CuCCD: Implementation of CCD in Cuda

Sangeeta Kumari
The Ohio State University
kumari.14@osu.edu

ABSTRACT

Matrix factorization (MF) finds its application in many algorithms like collaborative-filtering-based recommendation, data compression, word embedding, and topic modeling. Coordinate descent (CD) has been proved to be an effective technique for matrix factorization in recommender systems as they have a more efficient update rule compared to Alternating Least Squares (ALS) and are faster and have more stable convergence than Stochastic Gradient Descent (SGD). Cyclic Coordinate Descent (CCD) is one of the CD algorithms and the goal of this work is to study the parallelization challenges of CCD and optimize it for running it on a single GPU.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

KEYWORDS

Matrix Factorization, Recommender Systems, Coordinate Descent, CCD

ACM Reference Format:
Sangeeta Kumari. 2018. CuCCD: Implementation of CCD in Cuda. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given a sparse m x n matrix A, where an entry A_{ij} denotes a rating by user i for item j, the matrix factorization problem is that of finding two dense matrices W (m x k "user" matrix) and H (n x k "item" matrix) such that the product WH^T approximates the non-zero elements in A as closely as possible. We can interpret this low-rank matrix factorization as a transformation that maps each user and each item to a feature vector (either w_i or h_j) in a latent space R_k. For each missing entry in the sparse matrix A, the product of W with H^T is used as the predicted rating.

The matrix factorization problem is stated as follows:

$$\min_{\substack{W \in R^{m\times k} \\ H \in R^{n\times k}}} \sum_{(i,j)\in\Omega} \left(A_{ij} - w_i^T h_j\right)^2 + \lambda \left(||W||_F^2 + ||H||_F^2\right) \quad (1)$$

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

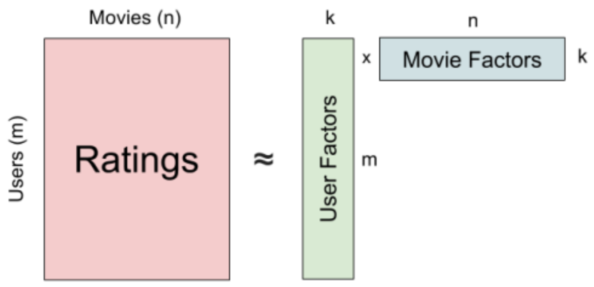

Figure 1: Matrix Factorization for recommender systems

where $A \in^{m\times n}$ is the observed rating matrix, m is the number of users, n is the number of items, λ is a regularization parameter to avoid over-fitting, and Ω is the set of observed ratings.

Once ratings matrix A has been filled i.e. mapping of each item ad user to latent factors has been complete, entries with high values can be used to make recommendations, since it is predicted that the user would rate the item highly.

2 RELATED WORK

Matrix factorization methods are widely used for extracting latent factors for low rank matrix completion and rating prediction problems arising in recommender systems [10] of on-line retailers. There are mainly three optimization techniques for problem (1) namely ALS [9], SGD [8] and CD [3].

2.1 Alternating Least Squares

ALS alternatively switches between updating W and updating H while fixing the other factor thereby updating W and H independently. Although the time complexity per iteration is $O(|\Omega|k^2 + (m+n)k^3)$, [9] shows that ALS is inherently parallelizable as each row of W or H can be updated independently from the updates of other rows and [4] have different variations of it to speed it up. However, parallelization of ALS in a distributed system when W or H exceeds the memory capacity of a computation node is more involved. On a single GPU, [6] has optimized memory access by reducing discontiguous memory access, retaining hotspot variables in faster memory, and using registers aggressively.

