CUDA-based Parallel Implementation of Collaborative Filtering

Abinash Sinha
University of Minnesota
sinha160@umn.edu

Achal Shantharam
University of Minnesota
shant012@umn.edu

Manish Keshri
University of Minnesota
keshr005@umn.edu

Prashanth Venkatesh
University of Minnesota
venka220@umn.edu

ABSTRACT
In this paper, we describe the implementation of collaborative filtering algorithm more specifically the user-based collaborative filtering using GPU. Collaborative filtering are of two types: memory-based and model-based. User-based falls under the category of memory-based. The overwhelming amount and rate of increase in amount of data mandates an efficient information filtering technique using which we can gather useful information for use. Collaborative filtering is one of these techniques used in recommender systems. predicts user preferences in item selection based on the known user ratings of items. Here we talk about how to improve the time taken by recommendation generation in terms of generating ratings that a user will give to an item he hasn't rated by looking at K users who are most similar to that user. We have optimized the time taken by each sub-process involved in this algorithm using CUDA in order to be run with most efficient running time in Nvidia GPUs.

Keywords
User-based Collaborative Filtering, Similarity Matrix, CUDA, Tiling

1. INTRODUCTION

Collaborative filtering (CF) predicts user preferences in item selection based on the known user ratings of items. As one of the most common approaches to recommender systems, CF has been proved to be effective for solving the information overload problem. CF can be divided into two main branches: memory-based and model-based as can be seen below:

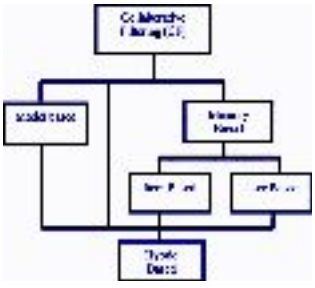

Figure 1. Types of collaborative filtering [2]

Memory-based filtering can further be divided into two types:

1. User-based filtering - This type of filtering finds out rating of a particular item unrated by a user on the basis of top K users which are similar to that user. Therefore, the main concern lies in finding out the similarity between users for which cosine similarity or Pearson correlation measure is utilised.

$$\text{Usersim}(u_t, u) = \frac{\sum_{u \in I_{u,u_t}} (R_{u,i} - \bar{R}_u)(R_{u_t,i} - \bar{R}_{u_t})}{\sqrt{\sum_{u \in I_{u,u_t}} (R_{u,i} - \bar{R}_u)^2} \sqrt{\sum_{u \in I_{u,u_t}} (R_{u_t,i} - \bar{R}_{u_t})^2}}$$

2. Item-based filtering - Here, instead, we find out similarities between items based on the ratings given by the users. This similarity measure can take many forms, such as correlation between ratings or cosine of those rating vectors like discussed above for user-based filtering.

We will focus on user-based filtering, where the main bottleneck in terms of computation is calculating similarities between users. This measure can be best visualized in terms of a similarity matrix by using the source ratings matrix where each row corresponds to user and each column corresponds to item. The value in each cell of this matrix basically tells us the rating given by the user (corresponded by row) for the item (corresponded by column). If the number of users and items are large then computing this similarity matrix would be a humongous task. Time complexity of calculation of similarity matrix would be O(m*m*n) where m represents the number of users and n represents the number of items.

This is where GPU could come to rescue given its capability to handle compute-intensive tasks like the case above. Dealing with matrix-related computations is a well known use case for GPU to handle in an efficient manner. A GPU (graphics processing unit) is a specialized type of microprocessor, primarily designed for quick image rendering. GPUs appeared as a response to graphically intense applications that put a burden on the CPU and degraded computer performance. They became a way to offload those tasks from CPUs, but modern graphics processors are powerful enough to perform rapid mathematical calculations for many other purposes apart from rendering. GPU nowadays is being heavily used for decreasing the time taken by such tasks by a huge factor as could be seen with the results of our research here.

