1 Problem Statement

Recommendation systems are an integral part of every social and consumer portal these days. They involve processing user and product interactions data to find out meaningful patterns. Based on these patterns, they stimulate and suggest future such interactions in the interest of the consumer and business. Processing these interactions contain a large amount of computation due to the very high number of users and products. Thus, the scale of the problem provides a good scope and importance for parallel programming these computations. There are various means to model these computations.

Specifically, we are interested in a method of recommendation called as item based collaborative filtering using latent models. This method involves two phases:

a. an initial modeling phase to represent the users and items in a lower dimensional plane. The interaction data is used to construct this model. The lower dimensions are called as the latent factors.

b. a recommendation phase, the method to exploit the model for predicting the probable future interactions and suggest them to the user.

2 Parallel Programming model

The parallel programming model we have chosen is using GPUs. Reasons for the selection include
1) Modeling phase involves sparse matrix factorization of interaction data. Harnessing the computation power of GPUs for this factorization would be ideal.
2) Inherently, the technique Alternating Least Squares which we have selected to reduce the prediction loss function embraces parallelism unlike other techniques like Stochastic Gradient Descent.
3) Parallel computation of items' similarity in recommendation phase is possible.
4) For every single user, the recommendation phase can be carried out independently.

3 Naming Conventions

n - Number of users.
m - Number of items.
r - Total number of ratings given.
l - Number of latent factors.
n_{u_i} - Number of items user i has rated.
n_{v_j} - Number of users who have rated item j.
Matrix R - Utility/Interaction matrix where rows are the users and columns are the items. Entries represent ratings. Non-rated entries are marked as 0.

Dimension is n*m.
Vector u_i - A lower dimensional vector representing user i of dimension l.
Vector v_j - A lower dimensional vector representing item j of dimension l.
Matrix U - A lower dimensional matrix representing user vectors stacked back-to back of dimension l*n.
Matrix V - A lower dimensional matrix representing item vectors stacked back-to-back of dimension l*m.
Matrix V_{I_i} - Submatrix of V containing latent vectors of items that user i has rated.
Matrix U_{I_j} - Submatrix of U containing latent vectors of users who rated item j.
Matrix W - A N*M matrix representing whether user i has rated item j where $e_{ij} = 1$ if rated $e_{ij} = 0$ otherwise.
Vector $R(i, I_i)$ - Ratings given/received by entity(user/item) i.
Lambda λ - Regularization parameter
Matrix E_{ab} - Identity matrix of dimension a*b
Item Similarity matrix S - A symmetric matrix of dimension m*m where entry s_{ij} represent how similar items i and j are
.* - dot product.

The design and implementation of modeling phase are based on references 2 and 3.

4 Data Representation

The number of possible interactions $n * m >> r$. Hence, we need to optimally represent the utility matrix. Based on this sparsity and our computation we need the following two representations:

4.1 Compressed Sparse Row (CSR)

A row major format composing of following arrays:

csr_a - Holds all non-zero entries of Q in row major order. Size = number of ratings(R)

$csr\_ia[i]$ - Holds the number of non-zero items till ith row. Size = number of users(N) + 1

csr_ja - Holds the column number of each entry in CSR_A. Size = number of ratings

4.2 Compressed Sparse Column (CSC)

A column major format composing of following arrays:

csc_a - Holds all non-zero entries of Q in column major order. Size = number of ratings(R)

csc_ia[*i*] - Holds the number of non-zero items till ith column. Size = number of items(M) + 1

csc_ja - Holds the row number of each entry in CSC_A. Size = number of ratings

Total memory needed to represent the data = 2r + (n + 1) + 2r + (m + 1)
Since $r << (n * m)$ this representation is more optimized than a single (n*m) 'R' representation.
For example,
R =

$$\begin{bmatrix} 0 & 0 & 4 \\ 5 & 1 & 0 \\ 0 & 3 & 2 \end{bmatrix}$$

then

$$csr\_a = [4, 5, 1, 3, 2]$$

$$csr\_ia = [0, 1, 3, 5]$$

$$csr\_ja = [2, 0, 1, 1, 2]$$

$$csc\_a = [4, 5, 1, 3, 2]$$

$$csc\_ia = [0, 1, 3, 5]$$

$$csc\_ja = [0, 1, 1, 2, 2]$$

5 Modeling phase - Matrix factorization

The goal of this step is to factorize the high dimensional utility matrix R into lower dimensional matrices U and V using squared loss function. Only the rated items will be able to aid during the factorization. Predicted ratings are given by $(U^T * V)$. Following is the loss function L

$$L = \sum (W. * (R - U^T V)^2)$$

To avoid overfitting, we use a regularized version of the loss function

$$L_r = \sum (W. * (R - U^T V)^2 + \lambda (\sum_i n_{u_i} |u_i|^2 + \sum_j n_{v_j} |v_j|^2))$$

