Final Project Report

Team Member

Foo Jia Yin (106062361)

Topic

Parallelism and Acceleration in Recommendation Systems
(Parallelized Collaborative filtering with CUDA)

Introduction

In the era of big data, recommendation system is widely utilized in from commercial e-shops to social networks, product review sites other SaaS applications. Many websites collect user profiles to provide some valuable information through personalized recommendation.

Collaborative filtering (CF) is a very common technique used in recommender systems. The key idea of collaborative filtering is to recommend items based on similarity measures between users or items. For instance, the items recommended to a user are those preferred by similar users. The technique is widely used due to its properties that (1) a single algorithm works on any kind of item and (2) it can handle the complexity of users' preference.

Due to the increasing scale of data in these applications is constantly increasing, in some practices, the computational time of collaborative filtering algorithm may be unsatisfying. Different approaches have been proposed to deal with the problem of scalability of recommender systems' algorithms. This project focus on implementation of parallelism approaches with CUDA and compare the computational time with the sequential approach.

Dataset

Source: https://grouplens.org/datasets/movielens/

Dataset	Movie (M)	User (U)	Ratings (N)
tiny (for debug)	5	5	21
100k	943	1682	100000
small	610	9725	100836
1m	6040	3706	1000209

Implementation

The goal is to predict the rating of unrated items for every user. Then, items with high predicted rating can be recommended to each user.

There are two approaches to implement Collaborative Filtering: (1) by user-user similarity and (2) by item-item similarity, both have been implemented in this project. Here we use item-item similarity for demonstration since the algorithms is totally same.

1. **Data structure**
   - M = no of items, U = no of users, N = no of ratings
   - Rating = <user id OR item id, value>

```
struct Rating {
    int id;
    double value;
};
```

   - Rating ratings[N] – sort by item and divided to M items
   - int count[M] (no of ratings for each movie)
   - double magnitudes[M]
   - double similarities[M*M]
   - double predictions[M*N] (output)

2. **Input processing**
   mapUser():
   - Read each (user, item, rating) input
   - Group ratings to (item, (user, ratings)) pairs by insert the ratings into self-sorted multimap (from STL)
   - Hence, the ratings is now sorted by items

```
    multimap<int, Rating> ratingsMap;
    readInput(filename, 0, ratingsMap);
```

   - Time complexity: O(NlogN)

   Parallelization: Not implemented is this step.

3. **Data standardization** + **Calculate rating magnitude** of each item
   magnitude = ||r|| for each movie is calculated by the following equation:

$$||r_x|| = \sqrt{\sum_{i \in N} x_i^2}$$

   calculateMagnitude(): For each item

- Calculate the mean of each item
  (we can treat missing data as 0, by doing this, the missing data will not have effect on items' similarities)
- For each user-item rating,
  - Substract the rating by its mean
  - Add the square of rating to the sum
- magnitude[m] = square root of the sum
- Time complexity: O(N)

Parallelization:

- Since the calculation of each item is independent, it can be parallelized to calculateMagnitude<<<M, 1>>>()
- Each thread performs independent calculation on each item
- Time complexity for each thread: user that rate the item = O(U)

4. Calculate item-item similarity

There are several similarity metrics can be used. In this implementation, cosine similarity is used as the similarity measure.

$$\text{sim}(\boldsymbol{x}, \boldsymbol{y}) = \cos(\boldsymbol{r_x}, \boldsymbol{r_y}) = \frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}$$

calculateSimilarity(): For each item-item pair (m_1,m_2)

- mag_prod = magnitude[m_1] * magnitude[m_2]
- For each user u that has rated m_1 and m_2
  dot_prod += rating[u,m_1] * rating[u,m_2]
- similarities[m_1,m_2] = dot_prod / mag_prod;
- Time complexity: $O(M^2U)$

Parallelization:

- Since the calculation of each item with all another item is independent, it can be parallelized to calculateSimilarities<<<M, 1>>>()
- Each thread performs independent calculation on each item, iterate through all other item
- Why not <<<M*M, 1>>>? Data locality of one item is preserved in each thread
- Time complexity for each thread: O(MU)

5. **Regroup the ratings by users**
   mapUser():
   - Read each (user, item, rating) input
   - Group ratings to (user, (item, ratings)) pairs by insert the ratings into self-sorted multimap (from STL)
   - Hence, the ratings is now sorted by users

   ```
   multimap<int, Rating> ratingsMap;
   readInput(filename, 0, ratingsMap);
   ```

   Time complexity: O(NlogN)

   Parallelization: Not implemented is this step.

