# PARALLEL IMPLEMENTATIONS FOR SOLVING MATRIX FACTORIZATION PROBLEMS WITH OPTIMIZATION

by

AMIR EMAMI GOHARI

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University
Summer 2016

PARALLEL IMPLEMENTATIONS FOR SOLVING MATRIX FACTORIZATION PROBLEMS WITH OPTIMIZATION

APPROVED BY

Prof. Ş. İlker Birbil
(Thesis Supervisor) ..................... [illegible]

Assist. Prof. Kamer Kaya
(Thesis Co-supervisor) ..................... [illegible]

Assoc. Prof. Taylan Cemgil ..................... [illegible]

Assoc. Prof. Kemal Kılıç ..................... [illegible]

Assist. Prof. Sinan Yıldırım ..................... [illegible]

DATE OF APPROVAL: 30/ 8/ 2016

*to my family*

**Acknowledgments**

First of all, I would like to express my deepest gratitude to my thesis advisers, whose expertise, understanding and guidance added a lot to my graduate experience. I am very grateful to Prof. Ş. İlker Birbil for his confidence in me and his encouraging supervision, which relieves me even under most stressful conditions. I owe my sincere thanks to Assist. Prof. Kamer Kaya for his continuous understanding and endless support throughout the course of preparing my thesis.

Words are not enough to express my thankfulness to my dearest friends Ashkan Eshghi, Kazem Afghah, Farid Irani, Atabak Maherkia, Hooman Hidaji and Tohid Teimourzadeh. Their presence has given me happiness and strength, and I wish to feel their love and support forever. And finally, I doubt that I will ever be able to convey my appreciation fully, but I must mention my eternal gratitude to my beloved family. Without them supporting and trusting me by all means, and their never-ending love, not only this thesis, but I myself would not be where I am, at all.

# PARALLEL IMPLEMENTATIONS FOR SOLVING MATRIX FACTORIZATION PROBLEMS WITH OPTIMIZATION

Amir Emami Gohari

Industrial Engineering, Master of Science Thesis, 2016

Thesis Supervisors: Prof. Dr. Ş. İlker Birbil
Assist. Prof. Dr. Kamer Kaya

Keywords: matrix factorization, parallel programming, MovieLens, nonlinear optimization.

## Abstract

During recent years, the exponential increase in data sets' sizes and the need for fast and accurate tools which can operate on these huge data sets in applications such as recommendation systems has led to an ever growing attention towards devising novel methods which can incorporate all the available resources to execute desired operations in the least possible time.

In this work, we provide a framework for parallelized large-scale matrix factorization problems. One of the most successful and used methods to solve these problems is solving them via optimization techniques. Optimization methods require gradient vectors to update the iterates. The time spent to solve such a problem is mostly spent on calls to gradient and function value evaluations. In this work, we have used a recent method, which has not been used before for matrix factorization. When it comes to parallelization, we present both CPU and GPU implementations. As our experiments show, the proposed parallelization scales quite well. We report our results on MovieLens data set. Our results show that the new method is quite successful in reducing the number of iterations. We obtain very good RMSE values with significant promising scaling figures.

# Table of Contents

# List of Figures

# List of Tables

CHAPTER 1

# INTRODUCTION

In the mathematical discipline of linear algebra, a matrix decomposition or matrix factorization is a factorization of a matrix into a product of matrices. There are many different matrix factorization techniques; each finds use among a particular class of problems[1].
Some of the most famous techniques for matrix factorization are:

- LU decomposition
- Block LU decomposition
- Rank factorization
- Cholesky decomposition
- QR decomposition
- RRQR factorization
- Interpolative decomposition

In our case, we are using *Rank Factorization.* Figure 1.1 schematically depicts rank factorization problem.

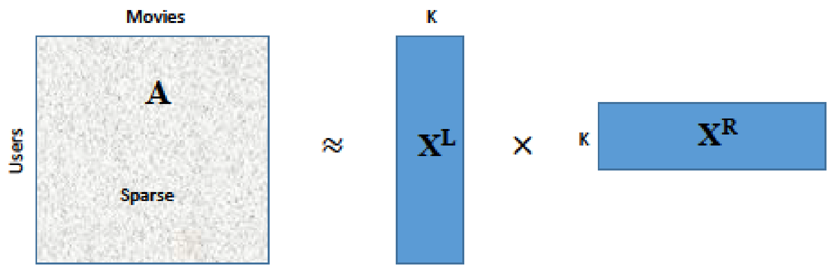


Figure 1.1: Illustration of Matrix Factorization

This problem can be formulated as

$$\underset{X^L, X^R}{\textbf{minimize}} \quad \|\mathbf{A} - \mathbf{X^L} \times \mathbf{X^R}\|_F^2 \quad , \tag{1.1}$$

where $\|.\|_F^2$ is the Frobenius norm, $\mathbf{A}$ denotes the $m \times n$ input matrix which in our case is a sparse matrix containing users' ratings for movies. $\mathbf{X}^L$ and $\mathbf{X}^R$ are the matrices which their product is going to reconstruct the approximation of $\mathbf{A}$ matrix. This formulation denotes that we are using optimization to solve a "Matrix Factorization" technique to solve the problem.

One of the most important applications of solving a large-scale matrix factorization via optimization techniques can be found in *Recommendation Systems*. Recommendation Systems are a subclass of information filtering system that seek to predict the 'rating' or 'preference' that a user would give to an item. Since the diversity of products are growing rapidly and the number of users who exploit these services increase every day, we must be able to solve huge problems of this type as fast and as accurate as possible.

Table 1.1 depicts the implementation of an optimization technique on different MovieLens data sets, measuring the time spent on function value and gradient evaluation (we call these two components *Funciton* in our table) compared to the whole time spent on solving each problem. As Table 1.1 shows, the main bottleneck in computation is due to gradient and function value evaluations.

Table 1.1: Portion of Time Spent on Gradient and Function Value Evaluation

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 1.46 | 1.68 | 86.78% |
| 1M | 14.62 | 15.62 | 93.65% |
| 10M | 234.09 | 245.11 | 95.51% |
| 20M | 554.36 | 582.18 | 95.22% |

Our objective in this paper is to come up with different parallel implementations to reduce the time spent in these functions. Although it is still important to reduce the number of optimization iterations, a significant portion of each iteration is reserved for the calls to gradient and function value evaluations. Hence, as it has been tried in this work, reducing the time spent on these functions through techniques such as parallelization, will effectively reduce the time needed to solve this type of problem.

## 1.1 Contributions of This Study

The main contributions of this paper can be summarized as such:

1. To implement a technique for optimization which has not been used for matrix factorization before, in such a way that it can be implemented in parallel.

2. To make several parallel CPU and GPU implementations using different techniques to observe how the efficiency of our algorithm improves.

3. To provide a framework, such that the reader can implement their own technique and improve its performance through parallelization.

we explained the algorithm we are working on. Next, in section 4 we will explain the way we designed our experiments and the data-sets we are going to work on. Finally we will have our conclusion and possible grounds for making future improvements.

## 1.2 Outline

The outline of the thesis is as follows. Chapter 2 gives a literature review of the matrix factorization problems with an emphasis on parallel programming. This literature review is followed in Chapter 3 by the introduction of our problem and our proposed approach. Our numerical experiments have been given in chapter 4 where we will explain the way we designed our experiments and the data-sets we are going to work on. we have also explained about each implementation and their features. We end this thesis with Chapter 5, which contains the conclusion and the planned future work.

CHAPTER 2

# LITERATURE REVIEW

In this section, we review some of the best efforts done so far to develop techniques which work better than other existing methods and the way our work makes difference. This topic has got quite popular recently and it has been tried to find the most successful works but the first attempts for execution of linear algebra operations could be found in works of [2] which implements and describes the necessary steps for parallelization of BFGS method. Also [3] proposes the usage of truncated-Newton methods combined with computation of the search direction via block iterative methods.

In recent years, the number of papers working on parallel implementation of optimization techniques has been enormous, hence, it is hard to go through all of them. But there are several works which reviewed the most important techniques and can be used as a reference; such as [4] which summarizes the recent research on factorization machines providing extensions for the ALS and MCMC algorithms. It observes how the behavior of each method would change as the size of hidden features increase. In this case, Stochastic Gradient method outperforms others, which we also use as a method to compare our optimization technique upon it.

Among the most important works in recent years, there are some papers which emphasize on the model and its parameters and try to modify it to obtain better results. [5]introduces a new method of matrix factorization called Bounded Matrix Factorization(BMF), which imposes a lower and an upper bound as the constraints on each missing elements that is going to be predicted. Then it presents an efficient algorithm for this new method and discusses that this algorithm can be scalable for web-scale data sets and can be computed on parallel systems with limited memory. [6] focuses on the problem of sparsity. It is mentioned that even though matrix factorization technique performs better than many other techniques in case of dealing with sparsity, but still it is far from perfect. Previous methods to solve this problem, usually take advantage of auxiliary information. But in this case, without having auxiliary information, the performance has been improved exploiting cosine similarity. Authors also devised an experiment which assesses the performance of this method compared to regular matrix factorization. Finally it is stated that if the number of available data increases, this novel method may not achieve such an advantage over regular matrix factorization. The issue of sparsity also has been addressed in the works of [7] where sparsity problem caused by considering only latent factors in recommendation systems has been examined. It deploys the attributes of users and items to strengthen its latent feature based model and defines them as explicit factors (as opposed to implicit or latent factors). In fact, this model tries to fuse implicit and explicit features into one model through *bias* and observes that it outperforms the models which are fully implicit based.

In some of the recent works, a new method has been devised and its performance has been compared to existing methods.[8] develops a new type of Stochastic Gradient Decent (Fast Parallel Stochastic Gradient Decent) that can be parallelized to solve web-scale problems. After describing the algorithm and mathematical logic behind it, some experimental results have been brought to show the performance of this new method compared to state-of-the-art methods such as: DSGD, HogWild and CCD++. Tests have been done on Netflix, Yahoo! Music and MovieLens datasets and it has been observed that the proposed method outperforms other ones. [9] proposes Coordinate descent, a classical method which has been used for large-scale problems, but has not been considered as a technique for matrix factorization. It develops the algorithm for the proposed method and observes that it performs faster than ALS and SGD with better convergence.

