MATRIX FACTORIZATION FOR RECOMMENDER SYSTEM: SVD

BASIC MATRIX FACTORIZATION MODEL :

Matrix factorization models map both users and items to a joint latent factor space of dimensionality f, such that user-item interactions are modeled as inner products in that space. Accordingly, each item i is associated with a vector qi ∈ f , and each user u is associated with a vector pu ∈ f. For a given item i, the elements of qi measure the extent to which the item possesses those factors, positive or negative. For a given user u, the elements of pu measure the extent of interest the user has in items that are high on the corresponding factors, again, positive or negative. The resulting dot product, qi T pu, captures the interaction between user u and item i—the user's overall interest in the item's characteristics. This approximates user u's rating of item i, which is denoted by rui, leading to the estimate

r̂ ui = qi T pu. (1)

The major challenge is computing the mapping of each item and user to factor vectors qi , pu ∈ f . After the recommender system completes this mapping, it can easily estimate the rating a user will give to any item by using Equation 1.

Applying SVD in the collaborative filtering domain requires factoring the user-item rating matrix. This often raises difficulties due to the high portion of missing values caused by sparseness in the user-item ratings matrix. Recent works suggested modeling directly the observed ratings only, while avoiding overfitting through a regularized model. To learn the factor vectors (pu and qi), the system minimizes the regularized squared error on the set of known ratings:

min q p * * , (, u i)∈ ∑κ (rui - qi T pu) 2 + λ(|| qi ||2 + || pu ||2) (2)

Here, κ is the set of the (u,i) pairs for which rui is known (the training set).

SGD TO MINIMIZE EQUATION 2 :

The Algorithm loops through all ratings in the training set. For each given training case, the system predicts rui and computes the associated prediction error

eui = def rui - qi T pu.

Then it modifies the parameters by a magnitude proportional to g in the opposite direction of the gradient, yielding:

- q q i i ← + g λ · · ( ) e p ui u i - ·q

• p p u u ← + g λ · · ( ) e q ui i u - · p

This approach combines implementation ease with a relatively fast running time.

For parallel SGD the looping through all points in the training set parallelly.

IMPLEMENTATION - CUDA

Code Structure : header files - svd.cuh , svd.h ; cuda kernels and main parallel functions svd.cu ; cpu-gpu implementation calling functions from above codes - svd.cpp

Functions :-

svd.cu -

void cudaFindRMSKernel - global : data segmentation of R & R_1 for calculating RMS loss

__global__ void cudaMultiplyKernel : data segmentation of P & Q for getting R_1

_global__ void cudaTrainingKernel : parallel SGD as dividing between threads columns of Q.

float cudaCallFindRMSKernel : calls the function and returns RMS

void cudaCallMultiplyKernel :calls function

void cudaCallTrainingKernel : calls function

svd.cuh - headers of functions in svd.cu to precede the function definition in svd.cu (due to C rules)

svd.h -

void gaussianFill - : Initialize user and item matrices for SGD

void readData : read data from training set

void decompose_CPU : function header ; function in svd.cpp :

void decompose_GPU : function header ; function will be in svd.cpp :
svd.cpp

void decompose_GPU : GPU algorithm
void decompose_CPU : CPU algorithm

DATASET

Data set used is a part of the following dataset with attributes :

Attribute Information

Movie ID : Arbitrarily assigned unique integer in the range [1 .. 1682].
Customer ID : Arbitrarily assigned unique integer in the range [1..943] (with gaps i.e., not continuous).
Rating : Number of 'stars' assigned to a movie by a customer; an integer from 1 to 5.
ID : Represents an instance . It is a random 10 digit number

Order :

Customer ID	MovieID	Rating	ID

U.info has details of the complete dataset
U1.base contains the part of dataset we use

RUNNING THE CODE

```
nvcc -c svd.cu
```

Run the above code to get svd.o

```
make
```

Next, run the above make command to get the executable

cuda_SVD

Above is the executable

Execution -

Using default values of number of users,items and dimensions :

./cuda_SVD ./u1.base

Giving user inputs :

```
./cuda_SVD filename <num_users> <num_items> <num_of_dimensions>
```

```
num_users = 943
num_items = 1682
Remain constant f = num_of_demensions can be arbitrary
```

