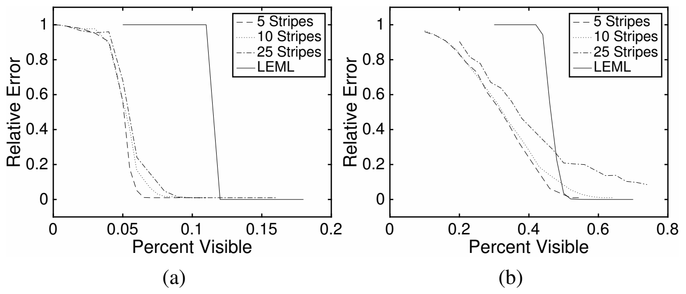

Fig. 3. *Phase transitions for the recovery of a rank one (a) and rank ten (b) matrices using* IMCFISH *and LEML.*

trix, $\mathbf{M} \in \mathbb{R}^{1000\times1000}$ with rank-one and rank-ten latent feature spaces $\mathbf{Z} \in \mathbb{R}^{50\times50}$. $\mathbf{Z}$ is the product of a matrix in $\mathbb{R}^{50\times r}$ and another in $\mathbb{R}^{r\times50}$; the entries of both matrices are drawn from the standard normal distribution. To create our random feature matrices $\mathbf{A}, \mathbf{B}$, we draw from the standard normal distribution two matrices in $\mathbb{R}^{1000\times50}$. We obtain $\mathbf{M}$ by multiplying our feature matrices with $\mathbf{Z}$.

The phase transitions for IMCFISH with 5,10 and 25 stripes are presented in Fig. 3, as are the phase transition for the LEML [14]-based IMC algorithm detailed in [9]. Given the output of IMCFISH $\hat{\mathbf{Z}}$, and the IMC algorithm's estimate of the latent feature space as the "ground truth" $\mathbf{Z}$, we compute the relative error of this estimate $\frac{\|\hat{\mathbf{Z}}-\mathbf{Z}\|_F^2}{\|\mathbf{Z}\|_F^2}$. In the same manner as GPUFISH, we manually tune our step size and regularization term for optimal recovery of $\mathbf{Z}$. In Fig. 3 the relative error of our recovery is plotted against the fraction of visible entries of $\mathbf{M}$. We note that all versions of IMCFISH, though they only use a portion of the feature matrices to perform gradient updates, are able to recover $\mathbf{Z}$ with fewer visible entires than the IMC algorithm proposed in [9].

Acknowledgements. This work was supported in part by grant #CCF-1566281 from the National Science Foundation and a GPU grant from NVIDIA.

5. REFERENCES

[1] M. Davenport, Y. Plan, E. van den Berg, and M. Wootters, "1-bit matrix completion," *Information and Inference*, vol. 3, no. 3, pp. 189–223, 2014.

[2] P. Jain and I. Dhillon, "Provable inductive matrix completion," *arXiv preprint arXiv:1306.0626*, 2013.

[3] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for large-scale matrix completion," *Math. Prog. Comput.*, vol. 5, no. 2, pp. 201–226, 2013.

[4] Emmanuel J Candès and Benjamin Recht, "Exact matrix completion via convex optimization," *Found. Comput. Math.*, vol. 9, no. 6, pp. 717–772, 2009.

[5] E. Candès and T. Tao, "The power of convex relaxation: Near-optimal matrix completion," *IEEE Trans. Inform. Theory*, vol. 56, no. 5, pp. 2053–2080, 2010.

[6] E. Candès and Y. Plan, "Matrix completion with noise," *Proceedings of the IEEE*, vol. 98, no. 6, pp. 925–936, 2010.

[7] B. Recht, "A simpler approach to matrix completion," *J. Machine Learning Research*, vol. 12, no. Dec, pp. 3413–3430, 2011.

[8] K.-Y. Chiang, C.-J. Hsieh, and I. Dhillon, "Matrix completion with noisy side information," in *Adv. Neural Inf. Proc. Sys. (NIPS)*, 2015, pp. 3447–3455.

[9] N. Natarajan and I. Dhillon, "Inductive matrix completion for predicting gene–disease associations," *Bioinformatics*, vol. 30, no. 12, pp. i60–i68, 2014.

[10] H.-F. Yu, P. Jain, P. Kar, and I. Dhillon, "Large-scale multi-label learning with missing labels," in *International Conference on Machine Learning*, 2014, pp. 593–601.

[11] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, "Parallel CCD++ on GPU for Matrix Factorization," in *Proc. General Purpose Computing Conference on GPUs*, 2017, pp. 73–83.

[12] D. Shin, S. Cetintas, K.-C. Lee, and I. Dhillon, "Tumblr blog recommendation with boosted inductive matrix completion," in *Proc. ACM Conf. Inf. Knowledge Management (CIKM)*. ACM, 2015, pp. 203–212.

[13] F. M. Harper and J. Konstan, "The MovieLens datasets: History and context," *ACM Trans. Interactive Intelligent Systems (TiiS)*, vol. 5, no. 4, pp. 19, 2016.

[14] H.-F. Yu, P. Jain, P. Kar, and I. Dhillon, "Large-scale multi-label learning with missing labels," in *Proc. Int. Conf. Machine Learning*, jun 2014, vol. 32.