Some of the most recent attempts for parallel implementation to solve the matrix factorization have tried to manipulate an existing and established method in order to make it more suitable for parallelization implementations. Chin et al [10] is a good example for this type of paper. They state that stochastic gradient (SG) method is one of the most popular algorithms for matrix factorization. However, as a sequential approach, SG is difficult to be parallelized for handling web-scale problems. they develop a fast parallel SG method, FPSG, for shared memory systems. By reducing the cache-miss rate and addressing the load balance of threads, they conclude that FPSG is more efficient than state-of-the-art parallel algorithms for matrix factorization. In another work [11] a high-performance distributed-memory parallel algorithm has been proposed that computes the factorization by iteratively solving alternating non-negative least squares (NLS) subproblems for W and H. It maintains the data and factor matrices in memory, uses MPI for interprocessor communication, and in the dense case, minimizes communication costs. This algorithm is also flexible since it performs well for both dense and sparse matrices, and it allows the user to choose any one of the multiple algorithms for solving the updates to low rank factors W and H within the alternating iterations. Another sample of this group of papers [12] proposes a fast and robust parallel SGD matrix factorization algorithm, called MLGF-MF, which is robust to skewed matrices and runs efficiently on block-storage devices (e.g., SSD disks) as well as shared-memory. MLGF-MF uses Multi-Level Grid File (MLGF) for partitioning the matrix and minimizes the cost for scheduling parallel SGD updates on the partitioned regions by exploiting partial match queries processing. MLGF-MF works such that CPU keeps executing without waiting for I/O to complete. Thereby, MLGF-MF overlaps the CPU and I/O processing, which eventually offsets the I/O cost and maximizes the CPU utility.

Another group of papers are focusing on the learning rate. Srivastava [13] emphasizes on the fact that considering the ratings in datasets such as MovieLens as numbers would result in a huge presumption which is not valid. Its example is the difference between rating 1 and 2 is not necessarily the same as the difference between 4 and 5. Based on this reasoning, it categorizes the ratings as ordinal data and proposes two methods called Logestic Regression Model (LRM) and Cumulative Model. In the next step, it designs several experiments to compare its work to the ones yielding best results before. Briefly speaking one can say these methods obtained better results for recommending movies with rating 5, but their performance decreases when it comes to all the ratings. Luo et al. [14] states that the learning rate is a key factor affecting the recommendation accuracy and convergence rate in matrix factorization; however, this essential parameter is difficult to decide, since the recommender has to keep the balance between the recommendation accuracy and convergence rate. Authors consider Regularized Matrix Factorization as the base model to discuss the effect of the learning rate in matrix factorization based CF to observe the effects of tuning learning rate adaptation. After proving the effect of learning rate on recommendation performance, they propose three learning rate adaptation namely: Deterministic step size (DSSA), Incremental Delta Bar Delta (IDBD) and Stochastic Meta Decent (SMD). After analyzing the characteristics of each parameter update in RMF, they propose Gradient Cosine Adaptation (GCA) and they observe the experimental results on five public large datasets.

CHAPTER 3

# PROPOSED APPROACH

Considering the mathematical aspect of the problem at hand, as Equation (1.1) shows, we want to solve an unrestricted optimization problem. We already mentioned that these problems are solved through optimization techniques. Hence, finding the proper optimization technique is an important facet of successfully solving the problem. We will discuss this matter extensively in section 3.1.

Having found the appropriate technique, we will implement the parallelization techniques and several tricks to observe how the performance of our program improves. This phase includes CPU implementations, where we use a multicore CPU and try to share the load of the process among its cores in the most efficient ways, as well as GPU implementations, where we incorporate GPU for the same purpose. For both parts, first we have developed a simple parallel code as our baseline, to see how different tricks would increase our speedup compared to our baseline.

The optimization method we use, needs the first order information of the function, as well as its value in each iteration. As Table 1.1 shows, most of the time is spent on these operations. Since we already know these are the expensive parts of our computations, lets take a look at these functions to see how they are mathematically formulated.

In our codes, we implemented these main calculations with three different functions. We use the matrix notation to briefly describe them. In the rest of the text, $\mathbf{A}$ denotes the $m \times n$ input matrix which is probably sparse, $\mathbf{A}_{i*}$ and $\mathbf{A}_{*j}$ denote data elements in the $i$th row and $j$th column of the $\mathbf{A}$, respectively, and $a_{ij}$ denotes a data element in the $i$th row and the $j$th column of $\mathbf{A}$. To clarify these notations, considering the data set we are going to use (MovieLens), matrix $\mathbf{A}$ is our original data, where $m$ users have rated $n$ movies. It doesn't mean each user has rated all the movies and most of them rated less than one percent of the available movies. That is the reason for sparsity of our matrix.

A solution $x$ is formed by two dense matrices $\mathbf{X}^L$ and $\mathbf{X}^R$ of dimensions $m \times K$ and $K \times n$, respectively. Let $x^L_{ij}$ and $x^R_{ij}$ denote the element in the $i$th row and $j$th column of $\mathbf{X}^L$ and $\mathbf{X}^R$, respectively. We call $K$ our **Factorization Rank** which is usually a small integer compared to number of rows ($m$) and number of columns ($n$).

- COMP($\mathbf{L}$, $\mathbf{A}$): This function calculates the difference (error) of our estimation of the data matrix; i.e. lets assume we are at point $x$. Values stored in vector $x$ compose our $\mathbf{X}^L$ and $\mathbf{X}^R$ matrices. This function calculates the error at each point $x$ by measuring the distance between matrix $\mathbf{A}$ and the guessed matrix which is the result of multiplication of $\mathbf{X}^L$ and $\mathbf{X}^R$ matrices. The mathematical model of this function is

$$\underset{\mathbf{x}}{Error} = \|\mathbf{A} - \mathbf{X}^L \times \mathbf{X}^R\|_F^2 \quad . \tag{3.1}$$

  Algorithm 1 simply shows the behavior of this function.

**Algorithm 1:** comp Function

```
1 $err \leftarrow 0$
2 for $a_{ij} \in \mathbf{A}$ do
3     $current \leftarrow 0$
4     for $k = 1 \cdots K$ do
5         $current \leftarrow current + (x^L_{ik} \times x^R_{kj})$
6     $err \leftarrow err + \|current - a_{ij}\|_2$
7 return $err$
```

- GRAD$_L$: This component calculates the gradient vector of the matrix $Z1$ which corresponds to the first $(m \times r)$ values of the gradient vector. This part can be mathematically formulated as

$$\underset{\mathbf{x}}{grad_L} = 2 \times \mathbf{X}^R \times (\mathbf{A} - \mathbf{X}^L \times \mathbf{X}^R) \quad . \tag{3.2}$$

Algorithm 2 shows how this function works.

**Algorithm 2:** gradL Function

```
1 for $j = 1 \cdots n$ do
2     for $ptr \in a_{.j} \neq 0$ do
3         for $k = 1 \cdots r$ do
4             $g_{jk} \leftarrow g_{jk} + X_{ptr,j} \times A_{ptr,j}$
```

- GRAD$_R$: This function calculates the last (r × n) elements of our gradient vector. This portion of our function can be mathematically formulated as such:

$$\underset{\mathbf{x}}{grad_R} = 2 \times \mathbf{X}^L \times (\mathbf{A} - \mathbf{X}^L \times \mathbf{X}^R) \tag{3.3}$$

In Algorithm 2, you can see the way this function works.

**Algorithm 3:** gradR Function

```
1 for $i = 1 \cdots m$ do
2     for $ptr \in a_{i.} \neq 0$ do
3         for $k = 1 \cdots r$ do
4             $g_{ik} \leftarrow g_{ik} + X_{i,ptr} \times A_{i,ptr}$
```

the code excerpt 3.1 shows the implementation of this function written in MATLAB.

Code excerpt 3.1: Function Evaluation and Gradient Calculation

```
function [f, df] = seuc_fun(x, Y, latent_dim, datasize)
[n,m] = size(Y);
        Z1 = reshape(x(1:n*latent_dim),n,latent_dim);
        Z2 = reshape(x(n*latent_dim+1:latent_dim*(m+n)),latent_dim,m);

        E = Y - (Z1*Z2).*(Y>0);
        f = full(0.5*sum(sum(E.*E)));
        f = f/datasize;

        G1 = -E*transpose(Z2);
        G2 = -transpose(Z1)*E;
        df = full([G1(:);G2(:)]);
        df = df / datasize;

end
```

## 3.1 Optimization Method

The mathematical model for the matrix factorization is a typical. There are various optimization algorithms that can be used to solve such a problem. As we shall see shortly, a significant percentage of the computation time is taken not by the operations in the optimization routine, but by the calls to the objective function and its gradient. Therefore, as long as we have one optimization method that converges fast in terms of iterations, we can safely devote our effort to parallelize the objective function and the gradient evaluations.

To select an optimization routine, we have solved the matrix factorization problem for MovieLens 1M data set with eight different optimization methods.

Here are the tested algorithms:

- ◇ SD: Steepest Descent
- ◇ CSD: Cyclic Steepest Descent
- ◇ BB: Barzilai and Borwein
- ◇ HFN: Hessian-Free Newton
- ◇ CG: Conjugate Gradient
- ◇ SCG: Scaled Conjugate Gradient

- ◇ PCG: Preconditioned Conjugate Gradient
- ◇ LBFGS: Large-scale Quasi Newton
- ◇ PMB: Preconditioned Model Building

We have programmed all these algorithms in MATLAB for this benchmarking study. The maximum number of iterations for each algorithm is 500 and each algorithm starts from the same randomly generated initial point. The MATLAB codes to reproduce these results are available online(http://www.cs.ubc.ca/ schmidtm/Software/minFunc.html). The figure 3.1 illustrates the final RMSE values obtained with each method. Except SD, the performances of the remaining methods are comparable. BB is the leading method.

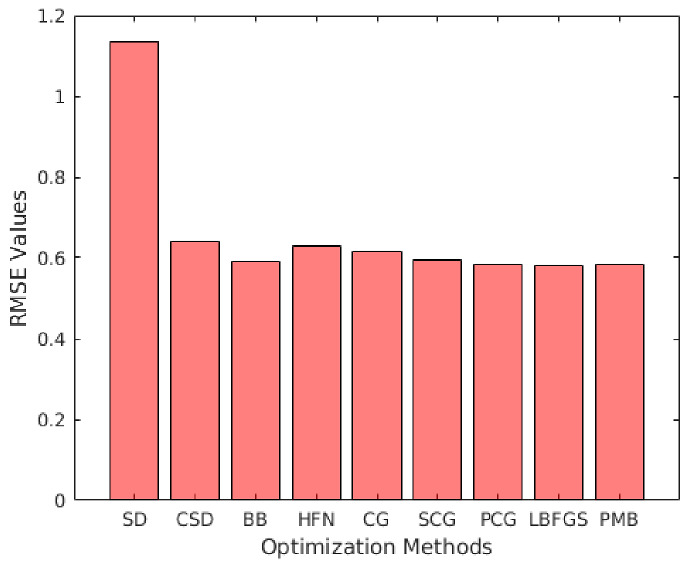


Figure 3.1: Comparison of Different Methods Based on their RMSE Values

To observe the quality of the obtained solutions, we have also evaluated the norm of the gradient at the final iterate of each method. BB has shown the best performance and it is followed by CSD and PMB.

