Matrix Factorization with OpenMP and CUDA

Wei-Ting Chen†, Po-Chun Yang†, En-Hung Chu†, Pin-Chun Hsu†
† Department of Computer Science, University of California, Los Angeles.
{weitingtw, enhungchu}@cs.ucla.edu;

Abstract—**High Performance Computing plays a big role in the modern Machine Learning world. In this project, we will go through a common scenario in Machine Learning, optimize the implementation in different ways, and finally evaluate them. We chose Recommender Systems as our main problem to solve, and our implementation is bulit upon an open source project provided by Pedrorio et al from Instituto Superior Técnico [1]. We applied Collaborative Filtering on this problem, which is a methodology widely used in Recommender Systems, and Matrix Factorization is a way to solve a collaborative filtering problem. The purpose of matrix factorization is to discover the latent variables within the relationship between two sets of variables, e.g. users and items. In this paper, we parallelize the calculation in Matrix Factorization using OpenMP and CUDA.**

I. Introduction

A. Collaborative Filtering

Collaborative filtering (CF) is a family of algorithms that could find the hidden features between users and items, and hence generate the recommendations [2]. It observes user behavior to make recommendations; moreover, there is no need to provide detailed information about each user – a great collaborative filtering model could learn these relationships by itself. To be specific, a collaborative filtering model is based on that user who interact with items in a similar manner should share some hidden preferences. Secondly, the model also follows the principle that users with shared preferences are likely to have similar tastes to the same items. For instance, if user A is interested in Action Movies and so do user B, then it is possible that a random action movie such as Transformers could fit both of their tastes. Briefly speaking, we can leverage the model to find users' hidden interests. In this section, we will go through how we implemented the algorithm.

B. Matrix Factorization

Matrix factorization has been widely used and proven to be an effective approach in machine learning such as recommender systems. A recommender system is used to recommend items to users and it can be seen in applications such as Netflix where there is a set of users and items. Given that the users have already rated some of the items, the purpose is to estimate how the users would have rated other items, which they have not rated before. Matrix factorization has an edge in discovering latent features that underlie between two sets of objects, which is coincidentally very helpful in discovering the relationship between users and items in a recommender system.

Here is a simple illustration of matrix factorization for a recommender system. Suppose there are 5 users and 5 items

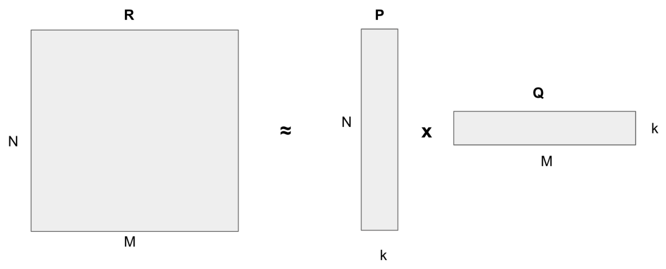

Fig. 1. Matrix Factorization: Rating Relationship

and it is represented as matrix R where the $R[1][0]$ represents the rating that user 1 gives to item A.

$$\mathbf{R} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 5 & 4 & 4 & - & 5 \\ - & 3 & 5 & 3 & 4 \\ 5 & 2 & - & 2 & 3 \\ - & 2 & 3 & 1 & 2 \\ 4 & - & 5 & 4 & 5 \end{pmatrix} \end{array}$$

The goal of matrix factorization is to find two matrices such that

$$P \times Q^T \approx R$$

where P and Q is of dimensions $N \times K$ and $M \times K$ respectively as shown in Fig. 1. Here N is the number of users, M is the number of items and K is the number of latent features to be discovered. We can view this process as compressing sparse information in $\boldsymbol{R}$ into much lower-dimensional spaces $N \times K$ and $K \times M$. by multiplying the user matrix and the item matrix, we could theoretically find an R' that is an approximation of R. An intuitive approach to find such matrices is to randomly initialize P and Q and perform Stochastic Gradient Descent (SGD) to approximate the product of P and Q with matrix R. This will stop until the error reaches a certain threshold. This in fact turns out to be successful to find P and Q. However, in the implementation, a regularization component is often considered to obtain better results.

To elaborate, we take the dot product of the two vectors: **u** and **i** of the user u for item i, and then we can express this relationship as:

$$\mathbf{r}_{ij} = \sum_{k=1}^{k} \mathbf{p}_{ik}\mathbf{q}_{kj}$$

Now, we have to find a way to obtain P and Q, and that's when SGD comes into play.

C. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an iterative method for optimizing an the objective function [3], and SGD can perfectly fit into such use cases as objective function for Matrix Factorization is differentiable [3].

$$minimize \;\; \mathbf{e}_{ij}^2 = (\mathbf{r}_{ij} - \sum_{k=1}^{k} \mathbf{p}_{ik}\mathbf{q}_{kj})^2$$

Here we consider the squared error since that the estimated rating could be either higher or lower than the rating score. To minimize the error, we will do partial differentiation to the above equation, with the two variables **p** and **q** respectively. As the formula is differentiable, we could finally obtain the gradient [4].

II. Implementation

A. Serial

The serial implementation is going to be served as the control group in the experiments. Therefore, the serial implementation is standard and straightforward. Our implementation follows the pseudo code below.

```
1   function mat_fac(P,Q,R,K, steps, alpha){
2      for step in range(steps)
3         for i in range(len(R))
4            for j in range(len(R[i])
5               err = calculate_err(R[i][j],
                      dot(P[i,:], Q[:,j]);
6               for k in range(K)
7                  update_err(P[i][k],
                      alpha, err);
8                  update_err(Q[k][j],
                      alpha, err);
9   }
```

From line 3 to 4, the function iterates over every single entry in matrix R. In line 5, it calculates the error based on the current value inside the matrix R and the dot product between the corresponding row and column respectively in the other two matrices. Then in line 7 and 8, the code updates the new values in matrix P and Q. [5]

B. OpenMP

The OpenMP version uses the same code logic as the serial version. However, since the OpenMP version requires a certain degree of parallelization in order to speed up the overall runtime, the OpenMP code is restructured. Notice that in the serial version, the code directly manipulates on matrix P and Q. In order to parallelize this specific part, the OpenMP needs to store the two matrices before each iteration so that the updates can be done in parallel because we would like to update the matrix based on the values in the previous iteration. This is achieved by the following code snippet.

```
    #pragma omp parallel for private(k, n,
        m) schedule(static)
       for (int k = 0; k < K; k++) {
          for (int n = 0; n < N; n++) {
             tempP[n][k] = P[n][k];
          }
          for (int m = 0; m < M; m++) {
          tempQ[k][m] = Q[k][m];
        }
     }
```

Then we would like to parallelize the computations for the errors like the serial version does. Notice that for each cell in matrix R, its value is computed by using the dot product between the corresponding row and column in matrix P and Q. Therefore, there is no dependency between these computations and we can parallelize them by assigning to each thread a certain portion of these dot product computation. This is done by the following code.

```
#pragma omp parallel for private(z, k)
    schedule(static)
       for (int z = 0; z < Z; z++) {
          prediction[z] = 0;
          errors[z] = 0;
          for (int k = 0; k < K; k++) {
             prediction[z] +=
                 P[nonZeroRow[z]][k] *
                 Q[k][nonZeroColumn[z]];
          }
          errors[z] =
              R[nonZeroRow[z]][nonZeroColumn[z]]
              - prediction[z];
       }
```

The final step is to update the original P and Q. Recall that we have already the values in variables *tempP* and *tempQ*. This allows us to parallelize the updates as mentioned, which is fairly straightforward by taking the errors for each entry and the variable alpha. So far, we successfully parallelize the computations for matrix factorization. Notice that the difference of the OpenMP version and the serial version is that OpenMP version updates the matrix all at once while the serial version updates for each iteration in the inner loop. This difference in practice doesn't have much influence in the results in the experiments.

C. CUDA

To enable our parallel computations in CUDA, we have to first understand how it works. In every GPU, there will be many kernels, and each kernel is in charge of one **grid**. Each grid has many thread blocks, and every block consists of a huge amount of threads. Our host (CPU) will copy data from CPU to GPU memory, and load GPU program to execute. Once it's finished, the result will be shipped back to our host. When a kernel got kicked off, threads will be distributed to many Streaming Multiprocessors. All threads in the same Thread Block will be put to the same place. Firstly, we look into our address calculation code. Each kernel is an array of thread blocks which run the same program code on different data, and we could easily compute the mapping between thread ID and data. We define the number of blocks as the number of non-zero elements; at the same time, we define the number of threads in each block as the size of our hidden feature. For instance, if we have a matrix R with 150 non-zero elements, and also we want to do dimension reduction down to 15, then we will have blockSize is 100 and threadSize = 15. In `cuda/src/updateLR.cu`, we implement the main algorithm of parallel computing. The following code snippet could help us retrieve our computation unit by index:

```
l = blockIdx.x;
k = threadIdx.x;
```

Then, we will take a look at how to calculate LR and also how we determine the delta value (i.e., the difference between our guess and the ground truth). Firstly, we always use our thread 0 to be the parent thread when forking. Later we will add up our delta value, and each thread will sum its value back to thread 0's. One thing to note is that we actually parallel the whole for loop with variable k to be a single line program. when we have k threads running concurrently, we could store our `predict` value temporarily, and then using `atomicAdd()` to aggregate all these sum in each thread.

```
if (k == 0) {
   prediction[l] = 0;
   delta[l] = 0;
}
__syncthreads();
double predict = L[nonZeroUserIndexes[l] *
    (*numberOfFeatures) + k] * R[k *
    (*numberOfItems) +
    nonZeroItemIndexes[l]];
atomicAdd(&prediction[l], predict);

__syncthreads();
```

Also, be aware of that we have to do sync threads everytime we finish an operation. The next step would be that we are going to sum up the delta value back to each block. We still apply the same methodology as before, says, we always focus on the thread 0:

```
if (k==0) {
   delta[l] = A[nonZeroUserIndexes[l] *
       (*numberOfItems) +
       nonZeroItemIndexes[l]] -
       prediction[l];
}
__syncthreads();
```

Finally, we calculate the gradients for doing partial derivative on L and R respectively. We will aggregate the gradients back to our L and R to help us optimize our computation in the next iteration.

```
double gradientL = *convergenceCoefficient
    * (2 * delta[l] * R[k *
    (*numberOfItems) +
    nonZeroItemIndexes[l]]);
double gradientR =
    *convergenceCoefficient * (2 *
    delta[l] * L[nonZeroUserIndexes[l]
    * (*numberOfFeatures) + k]);

__syncthreads();

L[nonZeroUserIndexes[l] *
    (*numberOfFeatures) + k] += gradientL;
R[k * (*numberOfItems) +
    nonZeroItemIndexes[l]] += gradientR;
```

The number of iterations is defined in `cuda/src/matFact.cu`. Just to note, we create, initialize, copy and free all CUDA variables in that separate file, too. We leverage `cudaMalloc()` and `cudaMemcpy()` and so on to help us achieve our goal.

