User-based Collaborative Filtering Algorithm Implementation on GPU

Tianyi Xie

University of Michigan, United States

Abstract

Machine learning becomes a more and more popular topic nowadays, especially in recommendation system filed. Collaborative filtering is one of essential and efficient recommendation algorithm. However, collaborative filtering algorithm requires more powerful computation capability. In industry, we are dealing with tons of data which should be analyzed in a very short time. Therefore, it is quite necessary to have the aid of parallel computing algorithm. In this report, we are going to use GPU to realize our collaborative filtering parallel computing idea.

Keywords: Collaborative Filtering, GPU, Parallel Computing

1. Motivation

Collaborative filtering is a widely used recommendation algorithm in recommendation system. It calculates the similarity between each user based on their rating for different items. By doing this, we can find customers' favor or interest and thus make a recommendation for them. But in reality, since the numbers of users and items are huge, the serial version of collaborative filtering definitely doesn't work out since it takes too much time to run. In many cases, we may not get the result due to incredible large computing complexity. Distributed computing platforms, such as MPI, are deployed to solve the scalability problem. Other platforms, such as OpenMp, Hadoop[1] and Spark, can also help the dilemma to some extent. But they are not efficient. The performance of GPU turns out to be much better than other platforms. So in this report, we are trying to present the design of collaborative filtering algorithm in CUDA and their performance correspondingly.[2]

2. CUDA/GPU

GPU (Graphics Processing Unit) is driven by the insatiable market demand for real time, high definition 3D graphics, and has evolved into a highly parallel, multi-thread, many-core processor with tremendous computational horsepower and very high memory bandwidth. Compute Unified Device Architecture(CUDA) is a general purpose parallel computing architecture, which is more and more popular as it makes full use of the computing power of the GPUs to solve complex computational problems efficiently. Heterogeneous computing systems integrated with CPUs and GPUs have offered a new solution for parallel acceleration and gradually become a new hotspot. [3]

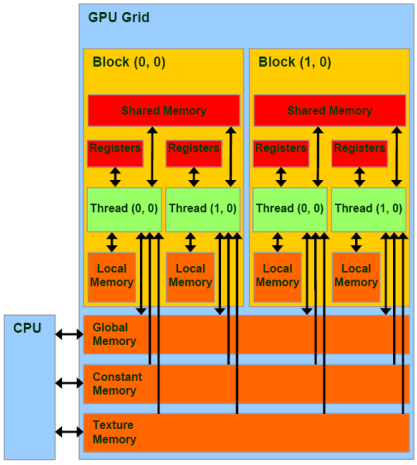

Figure 1: GPU

3. Algorithm Introduction

. Collaborative filtering is a method of making automatic predictions(filtering) about the interests of a user by collecting preferences or taste information from many users. There are many types of collaborative filtering recommendation algorithms. The most common algorithms are user-based and item-based collaborative filtering. For example, if item A have similar fans with item B, then we may recommend item A to item B's fan's group. Note that these predictions are specific to the item, by using information gleaned from many items.

. Other algorithms includes SVD approach which is also a efficient and reasonable approach to make a recommendation. We compute the singular value and singular vector. Then by selecting n largest singular value, we divide data into n similar group. Thus we can analyze common interest of customer in each group, and therefore, make a prediction based on information we have.

. There are several cool recommendation algorithms. But in this paper, we will mainly focus on user-based collaborative filtering approach.

3.1. User Based Collaborative Filtering Algorithm Introduction

As the number of users and items on the web increase dramatically, recommendation system is experiencing a fast growth and is changing the way for people to receive message. User based collaborative filtering turns out to be one of the most successful technologies in recommendation systems, which has been deployed and improved over the past decade. We first collect data from users' rating score for each item. Then, by applying collaborative filtering algorithm, we can match the user with people who share similar interests or tastes with him.