Notations used:

1. S = m x m similarity matrix where m represents the number of users
2. R = n x m ratings matrix where n represents the number of items or movies in our case

A related work in this field was done in [1]. The approach followed in their research work was:

1. Each thread is basically calculating each value of similarity matrix, S
2. Each block takes care of configured TILE_WIDTH x TILE_WIDTH size of tile in similarity matrix, S
3. Each tile of S requires to load two corresponding TILE_WIDTH * n rectangles in ratings matrix, R
4. Threads calculate partial correlation coefficient independently
5. Partial correlation is accumulated in shared memory

2. DESIGN

2.1 Opportunities Identified

We identified several opportunities for parallelization and improvement in the parts of collaborative algorithm.

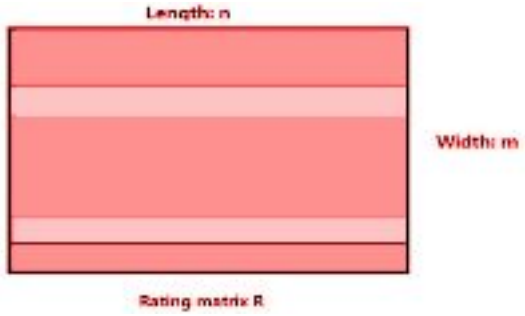

Figure 2. Ratings Matrix

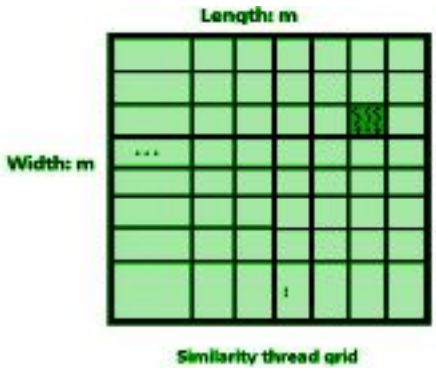

Figure 3. Similarity Matrix

Following are the use cases identified as our requirements to be met for optimization:

1. The ratings matrix, R is represented with m rows and n columns where m and n are the number of users and the number of items respectively.
   There could possibly be a benefit in transposing this original matrix and then performing processing of other remaining parts of algorithm. The problem of transposing matrix could be optimized.

2. Average of ratings of all **n items** of each user is independent, $\mathbf{R}_{\mathbf{user,average}}$
   Each thread could calculate the average of all ratings of all items given by that user. While calculating the average only the items which are given non-zero ratings are considered. This could either be done row wise since each row represents the user or column-wise by taking the transpose of ratings matrix in order to ensure coalescing. There is no reuse here.

3. Norm of difference between rating, $R_{user,item}$ and average, $\mathbf{R}_{\mathbf{user,average}}$ for each item, i of all items $\Sigma(R_{user,item} - \mathbf{R}_{\mathbf{user,average}})^2$
   This could be done in a separate kernel or we could do its calculation on the fly when average is being calculated and store them in global memory to be re-used later.

4. Calculation of dot product of each of $(R_{user\_i,\ item} - \mathbf{R}_{user})$ with $(R_{user\_j,\ item} - \mathbf{R}_{user\_j})$ where user_i and user_j $\in$ U and U denotes the set of all **m users**. This could be visualized as **m x m matrix** basically $A^2 = A.A^T$
   This could be treated as a famous use case of matrix multiplication which or we could perform this calculation just using the ratings matrix, R. The transpose which is already calculated above could be re-used here.

3. IMPLEMENTATION

3.1 Optimizations implemented

Due to practical considerations, all the optimizations mentioned in section 2 could not be implemented. We describe some of the optimizations that we actually implemented in code and why we chose to implement it this way.

1. *Calculation of average rating values could not be done using reduction*:
   The main reason why we could not implement reduction for calculating the average was that we needed to average only the non-zero rating values. It was not possible to know the total number of non-zero values at the time of calculating the average without scanning the entire ratings of items for each user at least once. We transposed the original ratings matrix, R using tiled-transpose kernel. Instead, we implemented a column-parallel (for memory coalescing) average calculation kernel where each thread calculates the average of an entire column. Additionally, the average calculation is done by taking the transpose of the ratings vector to ensure coalesced memory accesses. Since the kernel operates on the transpose of the ratings matrix,

each thread calculates the average rating values for each user. We also discuss an optimization at the time of reading the data from the file which would help us in calculating the average values by reduction in the later sections of the paper.