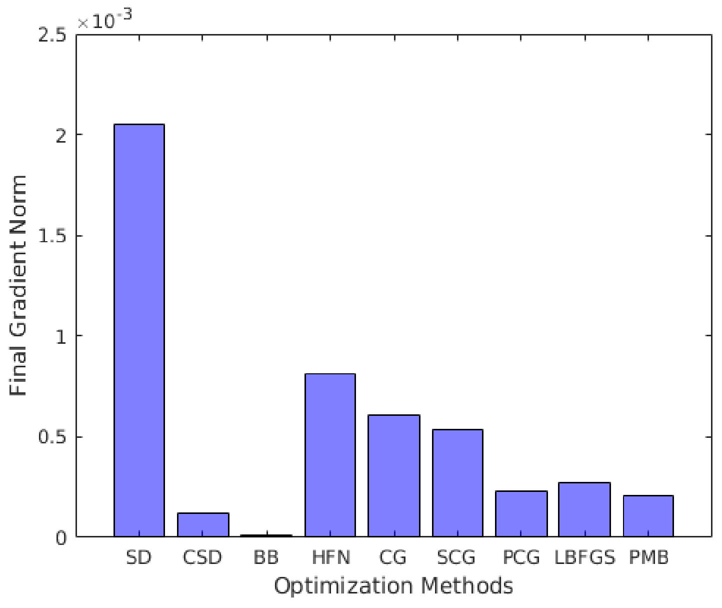


Figure 3.2: Comparison of Different Methods Based on their Gradient Norm

BB outperforms the other methods in terms of the final RMSE values and the precision of the first order optimality conditions. However, the following figure shows that the progression of BB in terms of RMSE values is not as good as the other methods in early iteration. This could have a significant effect for the certain application, where the end-users do not have time to run the optimization algorithm for more than only a few iteration. Surprisingly, the RMSE value of CSD jumps up at iteration nine, but then recovers immediately.

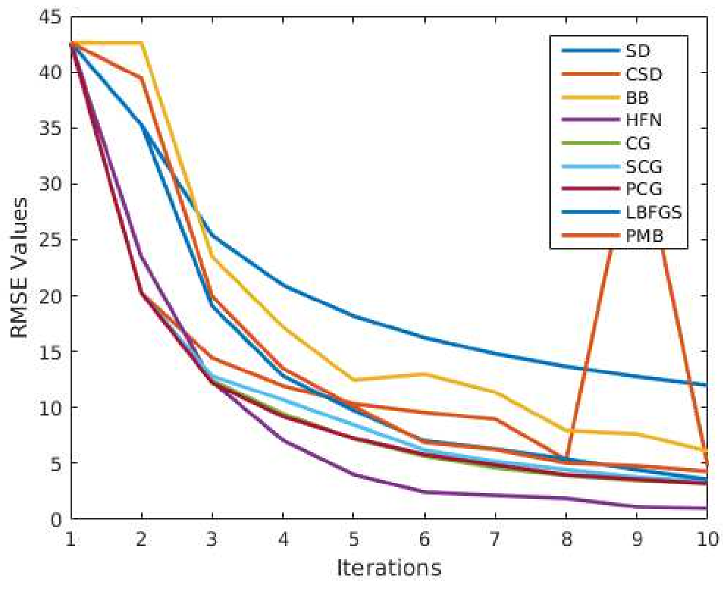


Figure 3.3: Comparison of Different Methods Based on their RMSE Trend

Overall, we have decided to use PMB. The reason for this choice is two-fold. First, our preliminary tests show that PMB is a competitive method, which has performed better than most of the other algorithms. Second, it is a new method, which could be an alternate unconstrained optimization method for the researchers in the machine learning community.

PMB uses the algorithm proposed by Figen Oztoprak [15] with some modification which is adding a L-BFGS preconditioning step, which makes the steps more efficient in terms of decrease in the function value. Limited-memory BFGS (L-BFGS or LM-BFGS) is an optimization algorithm in the family of quasi-Newton methods that approximates the BroydenFletcherGoldfarbShanno (BFGS) algorithm using a limited amount of computer memory. It is a popular algorithm for parameter estimation in machine learning[16, 17].

Like the original BFGS, L-BFGS uses an estimation to the inverse Hessian matrix to steer its search through variable space, but where BFGS stores a dense nn approximation to the inverse Hessian (n being the number of variables in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. Due to its resulting linear memory requirement, the L-BFGS method is particularly well suited for optimization problems with a large number of variables. Instead of the inverse Hessian $H_k$, L-BFGS maintains a history of the past m updates of the position $x$ and gradient $\nabla f(x)$, where generally the history size $m$ can be small (often m¡10). These updates are used to implicitly do operations requiring the $H_k$-vector product [18].

## 3.2 Base Model Analysis.

We need a baseline model to observe the effect of our parallelization tricks on our codes. To that end, we have implemented the code in C++ programming language. We tested all MovieLens data sets on this version of the code and brought the results in Table 3.1 for further comparisons.

Table 3.1: Baseline Model Analysis

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 1.46 | 1.68 | 86.78% |
| 1M | 14.62 | 15.62 | 93.65% |
| 10M | 234.09 | 245.11 | 95.51% |
| 20M | 554.36 | 582.18 | 95.22% |

The first column of this table corresponds to the time spent for calculating function value and gradient vector, which is the main focus of parallelization in this work. Percentage column shows the portion of time spent in the target functions. This percentage is important because as the number of cores/threads increase, it should get smaller. This can also be used as a measure, since a good parallelization can decrease this percentage more.

According to Table 3.1, the main function takes more than 90% of the total time spent by the code to finish the task.

CHAPTER 4

# NUMERICAL EXPERIMENTS

We use an Intel Xeon E7-4870 v2 processor with 2 sockets, each one has 15 cores running at 2.3 GHz and 30 MB of cache. We use OpenMP library for our CPU implementations. For our GPU experiments, we use an $NVIDIA\quad K40$ GPU. This GPU has a total of 2880 cores running at 745 MHz and 12 GB of memory. To compile our C++ codes, we use gcc version 4.9.2 compiler.We use CUDA version 6.5 for our implementations in GPU.

MovieLens in its original form, contains many rows, in each there are following elements:

1. User ID: An integer number assigned to each user.
2. Movie ID: An integer number assigned to each movie.
3. Rating: An integer representing the rating a user gave to a movie.

This rating matrix is considered sparse, since most of the elements are zero, which means users only rated a small portion of the movies listed on that data set. Furthermore, the number of movies each user rated may vary from one user to another. Some of them have 10 ratings, while some others have hundreds. As you may see later in our implementations, choosing a proper format to store the data has a huge effect on our performance. To that end, we have used CRS (Compressed Row Storage) and CCS (Compressed Column Storage) formats [19]. The CRS format consists of three vectors as well as some scalar values of interest: The column index(*col_ind*), *value* and row pointer (*row_ptr*) vectors. The nonzero elements on row $i$ of the matrix are located in indices $j$ in [*row_ptr(i)*, *row_ptr(i + 1)* - 1]. In other words, *col_ind(j)* and *vals(j)* for *j* in [*row_ptr(i)*, *row_ptr(i+1)*  1] represent the column index (unsigned integer) and value of matrix (double) on row *i* [20].

In some of our implementations, we have enhanced these matrix storage techniques by adding other arrays which yield additional functionality.

For our experiments on CPU we ran each version of the code on 1, 2, 4, 8 and 16 threads. Each of them has been repeated 10 times and the average for each value has been taken.

In GPU experiments, we ran each implementation on blocks of sizes 32, 64, 128, 256 and 512. Again, for each of these settings we ran our code for 10 times and took the average to obtain the values shown in the following tables.

In simple #pragma *omp parallel for*, the default scheduling is set to *static* which partitions the for loop iterations to the available cores statically where each core handles the same number of iterations. However, this may not be suitable for sparse and irregular data which is the case for MovieLens data; the number of ratings by a user may differ from that of another user. Some users have rated 10 or 20 movies, whereas others have rated hundreds of movies. That is why static scheduling may lead to an imbalanced distribution of the load.

## 4.1 Using Multicore CPU for Parallelization

In this section, we have implemented several parallelized versions of our code explaining the parallelization technique used in each of them. These versions use the multicore CPU parallelization techniques, which means using the available CPU cores, we try to distribute the process on available cores. The more efficient this process gets done, the more speedup we gain.

**Naive parallelization**: We consider this code as our base-line for the rest of our CPU implementations. In this code, we have implemented a simple parallel *for* loop with static scheduling. It is noteworthy that in this version, we have combined *comp* with *grad-L* to avoid repetitive access to the memory.

In simple #pragma *omp parallel for*, the default scheduling is set to *static* which partitions the for loop iterations to the available cores statically where each core handles the same number of iterations. However, this may not be suitable for sparse and

irregular data which is the case for MovieLens data; the number of ratings by a user may differ from that of another user. Some users have rated 10 or 20 movies, whereas others have rated hundreds of movies. That is why static scheduling may lead to an imbalanced distribution of the load.