Using Alternating Least Squares, we find U keeping V fixed and vice-versa
Following is the overall algorithm,

While ($Avg\_LSE\_Error <= Threshold$) repeat

1. Calculate $(V_{I_i} V_{I_i}^T + \lambda n_{u_i} E)$
2. Solve $u_i = (V_{I_i} V_{I_i}^T + \lambda n_{u_i} E)^{-1} V_{I_i} R^T(i, I_i)$
3. Calculate $(U_{I_j} U_{I_j}^T + \lambda n_{v_j} E)$
4. Solve $v_j = (U_{I_j} U_{I_j}^T + \lambda n_{v_j} E)^{-1} U_{I_j} R^T(i, I_j)$
5. Calculate Average Least Squared Error

Steps 1 and 3 are solved by a kernel called as Entity-matrix multiplication kernel.
Steps 2 and 4 are solved by a kernel called as LDL decomposition kernel.
Step 5 is done in a three level hierarchical kernel dividing the workload - Reduction kernel.

6 Recommendation Phase

Following is the overall algorithm,
1) Compute the similarity matrix S using a distance kernel like Euclidean or dot product.
2) Presort the similar items for all the items.
3) For each user u_i

a. Group rated items as I_{ir}
b. Group unrated items as I_{iu}
c. For each unrated item u in I_{iu}
d. Find 'k' similar items in group I_{ir} and compute their average rating R_{iu}
e. Pick the top 'k' unrated items for the user using R_i

'k' in steps d and e need not be same but we made it so for simplicity.

6.1 Distances

After each item in the database is described in terms of latent factors, the next step is to calculate the distances between every possible pair of items. Later stages of the recommendation system will have to constantly look up pair distances, so precalculating and storing pair distances will cut down on running time.

We choose to use Euclidean distance. The distance between items i and j, assuming we have n latent factor dimensions, is $d(i,j) = \sqrt{\sum_{k=1}^{n} (c_{i,k} - c_{j,k})^2}$. Because Euclidean distance is symmetric, we actually only need to calculate half the total possible distances. Given m items, this works out to be $\frac{m(m-1)}{2}$ distance calculations. As the number of items in our database grows, the number

of distance calculations increases roughly by $O(m^2)$.

Normally a collection of distances would be represented as a $m \times m$ matrix, but half the matrix is wasted if the chosen distance is symmetric, as is our case. Instead, we choose to flatten the matrix into a 1-D array where distances are laid out as $(i \to i+1), \ldots, (i \to j), \ldots, (i \to m)$ where $i < j, i < m$ for $i = 1, \ldots, m-1$.

Workload for GPUs is divided as follows: every item is assigned one block (m total items, and thus m total blocks), and each block uses its available threads to calculate the distance between its assigned item i and every item j where $i < j$. Here, there is some inefficiency in balancing work loads. Blocks that are assigned items that appear earlier have to do more work than blocks that are assigned items that appear later. For example, a block assigned $i = 0$ will have to calculate distances to $j = 1, \ldots, m$ where a block assigned item $i = m-1$ only has to calculate one distance to $j = m$. We can possibly improve inefficiency in future work.

As we increase the number of threads per block, we see a performance improvement:

# threads	Time (ms)
50	11959
100	6637
200	4614
300	3943
400	3825
500	3817
Serial Execution	6687

The above numbers are run on a database of roughly 10K items; using more items reduces the impact of GPU overhead and widens the gap between serial and parallel execution.

6.2 Sorting

Presorting distances is another way to save work in the future. For every item i, we want to sort the distances (in ascending order) to all $j \neq i$. In this load balance scheme, each block handles one item and performs a sort on that item's distances to all other items. The sorting algorithm is an iterative quicksort, and for m items, the total average work to be done is $O(m^2 \log m)$.

As before, when we add more threads to each block, performance improves:

# threads	Time (ms)
25	6262
50	4282
75	3205
100	3166
125	2154
150	2000

The above numbers are generated for a database of roughly 1K items. Our current implementation uses dynamic memory allocation in each thread, and

the amount of space required scales with the number of items in the database. This oversight limits the number of threads we can use due to limited space on the GPU device. Future work should focus on removing this memory bottleneck so it can scale in performance just like the distance calculation.

6.3 Recommendation

With distances calculated and presorted, the recommendation phase begins. Given all users' ratings of items and sorted item distances, the system must return a list of the top k items that match each user's taste; the k items must be drawn from the pool of items each user has *not* rated.

As in the previous load balancing schemes, each block handles the computation for one user. Each block splits the items into two categories, *rated* and *unrated*. For every *unrated* item, a block searches for the k most similar *rated* items and calculates the average rating of those k items. The unrated item is then assigned this average rating.

Once all unrated items have been given a calculated rating, the block does one final search to find the top k unrated items and returns them as recommended items for the given user.

The value of k is recycled between the search for rated items and in the number of recommended items returned. This is done for convenience and does not necessarily have to be this way - the values can differ if desired. Also, performance of this section is bottlenecked by the search for the k closest *rated* items. We tried to use NVIDIA's Thrust library to perform efficient set-membership operations (determining if item $i \in RI$ (rated items)), but this proved much slower than our naive implementation. This is most likely because we are not using it properly, and fixing this is probably the easiest improvement we can make to running time.

Just as in the presorting subphase, the recommendation subphase uses dynamic memory allocation and is limited in the number of threads that can be run in each block. Similarly, future work should focus on reimplementing this function to remove dynamic memory allocation and allow performance to freely scale. The total runtime for roughly 1K users and 1K items using 25 threads is 2.0312×10^6 milliseconds; for 50 threads, the runtime is 1.4824×10^6 milliseconds.

7 Code Description

All the kernels are defined in the file device.cu
Following are the details about the kernels and their workload:

KernelFunctionName	Threads	Workload
EntitiesMultiply	(n or m)	((n_{u_i}) or (n_{v_j})) and l*(l-1)
LDLSolve	(n or m)	(l*l)
PredictRatings or Reduction0	r	1
MSE	(n or m)	(n or m)
CalcDistance	m	(m-i) for thread i
ProcessUser	n	m-n_{u_i}

Folders

parallel_code - GPU implementation of the recommender system

serial_code - Sequential version with the same memory layout as the parallel version

logs - Sample logs used in reporting

data - Dataset files

Each folder contains a readme.txt to help in providing information
To implement the serial version, we simulate the threadids sequentially and reuse the kernel code

user.h - a user class to hold information about users
items.h - an item class to hold information about items
umatrix.cc and .h - Data representation files
model.cc and .h - Top level user file holding objects of users, items and umatrix
device.h - a header file to expose device functions
device.cu - original file containing all the kernel functions and the main loop for both the phases
readme.txt - containing information of how to run the code.

8 Target Platform

HPCC login2, 1 node, 1 GPU per node.
GPU : Tesla K20m.
Total global memory: 7 GB.
Total shared memory per block: 49 KB.
Maximum threads per block: 1024.
datasets:
dataset1: m = 943, n = 1682, r = 100000
dataset2: m = 6040, n = 3952, r = 1000000

9 Results and Analysis

Experiements are run with $\lambda = 0.001$ and l=100
As shown in Fig 1, the entity multiplication kernel takes the maximum time

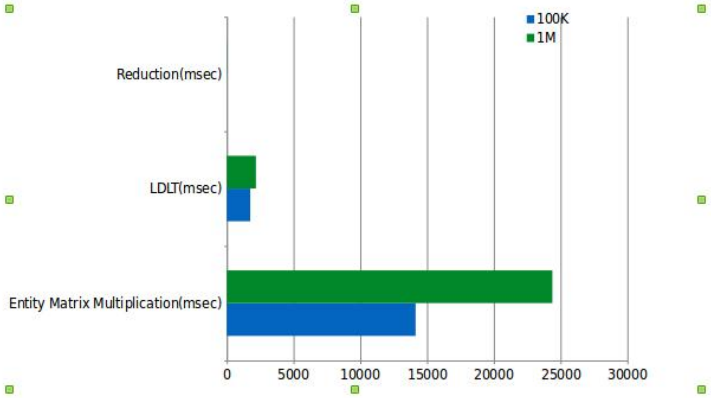

Figure 1: Modeling Phase Kernel time bar

in both the 100K and 1M datasets followed by LDL decomposition kernel. The reduction kernels take the minimum time among all. Hence, there are room for improvements along entity multiplication kernel and along L (as time increases between 100k and 1M).