2.2 Stoachastic Gradient Descent

SGD loops through all ratings in the training set. For each given training case, the system predicts A_{ij} and computes the associated prediction error R_{ij}:

$$R_{ij} = A_{ij} - w_i^T h_j \quad (2)$$

and then modifies the parameters by a magnitude proportional to the learning rate η in the opposite direction of the gradient yielding:

$$w_i \leftarrow w_i - \eta(\lambda w_i - R_{ij} h_j) \tag{3}$$

$$h_j \leftarrow h_j - \eta(\lambda h_j - R_{ij} w_i) \tag{4}$$

For each rating A_{ij}, SGD needs O(k) operations to update w_i and h_j. If we define ÎI' consecutive updates as one iteration of SGD, the time complexity per SGD iteration is thus only $O(\Omega k)$. Compared to ALS, it is faster in terms of the time complexity for one iteration. However, since the w and h updates are not independent and updates for the ratings in the same row or same column of A involve same variables, parallelizing SGD techniques has been difficult although a variety of schemes like DSGD [7], HogWild [1], DSGD++ [2], FPSG [8] and GPUSGD [5] have been devised to improve its performance. Besides being very sensitive to dataset and learning rate changes, SGD is also slow to converge.

2.3 Coordinate Descent

The basic idea of coordinate descent is to update a single variable at a time while keeping others fixed. CCD and CCD++ [3] algorithms have been developed with this update methodology. Although CCD++, where the update is done per feature, is a more efficient variant of CCD and [4] has optimized it for GPU, this workâĂŹs focus is on parallelizing CCD.

CCD updates rows of W and columns of HT in alternating fashion. Therefore, the update sequence for a full iteration for CCD is w_1,..,w_m and h_1,..,h_n, where w_i is a row of W, and h_j is a column of HT. If only one variable wit is allowed to change to z while fixing all other variables, the one-variable subproblem is formulated as:

$$\min_z f(z) = \sum_{j \in \Omega_i} (A_{ij} - (w_i^T h_j - w_{it} h_{jt}) - z h_{jt})^2 + \lambda z^2 \tag{5}$$

And the unique value of z_* is obtained as:

$$z^* = \frac{\sum_{j \in \Omega_i} (A_{ij} - w_i^T h_j - w_{it} h_{jt}) h_{jt}}{\lambda + \sum_{j \in \Omega_i} h_{jt}^2} \tag{6}$$

Although computing z_* from scratch takes $O(|\Omega_i| k)$ time, if we maintain a residual matrix, R as below, the time is reduced to $O(|\Omega_i|)$.

$$R_{ij} \equiv A_{ij} - w_i^T h_j, \forall (i,j) \in \Omega \tag{7}$$

Therefore, it overcomes the challenges of both ALS and SGD by getting rid of any parameter sensitiveness and allowing a faster update and convergence with inherent parallelizable capability.

Algorithm 1 Sequential CCD

Input: Training matrix A, number of features K, regularization parameter λ
Output: W, H and modified R

```
1:  Initialize W, H with UniformReal values
2:  Initialize residual matrix R = A - WH
3:  for $iter = 1$ to $max\_iterations$ do
4:      for $A_{i*} \in A$ do
5:          for $k = 1$ to $K$ do
6:              $z^* = \frac{\sum_{j \in A_{i*}} (R_{ij} + W_{ik} H_{jk}) H_{jk}}{\sum_{j \in A_{i*}} (\lambda + H_{jk}^2)}$
7:              $R_{ij} -= (z^* - W_{ik}) H_{jk}$
8:              $W_{ik} = z^*$
9:          end for
10:     end for
11:     for $A_{*j} \in A$ do
12:         for $k = 1$ to $K$ do
13:             $s^* = \frac{\sum_{i \in A_{*j}} (R_{ij} + W_{ik} H_{jk}) W_{ik}}{\sum_{i \in A_{*j}} (\lambda + W_{ik}^2)}$
14:             $R_{ij} = R_{ij} - (s^* - H_{jk}) W_{ik}$
15:             $H_{jk} = s^*$
16:         end for
17:     end for
18: end for
```

3 CCD ON GPU

3.1 Thread Assignment

In order to get coalesced access across a warp, a warp is assigned to each row of the residual sparse matrix in the kernel updating W. For similar reason, a warp is assigned to a column of sparse matrix while updating H. To update one row of R, we have to read all non zero elements in the corresponding columns of H and one value from W of that row. Iterating over all K values of W leads to updating of one user. Since each thread in a warp has its own values for z_star and h_square, we need to reduce the values in the warp. I have used __shfl__down for the reduction.

