Resource-aware Runtime System for ML Models across Heterogeneous Mobile Processors

Bernard, Nigel
Department of Electrical Engineering and Computer Science
University of California, Merced
Merced, California 95343,USA

Abstract

I extend the use the of the Lenskit Recommender Software to the Android platform to profile the execution of computationally intensive subtasks on the heterogeneous processors.

1 Introduction

ML models have a wide range of capabilities regarding mobile applications, such as speech to text, voice recognition and object detection.
On mobile platforms. However, the capacity for speedup and development of ML models is usually constrained for training on specific, optimized local architecture. The downsides of this approach are security concerns and network bandwidth for the inference that query and utilize the ML models implemented separately from the mobile platforms. In the interest of diversification and the downsides noted, research is emerging for development of ML models locally on mobile platforms. To allow for the optimal user experience, the ML model training should utilize the full capacity of the heterogenous processors within the mobile platform. This project was intended to extend the ML model training to allow for training on the mobile device processors and optimize execution of subtasks for an algorithm given profiling information.

In the interest of ML models, there are specific subtasks that work best for different processors. Clearly the GPU would execute the more computationally intensive aspects of a ML model training than the CPU or DSP. However, in order to account for user experience, it is desired to also analyze the current state of the respective processors to determine the correct schedule of a particular subtask of a ML model training. Factors to consider are execution time, power consumption, and memory allocation. A runtime model optimizes the scheduling of these subtasks depending on the relative weight of each of these factors.

In this project, I extend the execution of the Lenkskit recommender toolkit with openCL in order to enable parallelization. Computationally extensive subtasks within the K-NN item- item collaborative filtering and the funkSVD matrix factorization are considered. These subtasks will be scheduled on either the GPU or the DSP.
The rest of the paper is structured as follows. The next section gives a background of pertinent aspects of the design. Section 2 discusses the components of the two algorithms that are mainly eligible for speedup. Section 3 outlines the drawbacks of some aspects of the project. Section 4 outlines approach to SVD factorization and KNN. Section 5 outlines better ways to improve the algorithm speedup and approach and future work.

2 Background

2.1 DataSet

The Recommender toolkit requires a large data set to recommend items for a particular user. For this project, I utilized the MovieLens 100k dataset from https://grouplens.org/datasets/movielens/latest/. Ratings are on a scale from 1 to 5 with 3,600 tag applications applied to 9,000 movies with 600 users.

2.2 Processors (Mac OSX)

CPU: Intel Core i7-8559U CPU 2.70GHz
GPU: Intel Plus Graphics 655
65536 Local Memory Size
1 GB Global Memory Size

2.3 Lenskit Recommender Toolkit

This project uses the Lenskit recommender toolkit developed by the Lenskit group at the HPC Research Center Department of Computer Science and Engineering to provide a baseline for the K-NN and SVD algorithms to profile the speedup implementations. The MovieLens 100k dataset was used in order to provide the toolkit with users, items and ratings.

2.4 Algorithms
To narrow the depth of our analysis, I measured speedup on the K-NN Item Collaborative Filtering and SVD Matrix Factorization. The following provides background on the two algorithms.

2.4.1 K-NN item Collaborative filtering
Collaborative Filtering builds models using items rated from the target user as well as ratings from all other users in the particular dataset.
Convert the dataset into a UxI matrix for computations with ratings as elements within the matrix. Then Fill in the missing entries with 0 to allow for vector computations. Because the dataset is so sparse, I have to implement a sparse iterator that will make assumptions about the content of the data from specific calculation. If an element is assumed to be empty, the iterator will skip. This will cause the evaluation to be less accurate, but will clearly allow for faster iteration and computation.

The item-based approach for collaborative filtering is often preferred when compared to the user-based approach. Users are a very malleable data component, as users constantly added and deleted to a particular database. The items within a dataset do not change very often and so building a model based on those elements is more likely to result in a less fluctuating and consistent model.
KNN relies on cosine similarity in order to calculate the distance between neighbors to account for the dimensionality of these large datasets.
Then the Recommendation System returns the top K closest items in for recommendation.

2.4.2 SVD Matrix Factorization
For a given matrix that is sparse, I implemented an algorithm that operates on a packed matrix. Let's look at SVD to see how to factorize a packed matrix. Each original vector can be expressed as a linear combination of the eigenvectors of the covariance matrix. Each of the original user vectors can be expressed as a linear combination of the qualitative features [1].

$$\begin{aligned}\text{Alice} &= 10\%\ \text{Action fan} + 10\%\ \text{Comedy fan} + 50\%\ \text{Romance fan} + \cdots\\ \text{Bob} &= 50\%\ \text{Action fan} + 30\%\ \text{Comedy fan} + 10\%\ \text{Romance fan} + \cdots\\ \text{Zoe} &= \cdots\end{aligned}$$

[1]

SVD will return the dot product of the typical user and typical item vectors for the top K items for a given user. The feature vectors of users and items represent the affinity that a user has for a given item and vice versa. In the case of a sparse matrix, I cannot compute the feature vectors because of the nonexistent ratings in the array. Using a mean rating in order to fill in the elements will also result in biased predictions[1].
Instead, it is sufficient to simply ignore these nonexistent values within the feature vector calculations.Every feature iteration, FunkSVD uses an optimization evaluation in order to compute the feature vectors on the existing entries within the rating matrix by the following equation:

$$\min_{\substack{p_u, q_i\\ p_u \perp p_v\\ q_i \perp q_j}} \sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

[1]

The basic steps of the FunkSVD model builder is:

Compute the error of the prediction for the given rating and compute the rsme.

```
For # iterations:
    For # epochs:
    Update the user feature vector for every rating with the following computations:
        User feature vector pu: pu+α·qi(rui−pu·qi)
        Item feature vector pu: qi+α·pu(rui−pu·qi)
```

[2]

In the algorithm the # of features was 25, while the # of epochs was 125. The subtask most eligible for speedup is clearly the two feature vector update computations.