Code excerpt 4.1: Static Scheduling

```
#pragma omp parallel
  {
     int myId = omp_get_thread_num();
     totalcosts[myId][0] = 0;
     #pragma omp for schedule(static)
     for (int i = 0; i < NumRows; i++){
        double *myZ1 = Z1 + i*RankFact;
        double *myZ1U = Z1update + i*RankFact;
        memset(myZ1U, 0, sizeof(double)*RankFact);
        for(int ptr = crs_ptrs[i]; ptr < crs_ptrs[i+1]; ptr++) {
           double *myZ2  = Z2 + crs_colids[ptr] * RankFact;
           double curXhat = 0.0;
           for (int k = 0; k < RankFact; k++) {
              curXhat = curXhat + myZ1[k] * myZ2[k];
           }
           double diff = curXhat - crs_values[ptr];
           div_term[ptr] = diff;

           for (int k = 0; k < RankFact; k++) {
             myZ1U[k] += (myZ2[k] * diff)*scfac;
           }
           totalcosts[myId][0] += (diff * diff);
        }
     }
  }

```

**Dynamic Scheduling**: Here, we defined our chunk sizes regarding the number of rows we process. In OpenMP, *static scheduling* has almost no overhead, yet it does not guarantee load balancing when the task sizes vary, which is the case for sparse and irregular data. *Dynamic scheduling* solves the problem by dynamically assigning chunks to threads and it gives a single chunk to a thread at once. If we don't specify a chunk size, each chunk contains a single task (i.e., chunk size = 1) which yields a very good load balancing, yet, also significant overhead since each row will be scheduled dynamically. This is why we observe a significant reduction on the run-time (Fig. 4.1) until chunk size 128-256. But after some point, the scheduling cost is reduced enough, hence, increasing the chunk size deteriorates the balance. In our case, values 128-512 seem reasonable and we chose 256. Figure 4.1 depicts the effect of block size in our code. The numbers in picture 4.1 are the results of running our *Dynamic Scheduling* version of the code with eight threads on MovieLens 1 million data set. As Figure 4.1 shows, being able to choose proper block sizes in *dynamic scheduling* results in better scalability as well as better run-time as we will see later. If we use dynamic scheduling without defined chunk size, not only we won't get any better results, but also because of the huge overhead of dynamic scheduling, it would yield even worse results compared to *static scheduling*.

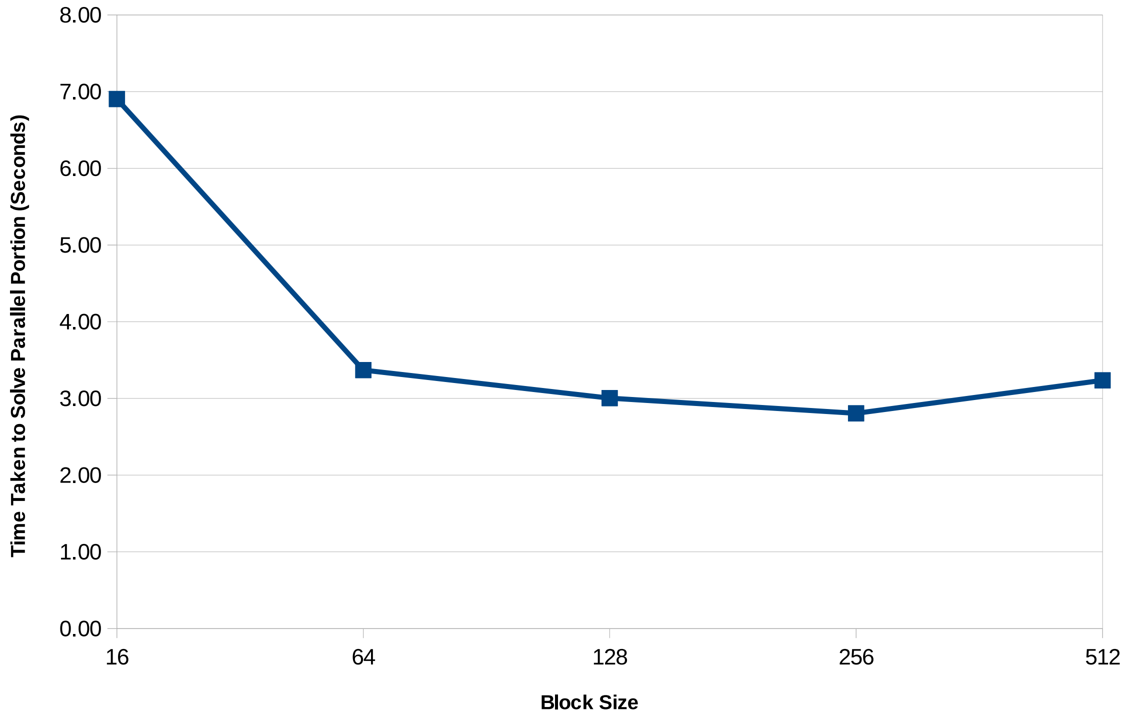


Figure 4.1: The Effect of Chunk Size on Speedup

By comparing Code excerpts 4.1 and 4.2, you can observe the changes made to one of the functions to make it work with *Dynamic Scheduling* policy.

Code excerpt 4.2: Dynamic Scheduling

```
#pragma omp parallel
  {
    int myId = omp_get_thread_num();
    totalcosts[myId][0] = 0;
    #pragma omp for schedule(dynamic, bSize)
    for (int i = 0; i < dataBlockSize; i++) {
      int row   = crs_rowids[i];
      int col   = crs_colids[i];
      double curX = crs_values[i];
      double curXhat = 0;
      for (int k = 0; k < RankFact; k++) {
        curXhat = curXhat +  Z1[RankFact*row+k] * Z2[col * RankFact + k];
      }
      div_term[i]= (curXhat - curX);
      totalcosts[myId][0] += div_term[i] * div_term[i];
    }
  }

```

**Loop Unrolling**: In this version, we implemented *Loop Unrolling* which is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as the space-time trade off. The transformation can be undertaken manually by the programmer or by an optimizing compiler [21]. This way, instead of one operation in each iteration of the loop, we executed several operations.
Code excerpt 4.3 illustrates the way we implemented *Loop Unrolling* in our work.

Code excerpt 4.3: Loop Unrolling

```
#pragma omp for schedule(dynamic, bSize)
    for (int i = 0; i < NumRows; i++){
      const double *myZ1 = Z1 + i*RankFact;
      double *myZ1U = Z1update + i*RankFact;
      memset(myZ1U, 0, sizeof(double)*RankFact);
      int start = crs_ptrs[i];
      int end = crs_ptrs[i+1];
      int ptr;
      for(ptr = start; ptr < end - 3; ptr += 4) {
        const double *myZ2_1  = Z2 + crs_colids[ptr] * RankFact;
        const double *myZ2_2  = Z2 + crs_colids[ptr+1] * RankFact;
        const double *myZ2_3  = Z2 + crs_colids[ptr+2] * RankFact;
        const double *myZ2_4  = Z2 + crs_colids[ptr+3] * RankFact;
        diff_1 = -crs_values[ptr];
        diff_2 = -crs_values[ptr+1];
        diff_3 = -crs_values[ptr+2];
        diff_4 = -crs_values[ptr+3];

        for (int k = 0; k < RankFact; k++) {
          double temp = myZ1[k];
          diff_1 += temp * myZ2_1[k];
          diff_2 += temp * myZ2_2[k];
          diff_3 += temp * myZ2_3[k];
          diff_4 += temp * myZ2_4[k];
        }

        div_term[ptr] = diff_1;
        div_term[ptr + 1] = diff_2;
        div_term[ptr + 2] = diff_3;
        div_term[ptr + 3] = diff_4;

        for (int k = 0; k < RankFact; k++) {
          myZ1U[k] += (myZ2_1[k] * diff_1 +  myZ2_2[k] * diff_2 + myZ2_3[k] * diff_3 +
               myZ2_4[k] * diff_4) * scfac;
        }
        error += diff_1 * diff_1 + diff_2 * diff_2 + diff_3 * diff_3 + diff_4 *diff_4;
      }
      if((end - start) % 4 == 0) {
        for(ptr = end - (end - start) % 4; ptr < end; ptr++) {
          double *myZ2_1  = Z2 + crs_colids[ptr] * RankFact;
          diff_1 =  0.0;
          for (int k = 0; k < RankFact; k++) {
            diff_1 += myZ1[k] * myZ2_1[k];
          }
          diff_1 -= crs_values[ptr];
          div_term[ptr] = diff_1;
          for (int k = 0; k < RankFact; k++) {
            myZ1U[k] += myZ2_1[k] * diff_1 * scfac;
          }
          error += diff_1 * diff_1;
        }
      }
    }
    totalcosts[myId][0] = error;
}

```

## 4.2 Amdahl's Law

To compare our speedup in different versions of the code, we used a measure called *Amdahl's Law.* In computer architecture, *Amdahl's law* gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl [22]. Amdahl's law can be formulated the following way:

$$S_{latency}(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{4.1}$$

where:

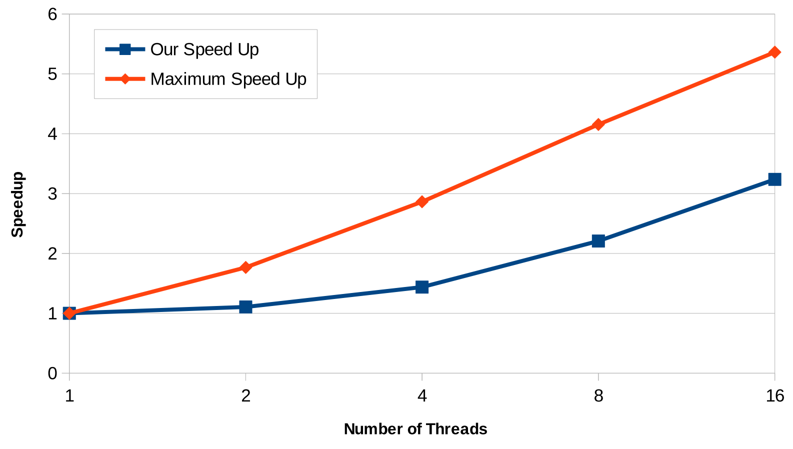
- $S_{latency}$ is the theoretical speedup in latency of the execution of the whole task;
- $s$ is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- $p$ is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. In our case as Table 4.1 shows, considering the total time spent to finish the code and the time spent to finish the part we tried to parallelize, we first measured $s$ and $p$ parameters for each version of the code on different data sets. Afterwards, we added the ideal speedup which basically replaces the number of cores with latency speedup ($s$). In other words, we assumed that if we use two cores, the parallel portion of the code will be executed in half the time. In reality, this never happens due to overhead of the parallelization process, but the closer we get to the ideal value, the better our parallel implementation is.