RESULTS :

The estimated/learnt complete user-item preference matrices are written to csv files for both cpu and gpu

Using this matrices a future user of the system can be recommended items of interest to him by finding higher value (rating) items according to the output Matrix

Eyeballing the matrices for the cpu and gpu shows the values are quite close upto 2 decimal points

The RMS of CPU is lower [1.28598] and that of GPU is higher [1.3241]

Runtime of CPU : 9.473005 s
Runtime of GPU : 1.658478 s

Parallel code runs faster for default values.
For arbitrary f - as f increases CPU time increase is greater than GPU time and GPU is much lower that CPU - error decreases with increasing f

Runtime CPU	Runtime GPU	Error CPU	Error GPU	f
5.759914	1.160033	1.27	1.335	20
5.183168	1.166956	1.29	1.36	10

6.4511	1.1618	1.29	1.32	30
11.469542	1.210144	0.985	1.1259	100
7.907414	1.232502	1.08	1.2667	50

Algorithm - Pseudocode of the code :

We use parallelism in three places - 1) Multiplying P & Q while training on the GPU
2) In the updating of P & Q
3) Calculating the RMS error by subtracting the product of P & Q from the original R (rating matrix) on GPU

Three Kernels are used for the above three functions

CPU :

1. Randomly initialize all vectors p_u and q_i, each of size 10.
2. for a given number of times (i.e. number of epochs), repeat:
   - for all known ratings r_{ui}, repeat:
     - compute $\frac{\partial f_{ui}}{\partial p_u}$ and $\frac{\partial f_{ui}}{\partial q_i}$ (we just did)
     - update p_u and q_i with the following rule: $p_u \leftarrow p_u + \alpha \cdot q_i(r_{ui} - p_u \cdot q_i)$, and $q_i \leftarrow q_i + \alpha \cdot p_u(r_{ui} - p_u \cdot q_i)$. We avoided the multiplicative constant 2 and merged it into the learning rate α.

GPU : same algorithm done in parallel by incorporating following changes -
Initialize P,Q&R as MatrixXf - variable size matrix initialization in Eigen
Alot dynamic memory to host_P,host_Q & host_R using malloc on CPU
Randomly (guassianFill) host_P,host_Q & host_R
Read ratings from the file line by line into data vector
In each iteration take three indices as rating.
rating[0] = one index ; rating[1] = second index ; rating[2] = value of the rating

Initialize number of blocks = 64
And number of threads = 64
Each block has that number of threads

Allocate linear memory to dev_P , dev_Q , dev_R1, dev_R0 matrices on each device using cudaMalloc on GPU

Copy host values of P,Q,R to each device using cudaMemcpy
Initialize buffer on the host to store data from file,cudaMalloc dev_data to store this on GPU.
Used gpuErrChk function to catch errors while copying .
Use a while loop with condition on the RMS < a tolerance value
2)Call the training kernel to find dev_P and dev_Q, this is the update step of the stochastic gradient descent - initially ,on the first run,when we have not calculated and taken its derivative we initialize the derivative to zero and pass it to the function. Each block receives its share of original P&Q withe the eta and derivative to update - Using atomicAdd for the additions

1) Call the multiply kernel to multiply the obtained P & Q from training -again P & Q distributed to blocks and threads are run on each blocks.Product stored in dev_R1

3)Call the RMS kernel - performs R1-R0 in parallel .Stores the difference in dev_sum. atomicAdd used to fill dev_sum.

Calculate derivative using these matrices - dev_sum , R1 .

Copy the dev RMS,P& Q value to hosts and write to csv file
Free all host and device memory allocations

Driver Function :

Take num_users,num_items & num_f from from command line
Initialize gamma = 0.001 , lambda to 0.0005
Read from file
Call cpu and gpu functions .

Default num_users = 943
num_items = 1682
num_f = 30

num_users = 943; num_items = 1682; num_f = 30;

const float gamma = 0.01; const float lamda = 0.005

REFERENCES :

http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-gower-netflix-SVD.pdf