2. *Calculation of the norm values in the average kernel:* While writing the kernel for the calculation of the norm values, we quickly realized that the norm values calculation can be done in the same kernel as that of the average calculation.

3. *Parallel implementation of the matrix transpose operation:* The main kernel, the one that calculates the similarity matrix takes as input the transpose of the ratings matrix. In order to do that, we wrote a kernel that does the matrix transpose operation in parallel using shared memory for coalesced accesses.

4. *Parallel implementation of the similarity matrix calculation:* Calculation of the similarity matrix is the part of the collaborative filtering algorithm that we believe can be parallelized the most. This is where we claim to achieve the most speedup. We first implemented a kernel that calculates the similarity matrix in a parallel fashion using just the global memory. We immediately identified that this is a kernel that can benefit greatly by using shared memory as there is a lot of reuse of the data.. The speedup over the serial implementation that we obtained for an input matrix of 600 x 9000 entries is around 260 times. The results are explained in detail in the next section.

5. *Parallel and tiled implementation of the similarity matrix calculation:* Having identified the points of data reuse, we implemented a kernel that calculates the similarity matrix by leveraging shared memory. We experimented with different tile sizes and the next section describes the results in detail.

3.2 Data Flow through Kernels

Our implementation consists of the following modules and kernels:

1. *Data reader:* This module is responsible for reading the input file which contains the ratings matrix.

2. *Matrix transpose kernel:* This is a kernel function that calculates the transpose of a matrix in parallel.

3. *Average and norm kernel:* This is a kernel function that computes the average and norm values for each row or user in the ratings matrix in parallel.

4. *Similarity matrix kernel (using global memory):* This is a kernel function that computes the values of the similarity matrix in parallel using global memory. Each thread is responsible for calculating the similarity value between a pair of users. In other words, each thread computes one entry in the similarity matrix.

5. *Similarity matrix kernel (using shared memory):* This is a kernel function that computes the values of the similarity matrix in parallel using shared memory and tiling. Each thread loads TILE_WIDTH x TILE_WIDTH elements from the ratings matrix and is responsible for calculating the similarity value between a pair of users.

Figure 4. describes the data flow through these kernels. We first read the data from the CSV file using the data reader and load it into a matrix. We then take the transpose of the ratings matrix by calling the transpose matrix kernel. Later, we calculate the average and norm values for each of the users by calling the *getAverageAndNorm* kernel. Once we have the average and norm values along with the transpose of the ratings matrix, we pass the pointers to these to the main kernel that calculates the similarity matrix values.

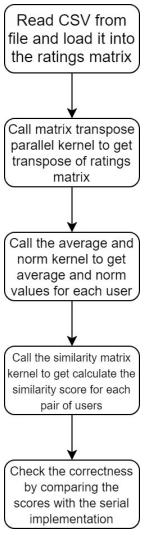

Figure 4. Data Flow through Kernels

4. VERIFICATION

For verifying correctness, we have used a serial implementation of the collaborative filtering algorithm as a benchmark. We run the serial implementation first to calculate the similarity matrix and then run the parallel version. After execution, we compare these 2 matrices element by element.

We consider the values to be acceptable if the absolute difference between the serial and parallel implementations falls below a certain threshold. We chose this threshold by conducting experiments with the recommendations that get generated after the similarity matrix values have been obtained. We found that if the absolute difference similarity values lie below 1e-5, the same recommendations were generated. Hence, chose that value for our threshold. Additionally, we use single precision floating point for our calculations since the ratings values are discrete ranging from 1 to 5 in steps of 0.5. For these values, we found that the single precision floating point representation is sufficient to guarantee 100% accuracy.