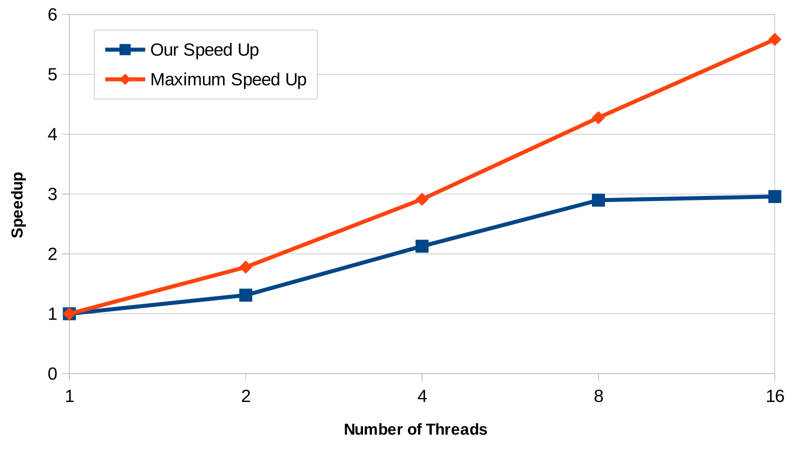
Table 4.1: Amdahl's Law for Naive Version on 10M Data Set. We obtained the ideal speedup by setting s = number of threads.

| Number of Threads | Function Time | Total Time | $p$ | $s$ | Our Speedup | Ideal Speedup (s=#threads) |
|---|---|---|---|---|---|---|
| 1 | 234.099 | 245.112 | 0.955 | 1 | 1 | 1 |
| 2 | 170.865 | 182.027 | 0.955 | 1.370 | 1.347 | 1.914 |
| 4 | 107.460 | 119.034 | 0.955 | 2.178 | 2.068 | 3.524 |
| 8 | 62.617 | 73.813 | 0.955 | 3.738 | 3.328 | 6.085 |
| 16 | 35.651 | 46.755 | 0.955 | 6.566 | 5.252 | 9.558 |

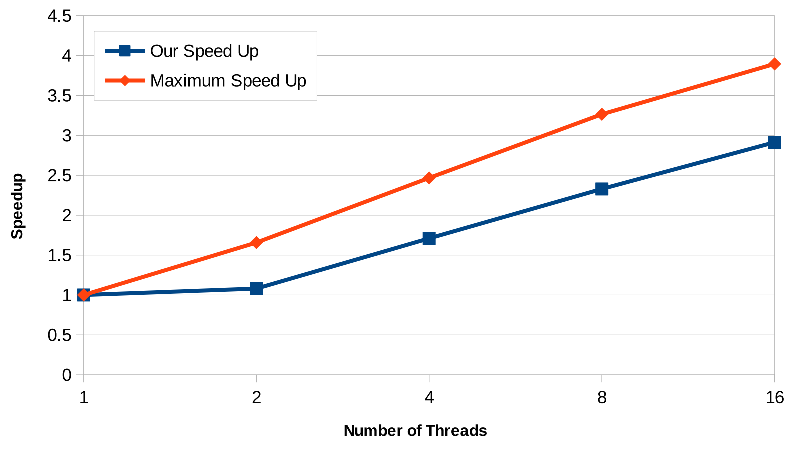
Figures 4.2, 4.3, 4.4 and 4.5 show the speedups in our different versions using Amdahl's law. As these figures show, our latest implementations have better scaling.



(a) Naive-100K

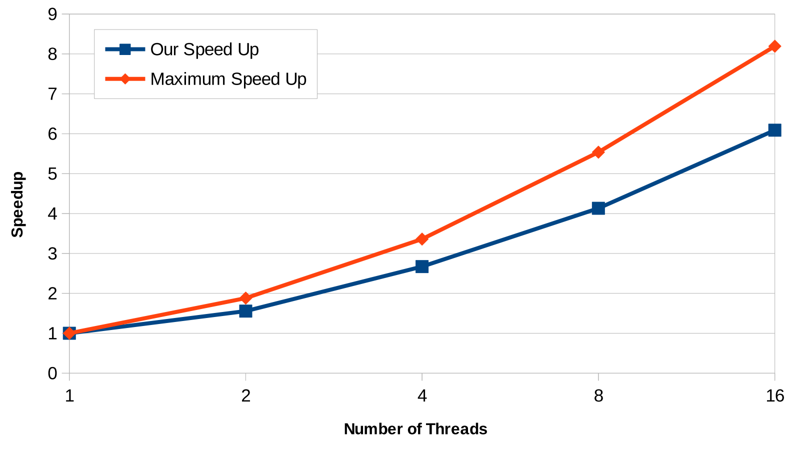


(b) Dynamic Scheduling-100K

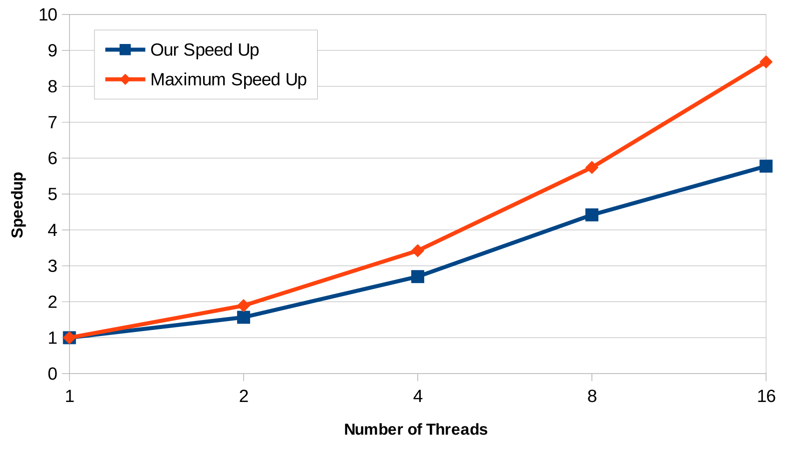


(c) Loop Unrolling-100K

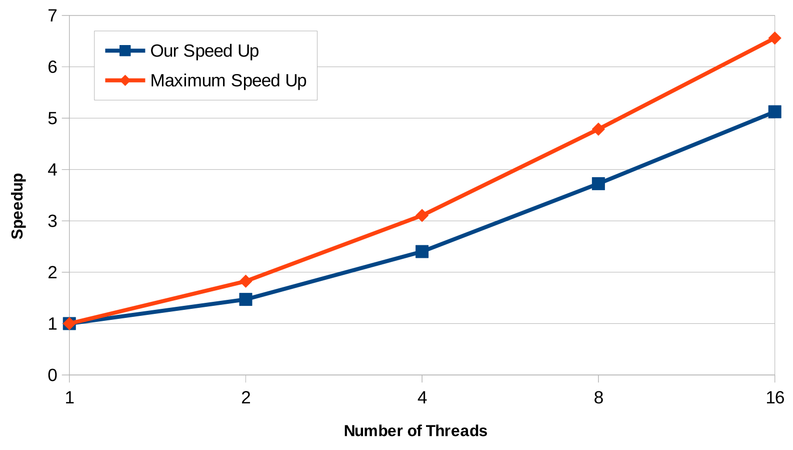
Figure 4.2: Amdahl's Law:CPU implementations on MovieLens 100K



(a) Naive-1M

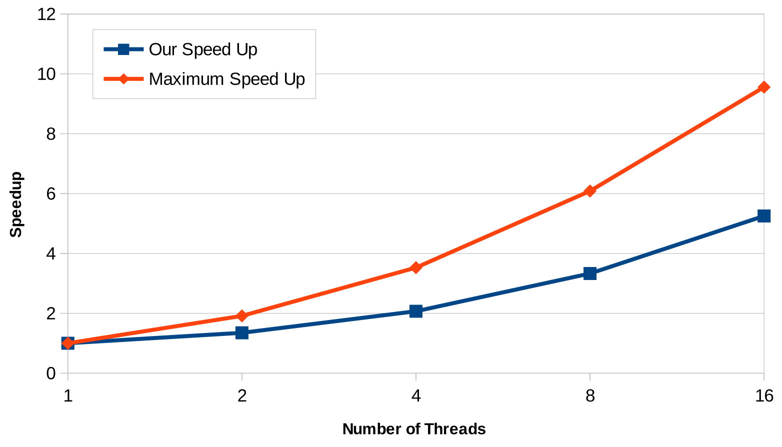


(b) Dynamic Scheduling-1M

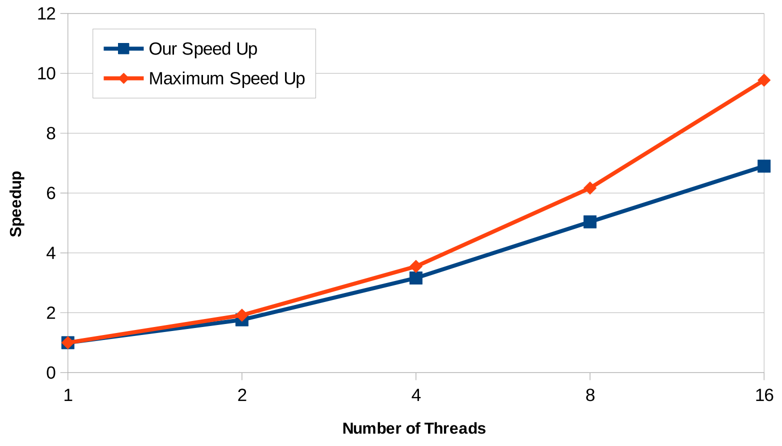


(c) Loop Unrolling-1M

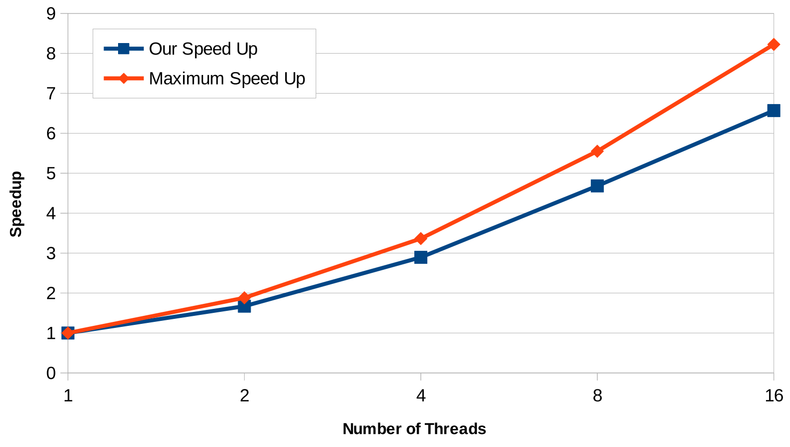
Figure 4.3: Amdahl's Law:CPU implementations on MovieLens 1M



(a) Naive-10M

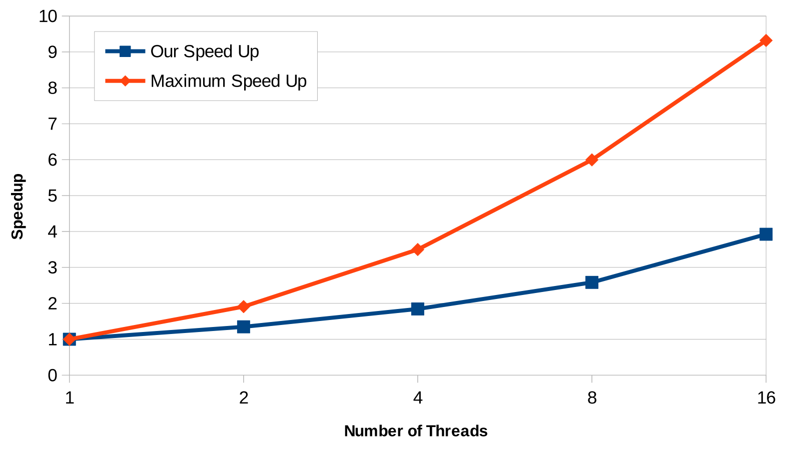


(b) Dynamic Scheduling-10M

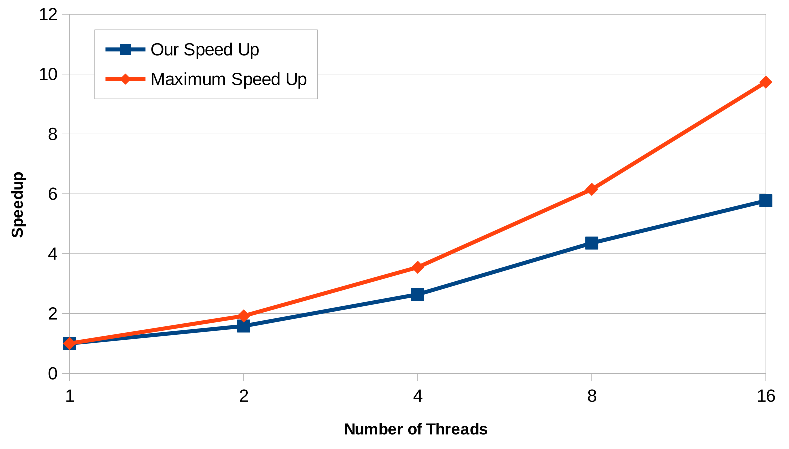


(c) Loop Unrolling-10M

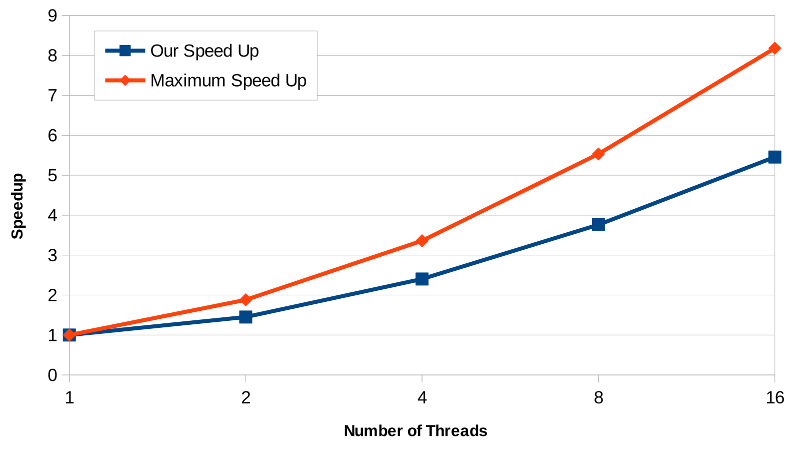
Figure 4.4: Amdahl's Law:CPU implementations on MovieLens 10M



(a) Naive-20M



(b) Dynamic Scheduling-20M

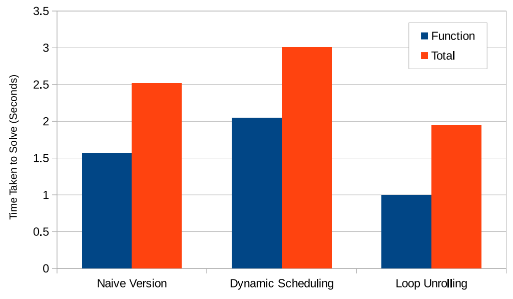


(c) Loop Unrolling-20M

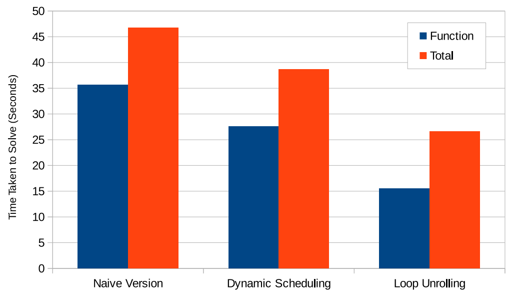
Figure 4.5: Amdahl's Law:CPU implementations on MovieLens 20M

## 4.3 CPU Implementations Time Comparison

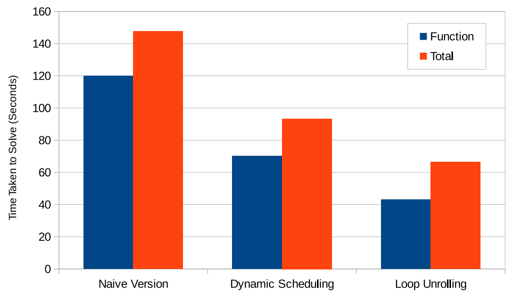
After we have explained how each version is working and what are their differences in terms of the way we have implemented them, let's take a look at the time it takes for each version to finish different versions of MovieLens datasets. As the figure 4.6 shows, our last version of the CPU codes (Loop Unrolling) has the best timing. Moreover, it can be seen that using dynamic scheduling with inappropriate block size would result in the worst performance.



(a) 1M - 16 Threads



(b) 10M - 16 Threads



(c) 20M - 16 Threads

Figure 4.6: CPU Run Times on Different Data-Sets

### 4.4 GPU Parallelization

For GPU parallelism, we have used CUDA/C programming language which is a parallel computing platform and application programming interface (API) model created by NVIDIA. The main benefit of using GPU for parallelization is that we may have thousands of threads (depending on our GPU) running at the same time. GPU parallelism has its own shortcomings, for instance, copying the data from RAM to GPU RAM is expensive. That's why we have to avoid that operation as much as possible to be able to develop an efficient implementation.

As explained in CPU part, we know which parts of our code are the expensive ones, so we focus on them to make the most improvement. To be able to measure our speedup, we have used a naive implementation as our baseline and compared the rest of them to that version. In the following parts of this section, we are going to explain about each GPU code.

**Naive CUDA version**: This version works as our baseline to compare our next versions of the code against it. As mentioned in CPU part, since most of the run time of our code is for our function, we parallelized that part which in turn, consists of other functions. You may find the runtime of this version in Table 4.2.

Table 4.2: GPU Baseline Model Analysis

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 6.24 | 6.66 | 93.68% |
| 1M | 22.88 | 24.55 | 93.20% |
| 10M | 219.28 | 238.94 | 91.77% |
| 20M | 431.46 | 477.04 | 90.45% |

As you may have noticed, compared to our baseline model for CPU, as the dataset size increases, GPU runtime is getting smaller. This can be attributed to the structure of GPU where we have so many cores (thousands) compared to CPU where we have less than 100 cores. Hence, even though CPU might be faster on smaller samples due to its stronger cores, dealing with large datasets benefits more from having a lot of threads.

Code excerpt 4.4, which is one of the functions in our code, better accentuates the similarities and differences between our GPU and CPU implementations and also can be used for further comparisons with other versions of our GPU versions.

Code excerpt 4.4: Naive CUDA version

```
__global__ void computeZ2update(int * ccs_ptrs,int * ccs_rowids,int * ccs_translator,
    double * div_term, double *Z1, double *Z2update, int RankFact, int NumRows, int
    NumCols, float scfac) {
  int myId = blockIdx.x * blockDim.x + threadIdx.x;
  if(myId < NumCols) {
    double *myZ2U = Z2update + myId*RankFact;
    for (int k = 0; k < RankFact; k++) {
      myZ2U[k] = 0;
    }
    for(int ptr = ccs_ptrs[myId]; ptr < ccs_ptrs[myId+1]; ptr++) {
      double *myZ1 = Z1 + ccs_rowids[ptr] * RankFact;
      double curVal = div_term[ccs_translator[ptr]];
      for (int k = 0; k < RankFact; k++) {
        myZ2U[k] += myZ1[k] * curVal * scfac;
      }
    }
  }
}

```

**Partially Fine-Grained version**: *Fine-Graining* in the field of parallel programming means that instead of assigning large tasks to threads, the granularity of the tasks assigned to threads are increased. In our function *computeZ2update*, a single thread was assigned to update one row of our Data matrix in our baseline model, which means it was *Coarse-Grained*. In the fine-grained version we assigned one thread to each non-zero element to finish this job. As Table 4.3 shows, by doing this, our runtime for all the versions of our datasets have been decreased. The reason is, while processing a row, for each non-zero in that row, we go through the main loop once. But since we are running this code in parallel, In each execution of the code, several rows are read to be executed at once. Now if some threads finish their job faster, they have to wait for other threads to finish their task, and then read another bunch of rows. Since the number of non-zeros in each row is not the same and it may vary a lot, those threads with few non-zeros (e.g. 20) must wait for the longest thread which may have many times more non-zeros. By *Fine-Graining* our function, all the threads will go through the same amount of process and they finish almost at the same time and then, they read the next group of non-zeros.

Table 4.3: Partially Fine-Grained GPU Version Analysis

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 4.25 | 4.66 | 91.16% |
| 1M | 18.91 | 20.56 | 91.97% |
| 10M | 191.41 | 210.12 | 91.10% |
| 20M | 369.04 | 420.65 | 87.73% |

To observe the difference in this implementation compared to the previous version, you may compare Code Excerpts 4.4 and 4.5

Code excerpt 4.5: Partially Fine-Grained version

```
__global__ void computeZ2update(int * ccs_ptrs,int * ccs_rowids,int * ccs_translator,
    double * div_term, double *Z1, double *Z2update, int RankFact, int NumCols, float
    scfac) {
  int myId = blockIdx.x * blockDim.x + threadIdx.x;
  int myColId = myId / RankFact;
  int localId = myId % RankFact;
  if(myId < RankFact * NumCols) {
   Z2update[myId] = 0;
   for(int ptr = ccs_ptrs[myColId]; ptr < ccs_ptrs[myColId+1]; ptr++) {
     Z2update[myId] += Z1[ccs_rowids[ptr] * RankFact + localId] * div_term[
         ccs_translator[ptr]];
   }
   Z2update[myId] = Z2update[myId] * scfac;
  }
}
```

**Fully Fine-Grained version**: Since we saw the effect of *Fine-Graining* in version 1, here we applied the same technique for computeZ1update. But to be able to do that, first we have to break the function which calculates the function value and Z1update into two separate functions. But in this case there was a trade-off between fine-graining the function and separating it into two functions because a portion of the calculations done in both of those parts are common. This is what happened in this version and as Table 4.4 shows, our runtime decreased on all the datasets, which means that this fine-graining has more impact on the runtime.