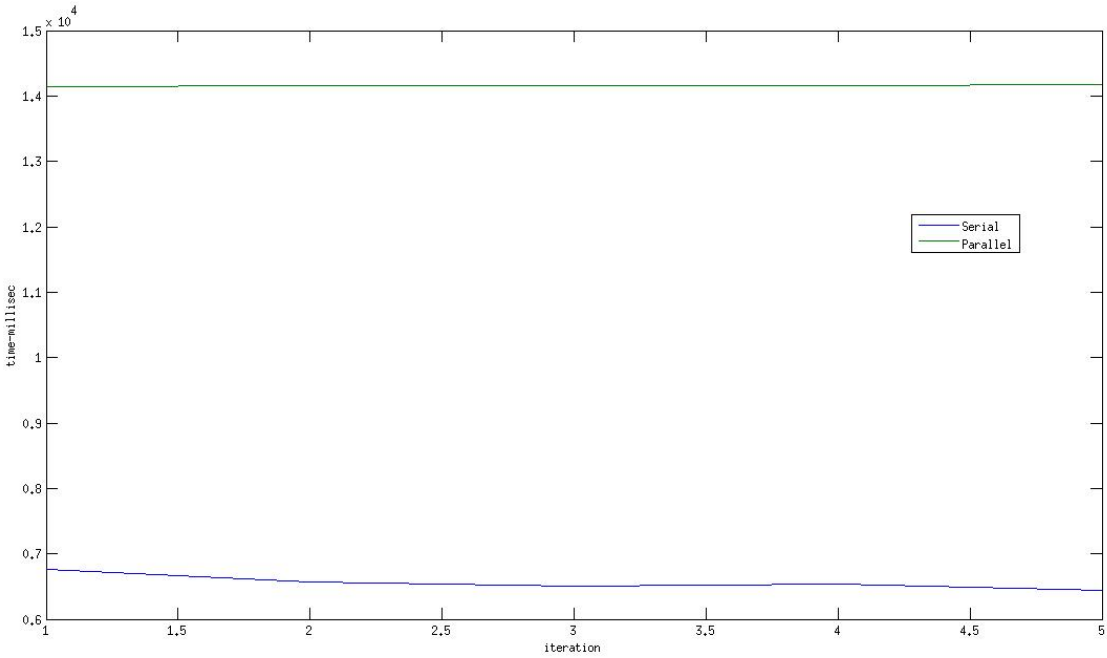

Figure 2: 100K-MaxTimeKernel-SerialVsParallel

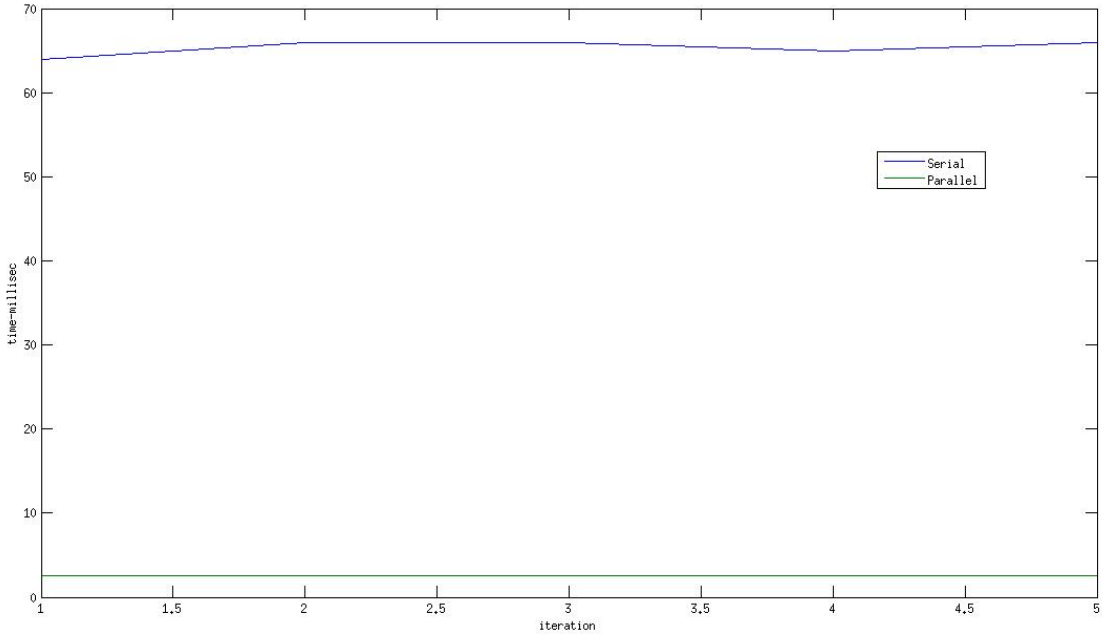

Figure 3: 100K-ReductionKernel-SerialVsParallel

Fig 2 and 3 shows the 100K dataset kernel that takes the maximum time (entity multiplication kernel) and minimum time (reduction kernel). As we compare both the tasks for the serial and parallel version, in the entity multiplication kernel, the serial version is faster than parallel version as the overhead involved in parallelization outweighs the benefit. But in reduction kernel since we use the number of threads as the number of ratings (which is large compared to n or m), the overhead is worth the parallelization.