The steps of user based collaborative filtering can be divided into following steps: **similarity matrix calculation, nearest neighbors finding, item rating prediction, and item recommendation.**

Assuming we have m by n user rating matrix, m is the number of user in our database and n is number of items. The value inside the matrix is the rating of each customer for each item. Some of entries inside the matrix are missing, since each customer cannot experience every item. Actually, in the practice, the rating matrix we collect is usually a sparse matrix. Our goal is to fill out the missing entry and thus recommend the top favorite items for each users that they haven't experienced yet. On the top of rating matrix, we apply Pearson Correlation between user u_1 and user u_2

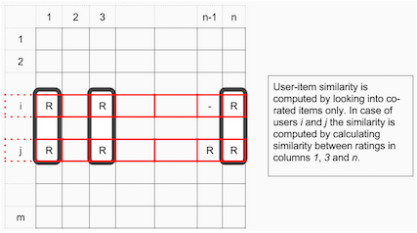

Figure 2: User-based Collaborative Filtering Method

1. **similarity matrix calculation:** Calculate the similarity matrix which dimension should be m * m. By adopting Pearson Correlation, we figure out similarity value for each entry of similarity matrix.

$$sim(i,j) = \frac{\sum_{k=1}^{n}(R_{i,k} - R_{i,avg}) * (R_{j,k} - R_{j,avg})}{\sqrt{\sum_{k=1} n(R_{i,k} - R_{i,avg})^2} * \sqrt{\sum_{k=1} n(R_{j,k} - R_{j,avg})^2}}$$

   Here $R_{i,k}$ denotes ith user with kth item rating. If the value of entry in the rating matrix is missing, we assign missing value with 0. And $R_{i,avg}$ and $R_{j,avg}$ respectively represents i and j user average rating score for all items.
2. **nearest neighbors finding:** After we find the similarity matrix, we can locate most similar user with user i based on scores in similarity matrix. The larger the correlation, the more similar the two users are. Basically, we could pick up Kth largest similarity score
3. **item rating prediction:** Then we find Kth most similar user with user i. Then we could fill out missing rating entry by taking average rating score or weighted average rating score of user i K nearest neighbor.
4. **item recommendation:** Recommend top J items to user i by predicting the missing values.

3.2. Complexity Analysis

We calculate each step time complexity in this section. Let's take a look at Table 1.

- For the first step, similarity matrix calculation, we have time complexity of O(m * m * n). This is because total there are m * m entries in similarity matrix and to calculate each entry, we should go over every entry in user i and j in the rating matrix. Totally the time complexity for the first step will be O(m * m * n).

- After finish calculating similarity matrix, we need to sort the score of similarity and then pick the most similar users. To sort value for each user, the time complexity is O(m * log(m)), and totally there are m users. Therefore, in step 2, we have total time complexity of O(m * m * log(m))

- For the item rating prediction step, we need to calculate average rating score of K users, and totally there are m users. Hence the total time complexity is O(M * K). We could regard K as a constant here. So it could also record as O(m)

- Similarly, we should recommend J item to m users. So the total time complexity in this step is O(m * J). It is O(m) if J is regarded as a constant here.

Step	**Complexity**
similarity matrix calculation	O(m * m * n)
nearest neighbors finding	O(m * m * log(m))
item rating prediction	O(m * K)
item recommendation	O(m * J)

Table 1: Time complexity for each step

4. Algorithm Explanation

From above table of time complexity, we find majority of computation come from the first two step. Actually, each entry in similarity matrix requires pairwise dot product between two vectors in rating matrix. Moreover, all steps except step share the same basic element calculation ($R_{i,k}$ - $R_{avg,i}$) and do not require independently data traverses. As a result of that, we could parallelize all four steps in CUDA separately.

. Assuming tile width TW be the width of a CUDA thread block. So for each block, there are TW * TW thread. Notice that the dimension of similarity matrix is m * m, where m is number of users. Hence, the number of block is m / TW * m / TW. Namely m / TW is the dimension of block size, Dim. Each thread is responsible for calculating one value of similarity matrix. If we use tx and ty represent position of thread inside each block, then bx and by represent position of block. Then we can compute row i and column j in CUDA kernel by following formulas:

$$i = by * Dim + ty$$

$$j = bx * Dim + tx$$

Since similarity matrix is symmetric, so it is not necessary to compute lower triangle value. Moreover, it doesn't make sense if we compute the similarity between user i and himself. In a nutshell, we could just compute upper triangle matrix. In the code we will continue computing value in similarity matrix only if j > i.