TABLE I
SERIAL VS OPENMP EXPERIEMENTS TEST CASES

	users	items	features
T1	30	40	10
T2	1000	1000	100
T3	200	10000	50

TABLE II
SERIAL VS OPENMP EXPERIEMENTS TIME

	T1	T2	T3
Serial	1.50 s	428.68 s	438.47 s
OpenMP (1 thread)	0.74 s	35.62 s	45.71 s
OpenMP (2 threads)	1.61 s	21.25 s	26.34 s
OpenMP (4 threads)	1.64 s	13.52 s	15.58 s
OpenMP (8 threads)	2.45 s	12.38 s	13.92 s

III. EVALUATION

A. Serial and OpenMP

We have designed three test cases for the serial vs. OpenMP comparison, which is available in Table 1. Further Table 2 displays the results obtained from running the test cases. We run all the cases with an Intel Core I5 CPU. In general, we can observe that the OpenMP runs significantly faster than the serial version, which meets our expectations since the OpenMP version takes advantage of parallelization. When we take a further look at the results for OpenMP, we've found that for T1, which is a case for smaller matrices, increasing the number of threads actually increases the time. For T1 and T2, increasing the number of threads does reduce the time, which meets our expectations. We can deduce that this happens because the benefits we gain from running multiple threads for T1 are less than the additional burden, such as context switching, for running in parallel. That being said, in reality, a mature recommender system usually have a large

TABLE III
CUDA EXPERIMENTAL RESULTS (# OF ITERATIONS = 50000, α = 0.001)

	inst500-500	inst1000-1000	inst200-10000
Average Time Elaspsed	13 s	26 s	1 m 17 s

amount of users and items respectively. Therefore, running in parallel with OpenMP seems to be a better option than the serial version.

B. CUDA Acceleration

In our experiments, we use CUDA toolkit V10.1, and we have one NVIDIA GTX 1080Ti as our main GPU. As a result, our experiments have shown that using CUDA could significantly improve our matrix factorization performance. In fact, it is not even at the same order compared to Serial and OpenMP. CUDA is way more faster than that, for instance, we could finish a 500×500 matrix factorization in 13 seconds, and this largely surpassed other methods. Interestingly, we noticed that we could finish a 1000×1000 tasks in 26 seconds, which is exactly half of the previous task's execution time. We might want to argue that as the size of our matrix grows 4 times, it will need twice the time to execute such larger task.

We made three experiments based on CUDA. We first run our experiments in some small datasets to ensure that our result is correct; then, we turned to the other direction and focus on `inst500-500`, `inst1000-1000`, `inst200-10000`. As we could see in Table 3, for these large dataset that need more than a minute for serial version, now it only need less than thirty seconds or so. For the task `inst200-10000`, it took 157MiB memory to run; and for task `inst1000-1000` it needed 145MiB. Finally for the task `inst500-500` it only need 139 MiB. This result matches our hypothesis that a larger dataset will need more memory, and yet the usage is not necessarily in proportion to the matrix size. To sum up, we would like to argue that CUDA has the best performance among three methods, with sacrificing accuracy and correctness.

IV. Conclusion

In this project, we dive into three different implementations of Matrix Factorization, with an aim of leveraging the most cutting-edge High Performance Computing frameworks to solve Recommender Systems problem. We have Serial version as our baseline, and we build OpenMP and CUDA version on top of it. We briefly introduces how we realize the functionalities, and how we evaluate them. As a result, OpenMP could easily beat the baseline significantly, and moreover, our CUDA version could even finish the top three largest tasks in around 1 minutes, which is even more better. Finally, we validated the powerfulness of parallelism and discussed the pros and cons of each framework.

References

[1] P. R. et al, "parallel and distributed computing, available on github," 2020. [Online]. Available: https://github.com/pedrorio/parallel_and_distributed_computing

[2] B. S. et al, "Item-based collaborative filtering recommendation algorithms," *WWW*, 2001. [Online]. Available: http://files.grouplens.org/papers/www10_sarwar.pdf

[3] H. R. et al., "A stochastic approximation method," 1951. [Online]. Available: https://projecteuclid.org/download/pdf_1/euclid.aoms/1177729586

[4] L. Bottou, "Large-scale machine learning with stochastic gradient descent," *COMTSTAT*, 2010. [Online]. Available: https://leon.bottou.org/publications/pdf/compstat-2010.pdf

[5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.