Table 4.4: Fully Fine-Grained GPU Version Analysis

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 2.49 | 2.91 | 85.73% |
| 1M | 11.98 | 13.66 | 87.67% |
| 10M | 113.91 | 132.77 | 85.79% |
| 20M | 223.42 | 276.57 | 80.78% |

**Shared Memory Reduction**: While computing the function value in each iteration, we measure the distance between non-zero elements in our original data matrix, and corresponding elements in the matrix we made. This way, we end up with an array which stores one number per each non-zero element. Afterwards, we add up the elements in that array and come up with one number which is our function value. The process of summing up the elements in an array in called *reduction.* In CPU implementations, we don't have any difficulties dealing with this operation and usually perform it in one loop which iteratively adds the elements to one variable. But this is not the most efficient way for reduction in GPU. In this version of the code, we have implemented a more efficient version of the reduction function. Table 4.5 shows the speed-up we gain by doing so. There is a crucial point one must notice while comparing the results of this version against previous ones which is the fact that we gain more and more speed-up as the dataset grows in size. The reason is reduction operation is exactly where GPU parallelization excels compared to CPU version. It is composed of a huge amount of repetitive and simple tasks, so we don't need strong cores, but instead we need many cores which can accomplish the simple tasks at the same time.

Table 4.5: Shared Memory Implementation Analysis

| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 0.55 | 0.96 | 57.26% |
| 1M | 3.17 | 4.84 | 65.45% |
| 10M | 30.24 | 49.56 | 61.03% |
| 20M | 59.52 | 101.61 | 58.57% |

**Other Compiler Optimizations**: Here, we implemented some compiler optimization tricks to further improve the performance of the code. Let's take a look at the functions and the way we state the variables in them. Code excerpts 4.6 and 4.7 show the difference in the last version of our code compared to the previous one.

Code excerpt 4.6: Function Evaluation in Compiler Optimized Version

```
__global__ void computeDivEuc(int* __restrict__ crs_ptrs, const int* __restrict__ crs_rowids, const int * __restrict__ crs_colids, const double* __restrict__ crs_values, double* __restrict__ div_term, double* Z1, double* __restrict__ Z2, int RankFact, int NumRows,int NumElements, double* totalcosts) {
  int myId = blockIdx.x * blockDim.x + threadIdx.x;
  if(myId < NumElements) {
    double myCost = 0;
    double *myZ1 = Z1 + crs_rowids[myId] * RankFact;
    double *myZ2  = Z2 + crs_colids[myId] * RankFact;
    double curXhat = 0.0;
    for (int k = 0; k < RankFact; k++) {
      curXhat = curXhat + myZ1[k] * myZ2[k];
    }
    double diff = curXhat - crs_values[myId];
    div_term[myId] = diff;
    myCost += diff * diff;
    totalcosts[myId] = myCost;
  }
}
```

Code excerpt 4.7: Function Evaluation in Shared Memory Reduction Version

```
__global__ void computeDivEuc(int* crs_ptrs, int* crs_rowids, int* crs_colids, double* crs_values, double* div_term, double* Z1, double *Z1update, double * Z2, int RankFact, int NumRows, int NumElements, double* totalcosts) {
  int myId = blockIdx.x * blockDim.x + threadIdx.x;
  if(myId < NumElements) {
    double myCost = 0;
    double *myZ1 = Z1 + crs_rowids[myId] * RankFact;
    double *myZ2  = Z2 + crs_colids[myId] * RankFact;
    double curXhat = 0.0;
    for (int k = 0; k < RankFact; k++) {
      curXhat = curXhat + myZ1[k] * myZ2[k];
    }
    double diff = curXhat - crs_values[myId];
    div_term[myId] = diff;
    myCost += diff * diff;
    totalcosts[myId] = myCost;
  }
}
```

Code excerpts 4.6 and 4.7 represent the C++ function which evaluates the value of the function. Code 4.7 depicts the way we wrote the function in our previous version (shared memory reduction version) and code 4.6 shows our last implementation. As you might notice, the body of these two functions are not different. The difference can be found in the way we announced the type of the variables. Before proceeding to describe these differences we explain *pointer aliasing*.

Two pointers *alias* if the memory to which they point overlaps. When a compiler can't determine whether pointers alias, it has to assume that they do. The following simple function shows why this is potentially harmful to performance:

Code excerpt 4.8: Pointer Aliasing

```
void example1(float *a, float *b, float *c, int i) {
  a[i] = a[i] + c[i];
  b[i] = b[i] + c[i];
}
```

At first glance it might seem that this function needs to perform three load operations from memory: one for a[i], one for b[i] and one for c[i]. This is incorrect because it assumes that c[i] can be reused once it is loaded. Consider the case where a and c point to the same address. In this case the first line modifies the value c[i] when writing to a[i]. Therefore the compiler must generate code to reload c[i] on the second line, in case it has been modified.

Because the compiler must conservatively assume the pointers alias, it will compile the above code inefficiently, even if the programmer knows that the pointers never alias.Fortunately almost all C/C++ compilers offer a way for the programmer to give the compiler information about pointer aliasing.

By giving a pointer the *restrict* property, the programmer is promising the compiler that any data written to through that pointer is not read by any other pointer with the restrict property. In other words, the compiler doesn't have to worry that a write to a restrict pointer will cause a value read from another restrict pointer to change. This greatly helps the compiler optimize code.

There is, however, one potential GPU-specific benefit to *__restrict__*. Recent NVIDIA GPUs have a cache designed for read-only data which can, for some codes, improve data access performance. This cache can only be used for data that is read-only for the lifetime of the kernel. To use the read-only data cache, the compiler must determine that data is never written. Due to potential aliasing, the compiler can't be sure a pointer references read-only data unless the pointer is marked with both *const* and *__restrict__*.

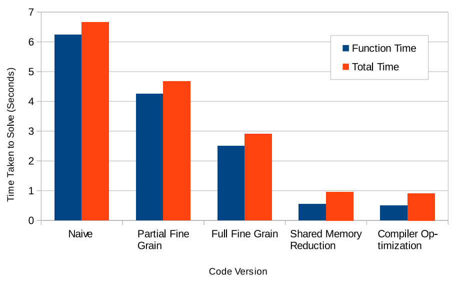
Based on these compiler optimizations, we found the applicable variables and used these techniques on them. Table 4.6 shows the run-time of this version on different data sets and the percentage of the time taken by the parallel portion of the code.

Table 4.6: Compiler Optimization Version Analysis

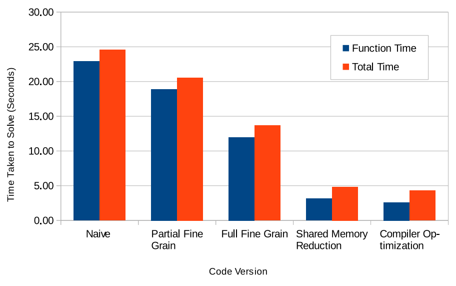
| Data-set | Function Time | Total Time | Percentage |
|---|---|---|---|
| 100K | 0.49 | 0.91 | 54.32% |
| 1M | 2.60 | 4.28 | 60.73% |
| 10M | 25.70 | 44.76 | 57.40% |
| 20M | 51.77 | 92.51 | 55.97% |

## 4.5 GPU Codes Comparison and Analysis

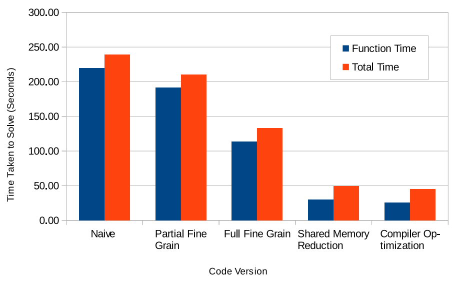
In this section, we will compare different GPU versions in terms of their speed-up and also will dig deeper to see which components of our codes take the most time.



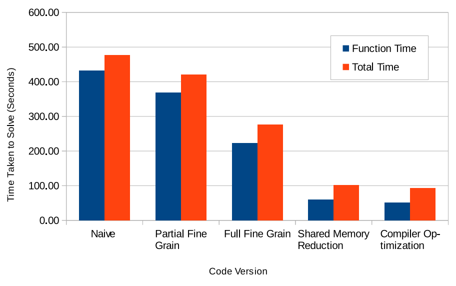
(a) 100K



(b) 1M



(c) 10M



(d) 20M

Figure 4.7: Comparison of Different GPU Versions

As Figure 4.7 suggests, with each implementation, the speed-up has been increased. There are a few notes to be considered about these figures. First, for each version you can see 2 bars; Function and Total. Function bar refers to the function which we have parallelized. Second, the difference between Function and Total bars are related to the preprocessing and sequential parts of our code. As we will say later, in our future works, this gap will be minimized even further. Finally, as figures suggest, some versions of our code have better speed-ups while being implemented on a specific datasets. This is related to the technique we have implemented in each version. Some of them are more beneficial while being run on smaller datasets and others are better when run on larger datasets.

Next, we will analyse the time spent on each version in terms of its components.