. Last but not the least, since the rating matrix could be a very sparse matrix(Not everyone can experience everything). Therefore we put data into **adjacency list** instead of adjacency matrix in order or save memory space. Because lots of values inside the rating matrix will be missing, it is not necessary to record them.

4.1. Parallel Global Memory Version

If we implement CF algorithm in global memory version, we do not take advantage of shared memory. To compute value in each thread, we need to fetch two corresponding rectangular data of rating matrix from global memory, which is kind of wasting time. Please check out pseudocode of global memory version below. **Notice that, we only show the pseudocode of similarity matrix computation kernel, since majority calculation comes from this step and the coding idea for steps is similar with first step.**

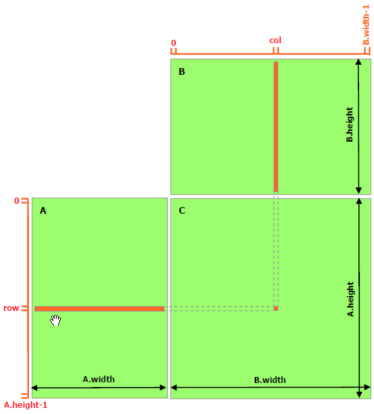

Figure 3: Global memory version

Algorithm 1 Global memory version

```
1: __global__ void MatMulKernel(Rating Matrix, Similarity Matrix, Average
   Value)
2:   If(i > j ) return;
3:   row = by * Dim + ty;
4:   col = bx * Dim + tx;
5:   For(i < n; i++)
6:     compute the dot product of each two rows in rating matrix
7:     compute sum of norms of row i
8:     compute sum of norms of row j
9:   End for loop
10:  compute Pearson_correlation
```

4.2. Parallel Shared Memory Version

In the shared memory version implementation, each thread block is responsible for computing one square sub-matrix C_{sub} of C and each thread within the block is responsible for computing one element of C_{sub}. Each block are sharing same the value inside this block. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory. So every time when we fetch the data, we can directly get it from shared memory. This is much faster than getting data from global memory. By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since matrix is only read (m / TW) times from global memory.[4] I post pseudocode of shared memory version in the following page.

5. Experimental Results with real data

5.1. Data set

We collect real data from Netflix. It contains 665 distinct users and their ratings for over 55000 movies. And the rating score ranges from 0 to 5, where 5 stands for highest score. Also, we generate fake data set to test the

Algorithm 2 Shared memory version

```
1: __global__ void MatMulKernel(Rating Matrix, Similarity Matrix, Average
   Value)
2:   If(i > j ) return;
3:   __shared__ Tile_A[TW][TW];
4:   __shared__ Tile_B[TW][TW];
5:   __shared__ Average_A[TW];
6:   __shared__ Average_B[TW];
7:
8:   For(i < B_DIM; i++)
9:     load Tile_A value;
10:    load Tile_B value;
11:    load Average_A value;
12:    load Average_B value;
13:    __syncthreads();
14:
15:    For(j < TW; j++)
16:      compute the dot product of two rows inside tile
17:      compute sum of norms of row i inside tile
18:      compute sum of norms of row j inside tile
19:      __syncthreads();
20:    End for loop
21:  End for loop
22:
23:  compute Pearson_correlation
24:  store Pearson correlation in global memory
```

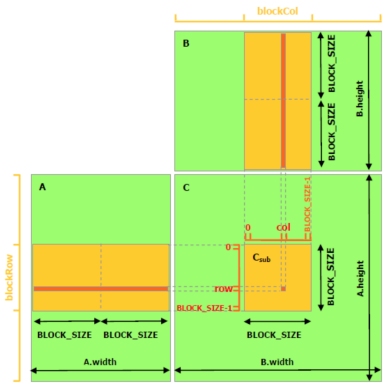

Figure 4: Shared memory version

performance of our algorithm. With different input size, we will check the time performance and analyze them.

5.2. Experimental results

In order to evaluate the scalability, we varied the number of users and the number of movies, respectively. We record corresponding time performance of CPU version, global memory version and shared memory version. And we will also check the speedup between serial version and shared memory version, and global memory version respectively. Table 2 is time performance with different input size of user. And table 3 shows time performance with different movies. **Notice that, the unit time in the table is milliseconds(ms). And also the timing starts from begin of computing similarity matrix and ends at making recommendations**

	m =150	**m = 300**	**m = 600**
serial version(ms)	3998.220	7563.597	31604.614
global memory version(ms)	205.504	226.311	355.513
shared memory version(ms)	179.288	182.093	213.164
speedup(compare to **shared memory version**)	22.29	41.55	148.9
speedup(compare to **global memory version**)	19.5	33.46	89.025

Table 2: Performance and speedup when varying the number of users(number of items n = 4800)

	n =1200	**n = 2400**	**n = 4800**
serial version(ms)	2172.548	3752.794	7749.672
global memory version(ms)	183.212	191.405	219.982
shared memory version(ms)	209.021	193.547	185.028
speedup(compare to **shared memory version**)	10.39	19.44	41.87
speedup(compare to **global memory version**)	11.86	19.64	35.38

Table 3: Performance and speedup when varying the number of movies(number of items m = 300)

5.3. conclusion

From above table, we can see the speed up is not stable. Speedup increases as input size gets larger. It indicates our program gets more and more efficient as parallelism increases. However, on the other hand, it also indicates our GPU is not fully utilized since the input data size is too small.

. Secondly, We can also see that shared memory version is much faster than global memory version in the large data set case. But if we just vary the number of movies, the performance of shared memory version is similar with global memory version. Therefore I believe shared memory will get better if we increase number of users in the program.

6. Experimental Results with fake data

6.1. Data set

In the last section, we realize that GPU is not fully utilized since speedup dramatically increases as the input size increases. Usually, if GPU is fully used, speedup should have a limit level. In this section, we will make some fake data to test the limit of GPU. With no doubt, this fake data should be much larger than what we collect from Netflix. Notice that we only request one nodes, one GPUs, and we set memory 400mb.

6.2. Experimental results

If the data increase to some extent, the computation power of GPU will be fully utilized. From table 4, it shows that speedup between serial version and global memory version becomes stable as increase of number of users. The speedup value varies around 160. As we double number of users each time, the similarity matrix should be four times bigger. Therefore, for global

memory version, it takes four times longer in m = 8000 case than in m = 4000 cases, which indicates GPU approaches its limit. However, shared memory version still outperforms global memory version. And it is obvious to see the speedup compared with shared memory version is incredible huge. For example, in the case of 8000 users, it only takes 6.415 seconds to finish all process while global memory version request 33.225 seconds. And it takes longer than one hour in serial version. **Notice that, the unit time in the table is seconds(s)**

	m =2000	**m = 4000**	**m = 8000**
serial version(s)	320	1298	5262
global memory version(s)	2.059	7.685	33.225
shared memory version(s)	0.400	0.948	6.415
speedup(compare to **global memory version**)	155.415	168.900	158.374

Table 4: Performance and speedup when varying the number of users(number of items n = 4800)

6.3. conclusion

Again, in the fake data set experimental result, we can see that shared memory version outperforms global memory version in large data set. Moreover, there is a limit in speedup. If GPU is fully used, the speedup gradually fluctuates around some number.

7. Future Work

As we can see, GPU has super horsepower in parallel computing field. The results demonstrated that our proposed CUDA-based collaborative filtering algorithm is efficient and scalable. And from our experiment, it shows that the more computation on the GPU, the higher the utilization of the computing horsepower. With the help of GPU, we can speed up our program, and hence some time consuming but more accurate algorithms become available. For example, distinguish similar groups by using the Hashimoto matrix, which is another recommendation algorithms, is very time consuming. But if we deploy Hashimoto matrix algorithm into GPU, customers data can quickly analyzed. Moreover, in practice, data base is much larger than it in experiment. Therefore, the computing horsepower of GPU can be fully utilized.

[1] J. Leverich, C. Kozyrakis, "On the energy (in)efficiency of Hadoop Clusters", SIGOPS Oper. Sys. Rev, 27th edition, 2010.

[2] Z.-D. Zhao, M. Shang, User-based collaborative-filtering recommendation algorithm on hadoop., International Workshop on Knowledge Discovery and Data Mining 0 (2010) 478–481.

[3] Wiki, Graphics processing unit 13 (2013) 123–456.

[4] CUDA, Cuda c programming guide version 4.1, NVIDIA Corp (2012).