5. PERFORMANCE

We have analyzed the performance of our collaborative filtering algorithm on the MovieLens small dataset. The dataset consists of 629 users who have rated 9000 movies. Thus, the size of the ratings matrix used is 629*9000. We have parallelized our implementations of Average and Norm values computations as well as the computation of Similarity Matrix**.**

Table 1. Average and Norm computations CPU and GPU

Host	Time taken in seconds
CPU	0.0156
GPU (without coalescing)	0.0075
GPU (coalesced access)	0.0016

Average and Norm computation requires each thread to compute the average of a particular user. Therefore, each thread computes average of one row. As we can see from the above results naïve implementation of Average and Norm in GPU was able to achieve a speedup of 2x compared to the CPU code. When we used transposed version of the rating matrix, we were able to achieve coalesced memory access thus improving the speedup to about 10x. However computing transpose of the rating matrix introduces a small overhead of 0.0004s. This overhead is significantly less compared to the performance gain obtained.

Table 2. Similarity matrix computations CPU vs GPU

Host	Time taken in seconds
CPU	7.6047
GPU (global memory)	0.0322
GPU (tiled memory access)	0.0192

Computing the similarity matrix is the main performance bottleneck in the applications. The parallelized GPU version using global memory is 262x faster compared to the CPU serial version. Higher performance was achieved using tiling operation and shared memory. The speedup was about 400x with tiling operation. The tiled version takes advantage of the precomputed transposed version of the ratings matrix stored in device memory, thus no performance overhead is introduced.

Table 3. Analysis of GPU execution time with variations of tile size on Tesla 1080 GPU

Tile size	Time taken in seconds	Speedup
8*8	0.039	192x
16*16	0.021	361x
32*32	0.019	402x

As we can see from the above table best execution speedup of 400x was achieved when we made use of tile size of 32*32. There was a huge performance improvement when we changed the tile size from 8*8 to 16*16 since the occupancy in the GPU significantly increased.

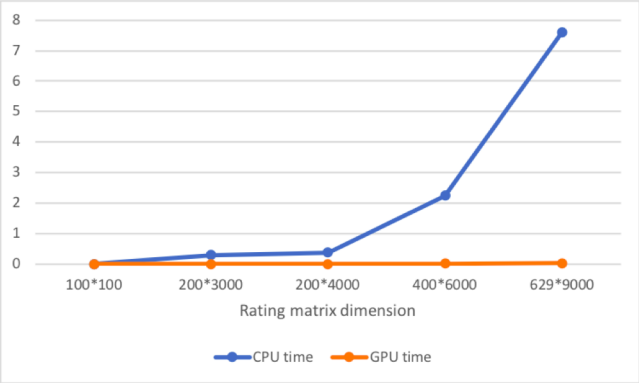

Figure 5. Computation time variation with increase in size of Matrix CPU vs GPU

We performed an analysis of the impact of the matrix size with the performance of GPU and CPU. As we can see from the above chart as the matrix dimensions increases the CPU computation time increases exponentially. The increase in GPU computation time is linear. The speedup obtained with GPU implementation increases with increase in data size. Thus, we claim that our parallel implementation is scalable even for larger datasets.

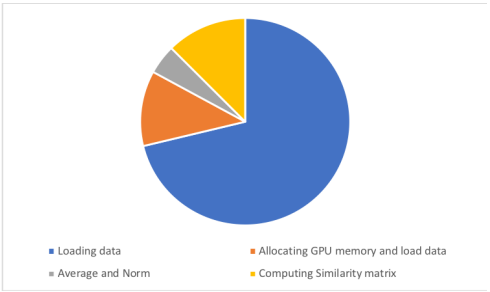

Figure 6. Performance analysis by time taken by each task to compute

We also performed an analysis to determine the time taken by each task. As we can see from the above chart, most time is taken by the operation to load the dataset file from the disk. The operations involving allocating the memory in device and

transferring data from host to device is also taking significant amount of time. The computation intensive operations involving Computing Similarity matrix and calculating average and norm take the least amount of time to execute.