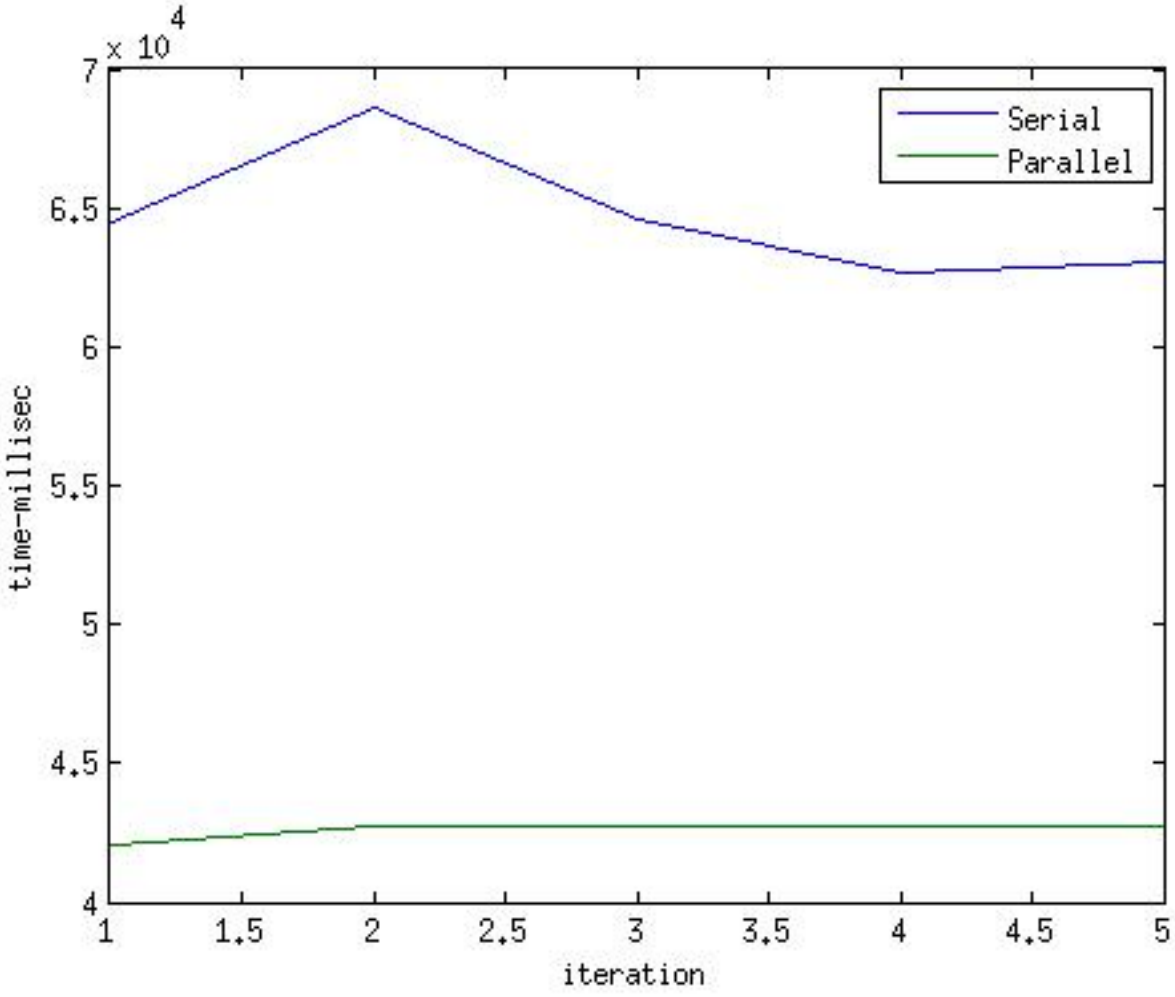

Figure 4: 1M-MaxTimeKernel-SerialVsParallel

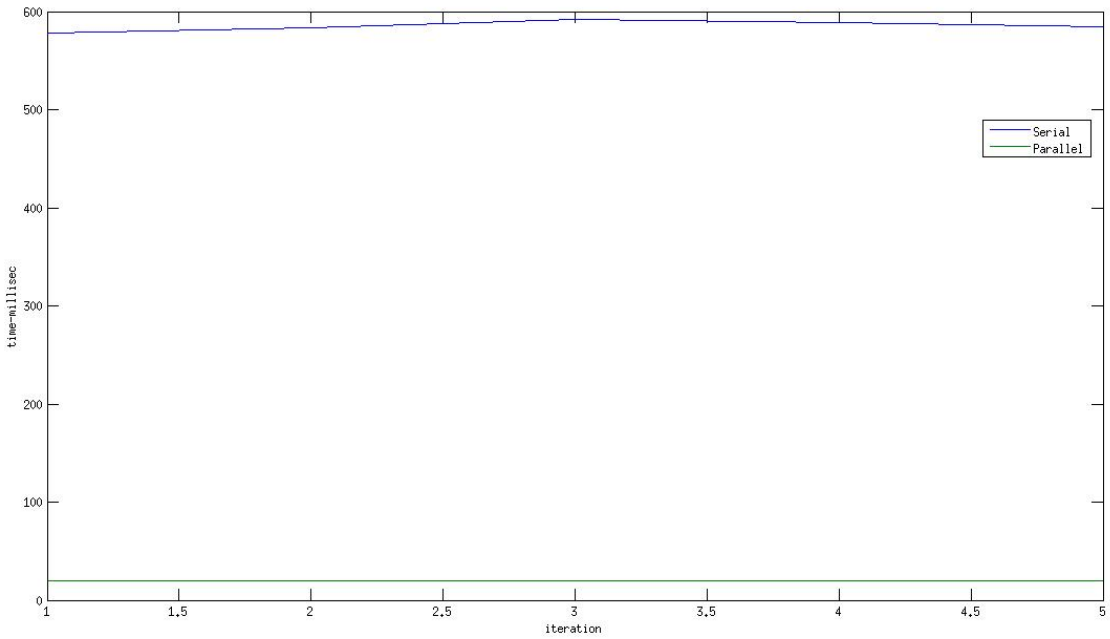

Figure 5: 1M-ReductionKernel-SerialVsParallel

Fig 4 and 5 are proof that as we increase the number of users and items sufficiently as in 1M, even the maximum time kernel like entity multiplication kernel will benefit from the parallelization. The parallel GPU version beats the serial version in both the maximum time taking entity-multiplication kernel and minimum time taking reduction kernel. Hence, our implementation works better as number of items and users get increased which will be the actual scenario.

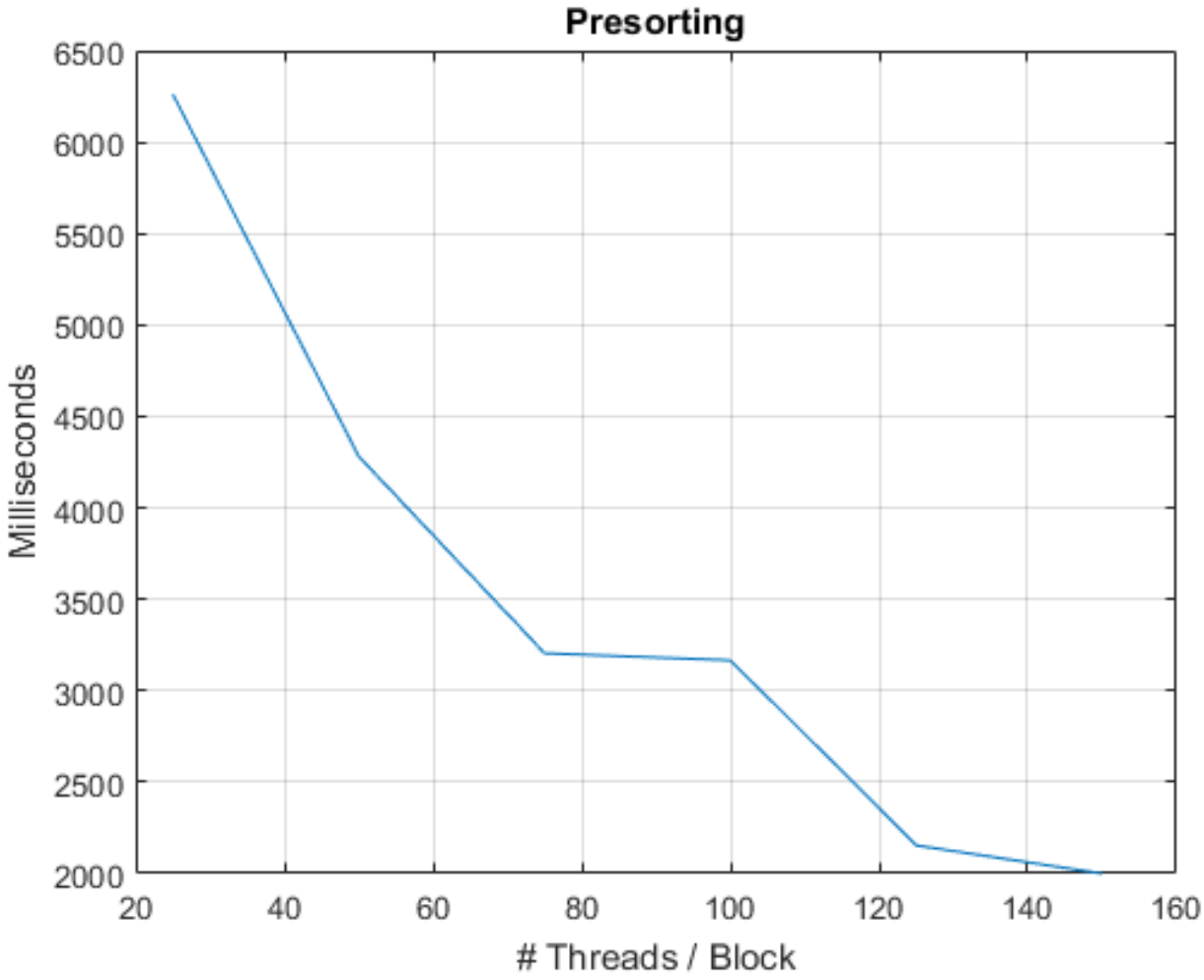

Figure 6: Presorting

Presorting in recommendation phase as well benefits from increasing the threads as the computation time decreases.

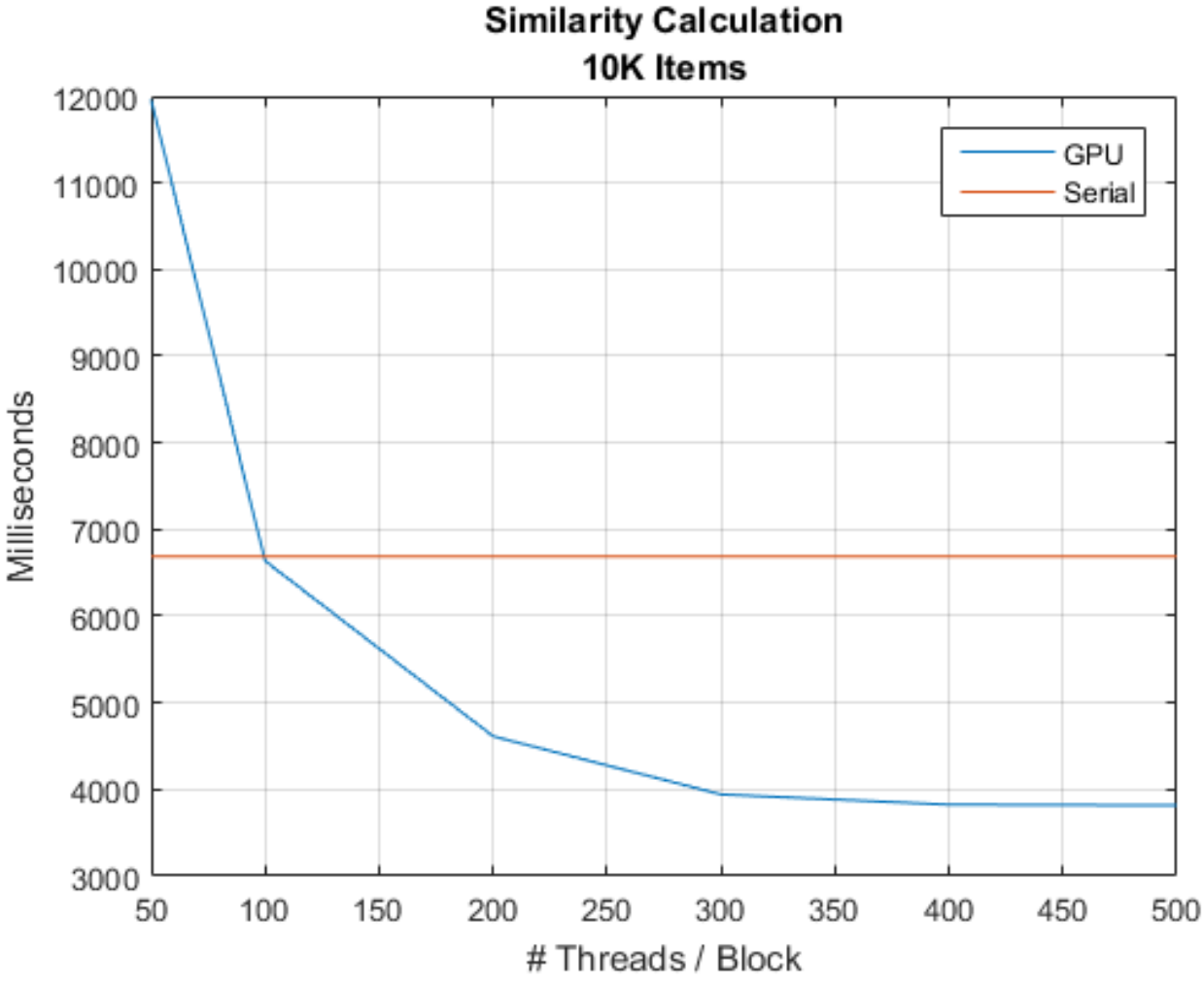

Figure 7: Similarity Calculation

Item similarity calculation in recommendation phase benefits from increasing the threads as the computation time decreases. as shown in Figure 7. It beats the serial version for more than 100 threads per block.

10 Lessons learned

a. Data representation is a very significant step in GPU programming, affects the way parallel algorithms are designed.
b. Child kernels are possible methodologies to hierarchially parallelize algorithms:

We tried child kernels at the Entity-matrix multiplication step of modeling phase spawing threads for each rated item in the user thread (and vice-versa for items). As this involves writing to a shared memory synchronously at each computation, this became a bottleneck for the algorithm and hence we removed it. Our belief is better designs using child kernels can be used to parallelize this step further.
c. Using better parallel GPU implementations of sorting algorithms like bitonic sorting should be looked into.

11 References

1.Item based collaborative filtering
2.NVIDIA Recommendation System
3.Parallelization of the Alternating-Least-Squares Algorithm With Weighted Regularization for Efficient GPU Execution in Recommender Systems
4.Alternating Least Squares for collaborative filtering