6. **Predict rating** for each user-item
   The rating r_{xi} of user x to item i is predicted based on the similarities between the item and other items those have been rated by the user:

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

   predictRatings(): For each user-item pair (u,m_1)
   - For each item that has been rated by the user (u,m_2)
     - sum_rating += similarities[m_1,m_2] * rating[u, m_2]
     - sum_sim += similarities[m_1,m_2]
   - predict rating[u, m_1] = sum_rating / sum_sim
   - Time complexity: $O(UM^2)$

   Parallelization:
   - Since the calculation of each user with all another item is independent, it can be parallelized to predictRatings<<<U, 1>>>()
   - Each thread performs independent calculation on each user, iterate through all other item
   - Why not <<<U*M, 1>>>? Data locality of one user is preserved in each thread
   - Time complexity for each thread: $O(M^2)$

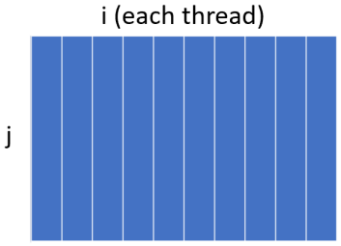
Optimization

1. **Optimizing pairwise operation**
   The time elapsed is mainly determined by calculateSimilarity() function, which is very time-consuming. It consists of pairwise operations of M*M.

Method 1:

```
for (int i = index; i < M; i+=stride )
  for (int j = 0; j < M; j++)
    // job with constant length
```

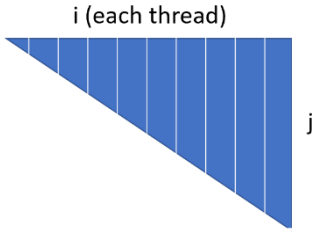
- Problem: Half of the calculation is duplicated



Method 2:

```
for (int i = index; i < M; i+=stride )
  for (int j = i+1; j < M; j++)
    // job with constant length
```

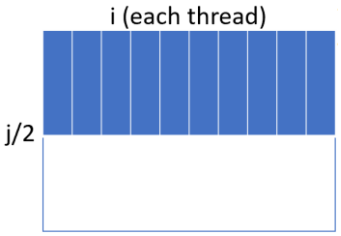
- Problem: total time is determined by the last completed job



Method 3 *(Currently used)*:

```
for (int i = index; i < N; i+=stride )
  for (int j = 0; j < N; j++)
    if ((i + j) % 2 && i < j) || (i + j) % 2 == 0 && i > j))
      // job with constant length
```

- Workload of each thread is now j/2



2. Optimizing data structure

Method 1:

- Data structure: use separate array for count, user id, rating, magnitudes, similarities

```
int movieList[M]; int userCount[M]; int userId[N];
double u_ratings[N]; magnitudes[M];
```

Result: time elapsed (ms)

dataset	algorithm	**Sequential**	**Parallel 1 <32,32>**
4x5	item	0.008	0.24
	user	0.008	0.34
610x9125	item	2605.565	7343.519 / 2323.9
	user	116.78	769.604 / 421.8
943x1682	item	728.141	7343.519 / 989.608
	user	567.318	4557.879 / 3373.767
6040x3264	item	4421.31	47038.702 / 23848.671
	user	8090.46	1365.302 / 711.995

Red: before optimizing pairwise operation, Blue: after optimizing pairwise operation

- Experimented with other combinations <<< blockPerGrid * threadPerBlock>>>, but still slower than sequential code

Method 2:

- Try to group the data for each item and user into profiles

```
struct Rating {
    int id;
    double value;
};

struct Profile1 {
    int count;
    double magnitude;
    Rating* ratings;
};

struct Profile2 {
    int count;
    Rating* ratings;
};
```

- Dynamically allocate memory for each item / user during mapMovie() / mapUser()

  For each item:

```
int cnt = m_u_ratings.count(it->first);
cudaMallocManaged(&mData[m], sizeof(T) + sizeof(Rating) * cnt);
mData[m]->count = cnt;
mData[m]->ratings = (Rating*) mData[m] + sizeof(mData);
movieList[m] = it->first;
```

Result: time elapsed (ms)

dataset	mode	**Sequential**	**Parallel 2 <M,1>**
4x5	item	0.01	0.876
	user	0.012	0.88
610x9125	item	2510.706	523.93
	user	Segmentation fault	285.996
943x1682	item	467.644	150.769
	user	308.696	155.981
6040x3264	item	9283.719	Segmentation fault
	user	17495.76	Segmentation fault

- Generally faster than the sequential code
- Problem 1: Presence of pointer Ratings* makes things complicated and hard to debug, I don't recommend anyone doing like this
- Problem 2: Undergoes segmentation fault due to failure of cudaMalloc() / cudaMallocManaged(), even there are enough memory in GPU. I can't fix that.

Method 3 (*Currently used*):

- Back to method 1, but group id and rating value together as Rating
  Item id, user id, rating → item id, rating(movie id, value)

```
struct Rating {
    int id;
    double value;
};
```

- No more passing pointer, yay!
- If a value in array is repeatedly used, use a register in kernel to store it
  For example,
  sim = similarities[m*M + ratings[n].id]

Result: time elapsed (ms)

dataset	mode	**Sequential**	**Parallel 2 <M,1>**	**Parallel 3 <M,1>**
4x5	item	0.01	0.876	0.619
	user	0.012	0.88	0.61
610x9125	item	2510.706	523.93	384.981
	user	Segmentation fault	285.996	101.199
943x1682	item	467.644	150.769	86.457
	user	308.696	155.981	75.279
6040x3264	item	9283.719	Segmentation fault	2675.681
	user	17495.76	Segmentation fault	4003.035

- Obvious improvement in speed
- No more segmentation fault

3. Experiment on other <<<blockPerGrid * threadPerBlock>>> combination

Conclusion:

- <<<M, 1>>> for item-based and <<<U, 1>>> for user-based achieve the best result
- Only in some cases, <<<M/8, 8>>> or <<<U/8, 8>>> perform better

dataset	mode	**Parallel 2 <M,1>**	**Parallel 2 <M/8,8>**
943x1682	item	384.981	362.527
6040x3264	user	4003.035	3435.059

- More threadPerBlock may result in segmentation fault

Instructions

1. **Input files**
   - There are 4 datasets placed under input/
     - input/ratings_tiny.dat
     - input/ratings_100k.dat
     - input/ratings_small.dat
     - input/ratings_1m.dat

   - Input format
     Movielens dataset is modified to meet the following input format:
     - First line: *U M N* (specifying no. of users, no. of movies, no. of ratings)
     - Following by N lines of: *user_id movie_id rating*
     - *U, M, N* are integers. *user_id, movie_id* are integers start from 1, where as *ratings* is between 1.0~5.0.

2. **Compilation and Execution**
   - Evaluating performance
     (This will print time elapsed in each stage and GPU memory used)
     ```
     make eval && srun -p ipc21 --gres=gpu:1 -N 1 ./eval input/[dataset] [mode] [threadNo]
     ```
     For example,
     ```
     make eval && srun -p ipc21 --gres=gpu:1 -N 1 ./eval input/ratings_1m.dat user 8
     ```

   - Print predict ratings
     (Output file will contain U*M-N lines of *user_id movie_id rating*)
     ```
     make predict && srun -p ipc21 --gres=gpu:1 -N 1 ./predict input/[dataset] [mode] [threadNo]  > [Outputfile]
     ```
     For example,
     ```
     make predict && srun -p ipc21 --gres=gpu:1 -N 1 ./predict input/ratings_1m.dat user 8 > Output.txt
     ```

   - Evaluating performance of method 2
     ```
     make v2 && srun -p ipc21 --gres=gpu:1 -N 1 ./v2 input/[dataset] [mode]
     ```

   - Evaluating performance of sequential code
     ```
     make seq && ./seq input/[dataset] [mode]
     ```

Appendix: Time elapsed in each stage

Sequential

Dataset	algorithm	Input Processing	Calculate Magnitude	Calculate Similarites	Predict Ratings	Total
4x5	item	0.129	0.003	0.004	0.003	0.01
	user	0.075	0.003	0.006	0.003	0.012
943x1682	item	90.959	0.269	465.615	1.76	467.644
	user	82.573	0.263	308.433	0.016	308.696
610x9125	item	83.326	0.428	2507.027	3.251	2510.706
	user	87.705	0.252	147.057	Seg. fault	
6040x3264	item	931.874	2.649	9255.989	25.081	9283.719
	user	782.601	2.7322	17493.03	0.005	17495.76

Parallel 2 <M,1>

Dataset	algorithm	Input Processing	Calculate Magnitude	Calculate Similarites	Predict Ratings	Total
4x5	item	0.298	0.34	0.028	0.508	0.876
	user	0.214	0.299	0.036	0.545	0.88
943x1682	item	57.347	1.538	99.15	50.081	150.769
	user	48.434	1.522	90.162	64.297	155.981
610x9125	item	113.392	3.054	462.547	58.329	523.93
	user	48.088	3.343	113.224	169.429	285.996
6040x3264	item	Segmentation fault				
	user	Segmentation fault				

Parallel 3 <M,1>

Dataset	mode	Input Processing	Calculate Magnitude	Calculate Similarites	Predict Ratings	Total
4x5	item	72.93	0.279	0.027	0.313	0.619
	user	82.56	0.27	0.037	0.303	0.61
943x1682	item	116.509	0.917	83.855	1.685	86.457
	user	128.393	0.853	73.501	0.925	75.279
610x9125	item	126.718	1.076	379.831	4.074	384.981
	user	140.972	1.021	99.201	0.977	101.199
6040x3264	item	762.175	5.902	2655.184	14.595	2675.681
	user	545.094	6.094	3990.43	6.511	4003.035