5.1 Performance bottlenecks

We were able to achieve our performance goals with respect to algorithm complexity and parallelization. Loading data from disk to host memory takes the most amount of time. GPU memory allocation and data transfer to device memory is another major bottleneck in the above implementation. Loading data into host memory can be improved by making use of efficient libraries written in other object oriented languages like Python and by making calls to the host code in C. Using cuda streams and handling the data in chunks would significantly improve the performance. Since the data is very sparse, if the data could be obtained in a better sparse matrix representation our performance could be further improved. MPI and GMAC based implementations will help in scaling this algorithm to larger datasets.

6. FUTURE WORK

1. The use case of calculating average of all ratings for every user could be implemented using reduction sum technique as shown below:

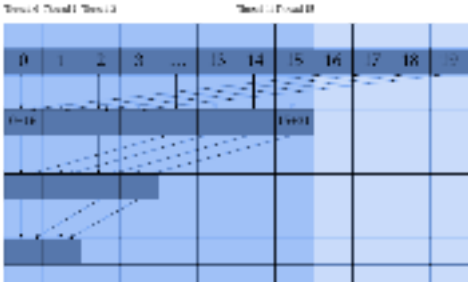

Figure 7. Diagram of reduction sum

This would be a better and more efficient solution if we know the number of items which have non-zero ratings beforehand. This count could be kept as and when a user inputs a non-zero rating for an item and the database where this matrix gets stored could keep incrementing it user-wise.
In such a case, we could have an average calculation kernel where in each thread would be responsible for each row (hence no-coalescing) and each thread could internally call another kernel (dynamic parallelism) from inside to perform reduction sum to calculate the sum of all ratings for each user and then we can calculate the average after we get this sum from this internal kernel call for reduction sum.

2. Calculation of rating of an unrated item for a user:
The calculation of rating of an item using top k users denoted by **u'** similar to given user, **u** if k is very large can become a problem.

$$r_{u,i} = \bar{r_u} + k \sum_{u' \in U} \text{simil}(u, u')(r_{u',i} - \bar{r_{u'}})$$

Although, generally only a few of the top k users are chosen. When the ratings matrix is dense which implies the users have filled ratings for many of the items which are there in ratings matrix, then in that case we might consider taking a large value of k considerable enough for optimization using GPU. In that case the product between top k similar user and absolute difference between the ratings of each of those users for that unrated item with their corresponding, $\mathbf{R_{user,average}}$ could be treated as vector multiplication which could be parallelized using GPU. Next, step could be performing reduction sum on the resultant vector that we would get. Therefore, we see a possibility of two kernels - one for vector multiplication and another for reduction sum to be utilized for efficient and fast processing.

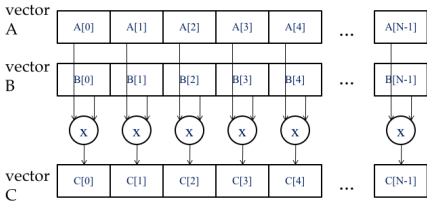

Figure 8. Diagram of vector multiplication

3. Thirdly, the loading of data into the memory could be done using double buffering mechanism and come up with a solution in order to be able to divide the processing of this algorithm in chunks if possible.

7. CONCLUSION

We divided the use case of collaborative filtering technique into several sub-cases each of which gets optimized using CUDA. Each of the kernels have a contribution in enabling the calculation of main computation of similarity matrix calculation. Like, the transpose of matrix has been implemented using shared memory in order to enable coalescing. The results of norm and average calculation done in first kernel of average calculation was getting used in next kernel of similarity matrix calculation kernel. Average calculation was done in a coalesced fashion by transposing the original ratings matrix, R. As we increase the size of data set we see GPU outperforming CPU in terms of execution.

8. REFERENCES

[1] Wang, Zhongya & Liu, Ying & Ma, Pengshan. 2014. A CUDA-enabled Parallel Implementation of Collaborative Filtering. *Procedia Computer Science. 30*. 66–74. 10.1016/j.procs.2014.05.382.

[2] Najdt Mustafa, Ashraf Osman Ibrahim, Ali Ahmed, Afnizanfaizal Abdullah. 2017. Collaborative filtering: Techniques and applications. *International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*. DOI= 10.1109/ICCCCEE.2017.7867668.

[3] David B. Kirk, Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach