# CS 179 Project Proposal

Ian Kuehne

## Summary

Collaborative filtering is a machine learning problem in which the preferences of users over a set of items is learned and used to predict unknown preferences. A classic example is the ratings Netflix shows to users: each user sees a rating based on what Netflix predicts that user will rate that movie. One of the primary methods applied to collaborative filtering is the computation of low-rank approximate factorizations of extremely large and sparse matrices, and the main algorithm used to compute such factorizations (based on stochastic gradient descent) appears to lend itself to GPU acceleration.

Note that this project is closely related to CS 156b, a class I am currently taking; I have cleared it with Yaser Abu-Mostafa (the 156b instructor) and I am awaiting permission from Al Barr.

## Background and Explanation

From 2006 to 2009, Netflix held a competition for the best recommender system: that is, given Netflix's recorded users, movies, and ratings, the goal was to predict ratings on new users and movies. The input thus consists of a list of some hundred million (user, movie, date, rating) tuples, and the goal is, given a new list of a few million (user, movie, date) tuples predict the corresponding rating for each.

The main solution to this problem is based on the idea that the input can be seen as a sparse matrix $M$, where each user $u$ has a row and each movie $m$ a column, and $M_{u,m}$ is $u$'s rating of $m$. If we let $U$ be the number of users, $M$ the number of movies, and $F$ be some number of "features", then we can "learn" $M$ by computing a $UxF$ matrix $A$ and an $FxM$ matrix $B$ such that $AB$ minimizes

$$\sum_{(u,m)\in D} (M_{u,m} - (AB)_{u,m})^2$$

where $D$ is the input data; i.e. so that $AB$ is a good approximation to the entries we have of $M$. Then, given a new $(u, m)$, we predict a rating of $(AB)_{u,m}$.

There exists a very simple and fairly efficient algorithm to compute such an approximate factorization based on gradient descent along the above error. Essentially, our gradient for feature $i$, user $u$ on $A_{u,i}$ is just $\varepsilon B_{i,m}$ where $\varepsilon$ is our error on $(u, m)$; on $B_{i,m}$ it is just $\varepsilon A_{u,i}$. There are various tweaks (for example, regularization) which in practice improve the performance of that gradient, but that is the idea.

The algorithm can likely be accelerated on the GPU by dividing up the work over the input data: just assign each thread some subset of the input points, and have them compute the gradient on each of those points.

**Previous Work and Challenges**

The main challenge with that approach is that any two inputs with the same user or movie will contribute to the same part of the gradient. The naïve solution is just to add to the gradient atomically, but that has the potential to be a performance problem, even though any pair of threads is unlikely to be working on the same part of the gradient. An alternative is to add to the gradient non-atomically ("hog-wild" parallelism); however, because of the birthday paradox conflicts become virtually certain with more than a few hundred threads. Another challenge is how to efficiently use the GPU's memory structure on sparse arrays. An obvious optimization is to have each block work on points from only a few users and movies, and store the related features in shared memory. Yet another problem is making sure that the GPU's memory bandwidth and processing power can be used simultaneously. Synchronously copying over all of the data (or as large a batch as will fit in the GPU's memory) and then processing is obviously inefficient; one solution would be to asynchronously copy over the data needed by one block, launch the kernel for that block, and then continue copying data while that block is running.

An internet search does not bring up any similar work, except for a program called cuMF which uses a related Alternating Least-Squares algorithm to solve the same problem. That implementation is cleverly optimized and written to use multiple GPUs; however, it is virtually undocumented, poorly written, and uses a somewhat different algorithm from the one I plan to implement.

**Deliverables and Goals**

The main deliverable is an implementation of the above matrix factorization algorithm on the GPU that produces acceptable (<0.93 RMSE) predictions on the Netflix dataset. The other goal is performance: a decent GPU should provide a very large performance increase, provided the challenges described above can be surmounted. Without any similar work to base predictions off of, it is impossible to give a concrete performance goal, but the final product should give adequate consideration to the GPU's memory structure and using the GPU's FLOPSs and memory bandwidth simultaneously.

**Timeline**

I already have a working implementation of the algorithm on a CPU; however, it has a number of qualities which make it unsuitable as a demo for this class: for example, it relies on a number of external libraries and programs (Eigen, Cython, etc.) which will not be installed on the computer where I will run my demo, and it does not batch the data (so uses substantially more RAM than I can expect to have in VRAM). I will have to rewrite it to remove dependencies and to make it more readily suitable for GPU acceleration, which will occupy the first week. In the second week, I will write a working but naïve implementation in CUDA.  In the third week, I will explore ways of splitting up the data to remove conflicts, reading features in to shared memory, and asynchronously copying and launching kernels to maximize throughput.