3.2 Coalesced access

In CCD, we access the number of nonzero elements (nnz) in the sparse residual matrix in a column fashion while updating H and in a row fashion while updating W. Thus, saving the R elements in just CSR or CSC representation will lead to uncoalesced access of R in at least one update. To avoid uncoalesced access, both the CSC (which I will be referring as Rt) and CSR representation (R) is used. CSC representation is used for updating H and CSR representation is used for updating W. When we consider updates to H, access pattern of W becomes important and vice-versa in case of updates to W. To avoid the uncoalesced access to W while updating H, it is better to keep W in transposed manner. This change does not effect the update of W because when W is updated, one element is accessed by all threads in a warp (as per our thread assignment).

Algorithm 2 Base version of CCD for GPU

```
1: for $iter$ = 1 to $max\_iterations$ do
2:     <<< Update W Device Kernel >>>
3:     $row\_start = R\_row\_ptr[warp\_id]$
4:     $row\_end = R\_row\_ptr[warp\_id + 1]$
5:     for $j$ = 1 to $number\_features$ do
6:         $h\_square = 0$
7:         $z\_star = 0$
8:         for $j = lane\_id$ to $R.nnz\_rows(j)$ do
9:             $col\_idx = R\_col\_idx[row\_start + i]$
10:            $z\_star += (R\_csr\_value[row\_start+i]+W_{it}*H_{jt})*H_{jt}$
11:            $h\_square += H_{jt} * H_{jt}$
12:        end for
13:        Reduce $z\_star$ and $h\_square$ across the warp using warp primitives
14:        __syncthreads()
15:        $z\_star = z\_star/(\lambda + h\_square)$
16:        for $j = lane\_id$ to $R.nnz\_rows(j)$ do
17:            $UpdateR_{ij}$
18:        end for
19:        if $lane\_id == 0$ then
20:            $W_{it} = z\_star$
21:        end if
22:    end for
23:    <<< Update Rt Device Kernel >>>
24:    <<< Update H Device Kernel >>>
25: end for
```

Dataset	Synthetic	Netflix
m	6040	480,189
n	3952	17,770
nnz_train	900189	99,072,112
nnz_test	100020	1,408,395
k	10	40
max_iterations	5	10

Table 1: Dataset description

Algorithm 3 CCD for GPU with Binning

```
1: for $iter$ = 1 to $max\_iterations$ do
2:     for $bin$ = 1 to $NUM\_BINS$ do
3:         $device\_update\_W$ <<< grid, block, 0, streams[bin] >>>
           $(H, W, R\_CSR, rowBinPtr[bin], rowCount[bin])$
4:     end for
5:     for $bin$ = 1 to $NUM\_BINS$ do
6:         $device\_update\_Rt$ <<< grid, block, 0, streams[bin] >>>
           $(H, W, R\_CSC, colBinPtr[bin], colCount[bin])$
7:     end for
8:     for $bin$ = 1 to $NUM\_BINS$ do
9:         $device\_update\_H$ <<< grid, block, 0, streams[bin] >>>
           $(H, W, R\_CSC, colBinPtr[bin], colCount[bin])$
10:    end for
11: end for
```

3.3 Load Imbalance

If warps within a block do not all execute for the same amount of time, the workload is said to be unbalanced. This means there are fewer active warps at the end of the kernel, which is a problem known as "tail effect". In order to have a more balanced workload among the warps in each block, we can divide the users/items in groups such that each warp in a block has considerably uniform work distribution. The grouping can be done either by explicit sorting of the rows(/columns) according to the the nnz values and then chunking that in order or by grouping row(/column) indexes with similar nnz values in groups. Sorting is a very expensive task so I chose the second approach. One thing to note is there does exist efficient histogram implementations in Cuda as in [11].

There are yet again two ways of doing grouping with the second approach. The upper and lower limits of each bin can be either decided dynamically according to the nnz values in a row/column or can be fixed beforehand. The first method does give a better distribution across each bin but also adds the overhead of dynamic limit creation. To avoid this, I decided to fix the lower and upper limits of each bin and skip the kernel execution in case a bin has no elements.

3.4 Data Reuse

While binning reduces the load imbalance between warps, uncoalesced access to W and H still amounts to a significant loss in performance. Consider the kernel which updates W. The only elements of H that gets access are the ones that corresponds to the location of R thus resulting in uncoalesced access of H and therefore more data movement from cache. Although we cannot get rid of the access pattern and get coalesced access, we can definitely reduce the severity by using tiling, which will restrict the size of H being considered. Similarly, we can reduce the size of W being considered at a time when updating H.