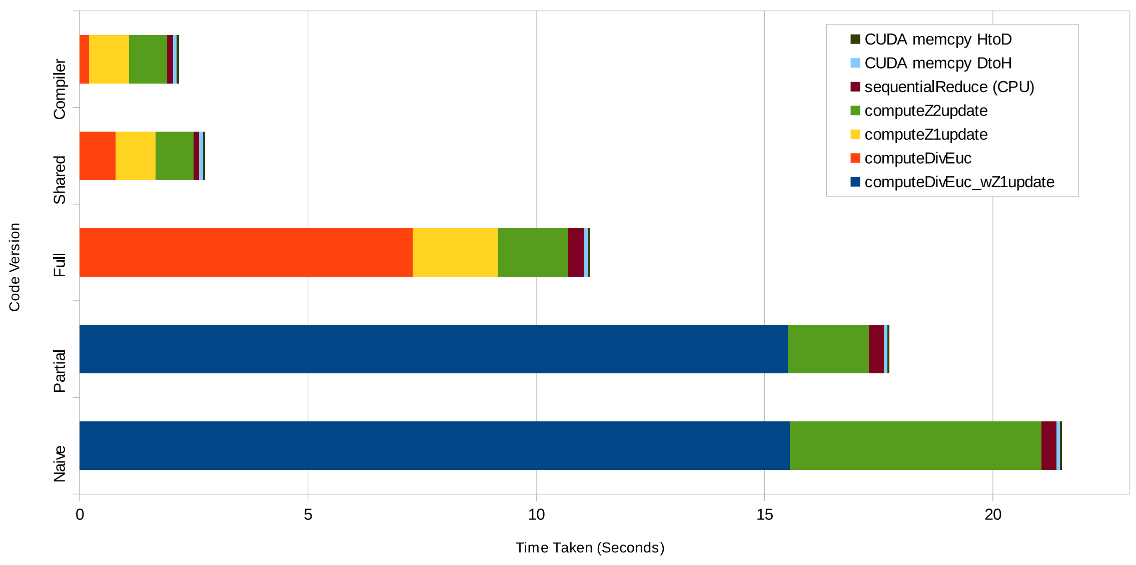
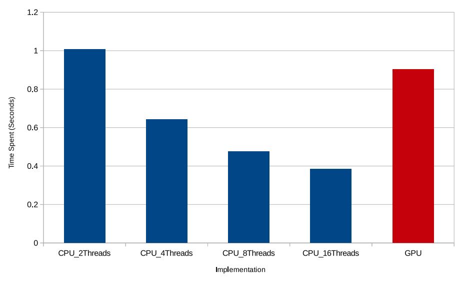


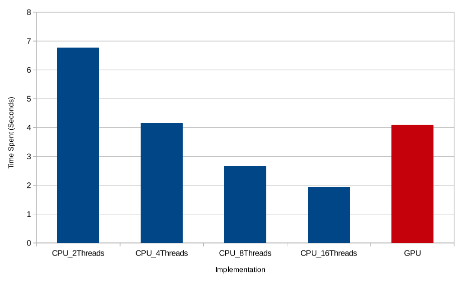
Figure 4.8: Analysis of the Time Components

## 4.6 CPU vs. GPU comparisons

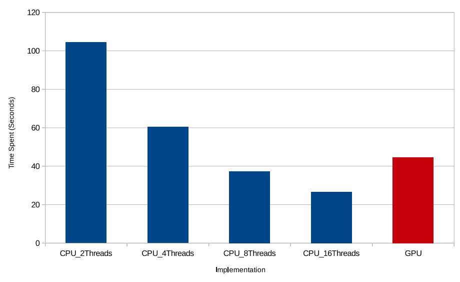
In this section we compare our best CPU implementation against our best GPU version to see how do they compare against each other. It is noteworthy that we only included our fastest version of CPU code, and we ran different number of threads on that version. On the GPU side, again we picked the fastest implementation. Figure 4.9 represents the results for these experiments.



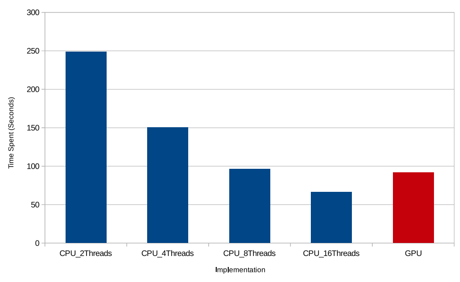
(a) 100K



(b) 1M



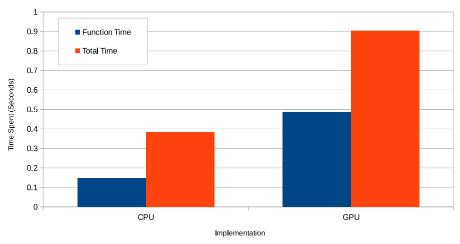
(c) 10M



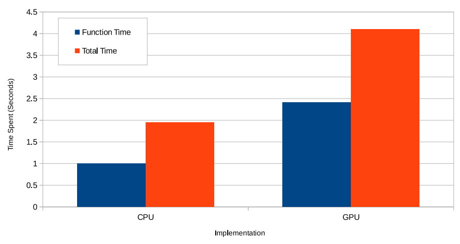
(d) 20M

Figure 4.9: CPU vs. GPU comparison considering total time

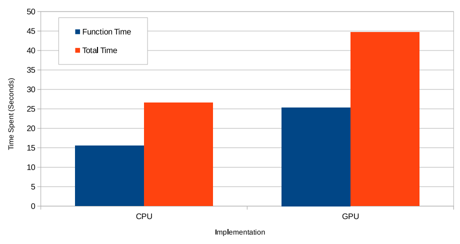
One thing which is quick to notice is that our GPU implementation's performance gets better speedups as the size of the data-set increases. For MovieLens 100K, GPU speed-up is almost close to the CPU implementation with 4 cores. But for MovieLens 20M, it is on par with CPU implementation with 16 cores. Moreover, using GPU has its own overhead which is more than CPU implementation, that is why if we observe the parallel portion of the code, rather than total time, we may see even more advantage for GPU version. This is largely due to the perfect optimization of this algorithm, such that it can benefit from having more threads. In Figure 4.10 you may see that our parallelized portion of the code is in fact faster than what appears in Figure 4.9.



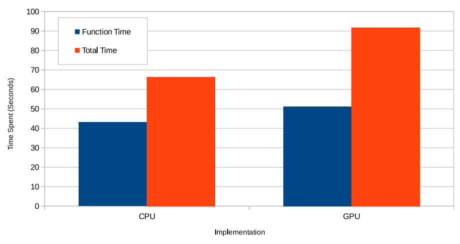
(a) 100K



(b) 1M



(c) 10M



(d) 20M

Figure 4.10: CPU (16 threads) vs. GPU comparison considering parallel function and total time

CHAPTER 5

# CONCLUSION AND FUTURE WORK

We have developed a framework for parallelization of optimization methods for solving matrix factorization problems. In this process, we have implemented several versions of parallel codes on CPU and GPU, while we explained the tricks used on each version. We tested our implementations on all sizes of MovieLens data set and showed the significant speedups gained.

This thesis can be used as a reference for those who already have their applicable optimization algorithm and want to improve its execution time tremendously. Moreover, for the developers who want to devise a fast and accurate software for the usages such as recommendation systems, this work provides the core algorithm as well as its implementation. Since this thesis combines optimization techniques with parallel programming tools and tricks, for any academician who is interested in interdisciplinary works related to ours, this work will provide a useful collection of required tools.

There are still many areas for further improving our work. Some of the most promising areas which would most likely result in advancements in our results are listed below:

- We started by improving a function which takes up almost 95% of our sequential code's execution time. But as Table 4.6 shows, at the end of our experiments we got to the point where that function was taking less than 60% of our time. This means that the parts which were formerly *cheap* parts of our code, now are getting *expensive* and working on them would be meaningful since it would further increase the performance of our code. According to these facts, working on the non-parallel portion of our code and making it parallelized would be an area of advancement.

- We have mentioned many parallel programming tools and observed how they would help us gain better speedups. But these are not the only available tools which can be used. Proper utilization of *Shared Memory* can increase the performance of our code and in our work, we exploited shared memory just for the reduction. *Tiling* is another technique which can be incorporated to further increase the performance. Moreover, Using SIMD instructions for CPU versions, since in our case the number of registers is small.

- Here, we used an optimization technique which has not been used for matrix factorization before and we observed how this method outperforms some of the most famous methods. Novel optimization techniques, may perform even better under this framework. Incorporating new methods and examining their performance would be another field where advancement would be achieved.

In this thesis we solved a matrix factorization problem. This does not necessarily mean that other problems cannot be solved via our implementations. It is also possible that our work would gain better results in solving other problems, hence, implementing our codes on different problems would be an area where might lead to exceptional results.

# Bibliography

[1] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*, volume 2. Siam, 2000.

[2] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. Parallel quasi-Newton methods for unconstrained optimization. *Mathematical Programming*, 42(1-3):273–306, 1988.

[3] S. G. Nash and A. Sofer. Block truncated-newton methods for parallel optimization. *Mathematical Programming*, 45(1-3):529–546, 1989.

[4] S. Rendle. Factorization machines with libfm. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(3):57, 2012.

[5] R. Kannan, M. Ishteva, and H. Park. Bounded matrix factorization for recommender system. *Knowledge and Information Systems*, 39(3):491–511, 2014.

[6] H. Wen, G. Ding, C. Liu, and J. Wang. Matrix factorization meets cosine similarity: Addressing sparsity problem in collaborative filtering recommender system. In *Web Technologies and Applications*, pages 306–317. Springer, 2014.

[7] W. Fang. Research on recommendation algorithm based on unified model with explicit and latent factors. *Journal of Chemical and Pharmaceutical Research*, 6(6):1303–1314, 2014.

[8] Y. Zhuang, W. S. Chin, Y. C. Juan, and C. J. Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

[9] H. F. Yu, C. J. Hsieh, I Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 765–774. IEEE, 2012.

[10] W. S. Chin, Y. Zhuang, Y. C. Juan, and C. J. Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(1):2, 2015.

[11] R. Kannan, G. Ballard, and H. Park. A high-performance parallel algorithm for nonnegative matrix factorization. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 9. ACM, 2016.

[12] J. Oh, W. S. Han, H. Yu, and X. Jiang. Fast and robust parallel sgd matrix factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 865–874. ACM, 2015.

[13] A. Srivastava. *Modeling ordinal data for recommendation system.* PhD thesis, University of British Columbia, 2014.

[14] X. Luo, Y. Xia, and Q. Zhu. Applying the learning rate adaptation to the matrix factorization based collaborative filtering. *Knowledge-Based Systems*, 37:154–164, 2013.

[15] F. Oztoprak. *Parallel Algorithms for Nonlinear Optimization.* PhD thesis, Sabancı University, 2011.

[16] R. Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the 6th conference on Natural Language Learning-Volume 20*, pages 1–7. Association for Computational Linguistics, 2002.

[17] G. Andrew and J. Gao. Scalable training of l 1-regularized log-linear models. In *Proceedings of the 24th international conference on Machine Learning*, pages 33–40. ACM, 2007.

[18] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[19] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1):1–14, 1989.

[20] E. F. DAzevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science*, pages 99–106. Springer, 2005.

[21] W. P. Petersen and P. Arbenz. *Introduction to Parallel Computing.* Oxford University Press, 2004.

[22] D. P. Rodgers. Improvements in multiprocessor system design. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 225–231. IEEE Computer Society Press,Los Alamitos, 1985.