3. Drawbacks and Adaptations

Lenskit on Android
Unfortunately, I was unable to extend the ML models for execution on the Android platform. The ClassLoader function to dynamically instantiate classes like the Rating Entity and the configuration files proved to be the main obstacle for Lenskit integration. This is because the android runs on the Android Runtime (ART) VM instead of the JVM to optimize size and processing speed. The ART VM utilizes .dex flies instead of .class files to load particular classes. So when the ClassLoader function is called from the Lenskit project ran on Android, a

"class cannot be loaded" error is generated. Lenskit utilizes javax.annotations in order to allow for different assumptions to be made about particular methods/classes across different modules within the lenskit project. Of course, the ClassLoader function has a functional counterpart in the ART VM: the DexClassLoader. However, the DexClassLoader is not available to Java-Libraries and only availiable to Android Libraries. Unfortunately, forcing the Lenskit libraries to a Android Libraries did not provide a solution. Lenskit requires several classes that the android runtime does not support. For instance, javax.annotations. Lenskit uses javax annotations in a variety of ways. The most notable is to instantiate specific getters and setters within the Rating Entity class. In addition, Lenskit also utilizes groovy imports which the android runtime system does not include. I also tried to retain the Lenskit libraries as java-libraries and pre-compile the necessary java classes to .class files and then .dex files using the Android SDK Tools. I tried loading these .dex files into the local android files by accessing all necessary classes by their FQN within the assets folder. However, in order to access these classes, the Lenskit libraries must no longer extend their preexisting abstract classes. Because of these complications I decided to implement the project locally on the MacBooks hardware as the resulting speedup and prediction model should be able to profile execution on the Android in the case that Lenskit is configured with the Android in the future.

4. Algorithm Extension

Project Difficulties in Implementation
K-NN Item-Item Collaborative Filter
I was not able to fetch the sparse iterator and the sparse matrix for the cosine evaluation speedup of the k- nearest neighbors in OpenCL. To do this, it would require me to pass the iterator object to the native side and iterate through the entire matrix. This can be implemented in future work.

SVD Matrix Factorization Speedup
The difficulty within the speedup of the SVD Matrix Factorization is the dependency of the feature vectors dependencies upon previous updates. This means that for an epoch, I can only parallelize computations that are isolated in item and users. Each rating within a safework iteration within an epoch must have a different user and item associated.
In order to compute the iterations for the epoch cycles, I tried several different implementations. Firstly was a conversion of the ratings matrix to a diagonal array. In this way, different ratings would be isolate from one another along the diagonal. However, the level of iterations would be 9000 for a 100k dataset. This is way too many iterations effective enough for iterations. In addition, list that contained the iterations often times contained only 1 element, which is to be expected within a sparse matrix.

The next implementation was to first create the User and Item Matrix in parallel with the serial computation. This was done by creating a sub Rating List to both isolate all empty elements for issuing for the GPU for speedup. Each safe iteration for the 100k ML data set started at 700 elements for a single iteration and progressively decreased until it gathered all ratings into a single list with the rows a different safework iteration and cols with a row the parallelizable elements. Unfortunately, this algorithm did not produce an appreciable speedup upon the initial serial computation for the epochs. Considering that each individual execution took around .0007 seconds, it seems as though the preparation for each iteration is the main cause for the lack of speedup for the algorithm. Padding the arrays desired for computation would make the memory access timer much faster. Also, before execution, the needed parameters for the optimization calculation would be collected into arrays coinciding with a specific rating's safe execution. It would be better to import these directly into the device all at once instead of just continuously passing those memory elements within every single iteration. As it stands now, each iteration first starts with the initialization of a buffer and a writing to a buffer before doing the kernel computation. The computation however took way to look to even be remotely feasible(15s).

The last implementation was to import the entire rating matrix and the required components for the calculation for kernel execution. In this case, accuracy is not optimal, as the feature vectors are computed in one single iteration. This means that ratings with the same user and item in common may result in a data race for the update of an element of a feature vector. This is not in the true spirit of funkSVD, however there are algorithms that compute the feature vectors in parallel.

5. Future Work

Runtime System Specifications

In order to implement the runtime system, a prediction model considering the execution of the subtasks of the K-NN and SVD algorithms will have to be considered. In particular, using the funkSVD recommender to evaluate the subtasks denoted as tasks, and their performance on the users denoted as the GPU, CPU and DSP. For a given size of the subtask, denoted as the features of the SVD algorithm. From this information, Lenskit will allow the storage and access of a Prediction model that the Runtime System can query for a given input size of a particular subtask.

The runtime system will have to account for the allowance of memory accesses throughout different processors. Based on a particular subtask, the continuous computation may execute on different processors but require the same memory to either access or write data. This will complicate the development of the runtime model because efficient scheduling will have to analyze these particular cases.

6 References

[1] K. Liao, "Prototyping a Recommender System Step by Step Part 1: KNN Item-Based Collaborative Filtering," *Medium*, 19-Nov-2018. [Online]. Available: https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn-item-based-collaborative-filtering-637969614ea.

[2]Rabahallah, Kahina, et al. "MOOCs Recommender System Using Ontology and Memory-Based Collaborative Filtering." *Proceedings of the 20th International Conference on Enterprise Information Systems*, 2018, doi: 10.5220/0006786006350641.