4 EXPERIMENTS AND RESULTS

Though I do not have root mean square error (RMSE) for the datasets that I had expected to report results on, I have compared the performance of CCD++, CCD on CPU without multithreading, base GPU version of CCD and the binned CCD on a synthetic dataset that can be found in [6] and the same can be seen in Figure 2. The dataset and parameters chosen to train the datasets have been mentioned in Table 1.

5 FUTURE WORK

Due to some errors, I could not run the datasets that I had planned. The tiled version of the code is implemented half and needs to be completed. Occupancy was not closely monitored which should be done to decide block size so that we ensure full or near to full occupancy.

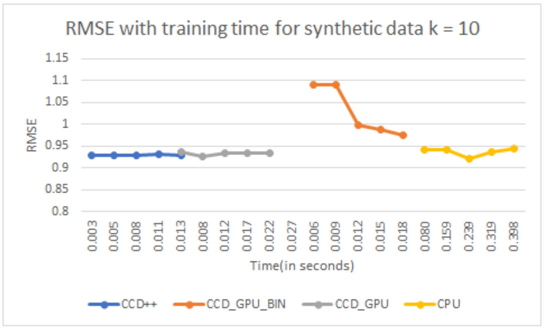

Figure 2: RMSE with training time for synthetic data with k = 10

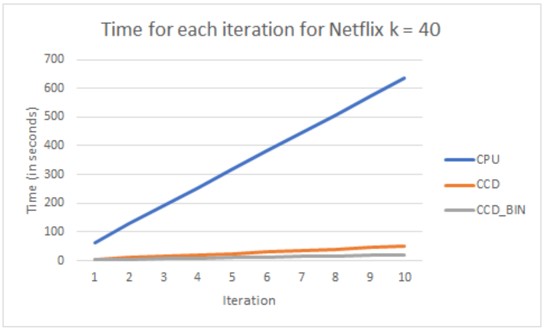

Figure 3: Cumulative time for each iteration for Netflix dataset with k = 40

6 CONCLUSIONS

It was observed that CCD++ converged faster as compared to CCD, though I have used it only for the synthetic dataset. There is a significant speed up in the Cuda version as compared to the sequential CPU version. In addition, binning gave a speedup of around 40-50% in both synthetic as well as Netflix dataset.

REFERENCES

[1] Stephen Wright Benjamin Recht, Christopher Re and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems* (2011), 693–701.
[2] Faraz Makari Christina Te.ioudi and Rainer Gemulla. 2012. Distributed matrix completion. *IEEE 12th International Conference on Data Mining* (2012), 655–664.
[3] Si Si Hsiang-Fu Yu, Cho-Jui Hsieh and Inderjit Dhillon. 2012. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. *Proceedings of the 12th International Conference on Data Mining* (2012), 765–774.
[4] Rakshit Kunchum Israt Nisa, Aravind Sukumaran-Rajam and P. Sadayappan. 2017. Parallel CCD++ on GPU for matrix factorization. *GPGPU@PPoP* (2017), 73–83. https://doi.org/10.1145/3038228.3038240
[5] Su Hu Jing Lin Jing Jin, Siyan Lai and Xiaola Lin. 2015. GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization. *Concurrency and Computation: Practice Experience* 28 (2015), 3844–3865.
[6] Israt Nisa. 2017. Toy Dataset in CCD++. https://github.com/cuMF/cumf_ccd.
[7] Peter J Haas Rainer Gemulla, Erik Nijkamp and Yannis Sismanis. 2011. Largescale matrix factorization with distributed stochastic gradient descent. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), 69–77.
[8] Yu-Chin Juan Wei-Sheng Chin, Yong Zhuang and Chih-Jen Lin. 2015. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems Technology* 6, 1 (2015).
[9] Robert Schreiber Y. Zhou, Dennis Wilkinson and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management* (2008), 337–348.
[10] Robert Bell Yehuda Koren and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.
[11] K. K. Yong and S. S. O. Talib. 2016. Histogram optimization with CUDA. (2016), 312–318.