UiT THE ARCTIC UNIVERSITY OF NORWAY

The Faculty of Science and Technology
Department of Computer Science

Parallelization of the Alternating-Least-Squares Algorithm With Weighted Regularization for Efficient GPU Execution in Recommender Systems

—

Michael Kampffmeyer

INF-3981 — Master's Thesis in Computer Science — June 2015

Abstract

Collaborative filtering recommender systems have become essential to many Internet services, providing, for instance, book recommendations at Amazon's online e-commerce service, music recommendation in Spotify and movie recommendation in Netflix.

Matrix factorization and Restricted Boltzmann Machines (RBMs) are two popular methods for implementing recommender systems, both providing superior accuracy over common neighborhood models. Both methods also shift much of the computation from the prediction phase to the model training phase, which enables fast predictions once the model has been trained.

This thesis suggests a novel approach for performing matrix factorization using the Alternating-Least-Squares with Weighted-λ-Regularization (ALS-WR) algorithm on CUDA (ALS-CUDA). The algorithm is implemented and evaluated in the context of recommender systems by comparing it to other commonly used approaches. These include an RBM and a stochastic gradient descent (SGD) approach.

Our evaluation shows that significant speedups can be achieved by using CUDA and GPUs for training recommender systems. The ALS-CUDA algorithm implemented in this thesis provided speedup factors of up to 175.4 over the sequential CPU ALS implementation and scales linearly with the number of CUDA threads assigned to it until the GPUs shared memory has been saturated. Comparing the performance of the ALS-CUDA algorithm to CUDA implementations of the SGD and the RBM algorithms shows that the ALS-CUDA algorithm outperformed the RBM. For a sparse dataset, results indicate that the ALS-CUDA algorithm performs slightly worse than the SGD implementation, while for a dense dataset, ALS-CUDA outperforms the SGD. However, generally the advantage of the ALS-CUDA algorithm does not necessarily lie in its speed, but also in the fact that it requires fewer parameters than the SGD. It therefore represents a viable option when some speed can be traded off for algorithmic stability, or when the dataset is dense.

Acknowledgements

Above all, I would like to express the deepest appreciation to my supervisor Håvard D. Johansen for his guidance and support, and for making time for me during his sabbatical leave. I would also like to thank my co-supervisor Robert Jenssen and Mike Taylor for many inspiring discussions during the course of this thesis.

On a personal note, I wish to thank Erlend Graff, Vegard Sandengen and Kristian Elsebø—the other iAD Master students—for good discussions and fun coffee breaks. Further, many thanks to Jan-Ove Karlberg, Einar Holsbø and Bjørn Fjukstad for proofreading this thesis.

Michael Kampffmeyer
Tromsø, June 2015

Contents

List of Figures

List of Tables

List of Abbreviations

ALS	Alternating Least Squares
ALS-CUDA	Alternating-Least-Squares with Weighted-λ-Regularization on CUDA
ALS-WR	Alternating-Least-Squares with Weighted-λ-Regularization
CF	Collaborative Filtering
COO	Coordinate List
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DSGD	distributed stochastic gradient descent
GPC	Graphics Processing Cluster
GPU	graphics processing unit
IF	information filtering
ILP	instruction-level parallelism
MAE	mean absolute error
RBM	Restricted Boltzmann Machine
RBM-CF	Restricted Boltzmann Machine for Collaborative Filtering

SGD stochastic gradient descent

SIMT single instruction, multiple threads

SM Streaming Multiprocessors

SMX Kepler Generation Streaming Multiprocessors

SP Scalar Processors

SSGD stratified stochastic gradient descent

SVD Singular Value Decomposition

List of Code Listings

List of Algorithms

/1 Introduction

Many Internet companies, like Ebay,[1] Amazon,[2] and Netflix,[3] are increasingly dependent on their ability to distill the interests and preferences of their customers from large datasets of historic user behavior [31]. This has led to increasing demands for information filtering (IF) systems that can process large amounts of data to produce personalized recommendations with both high precision and high recall.

IF systems have been used since the early 1990s, when manual filtering systems such as Tapestry [18] were introduced. Shortly thereafter, another class of IF systems, Collaborative Filtering (CF) recommender systems, emerged. Unlike information retrieval systems, which use a static content base to serve a dynamic information need (e.g., a search engine using a static index to serve dynamic queries), recommender systems assume a dynamic content base and a relatively static information need (e.g., music taste). Early automated CF systems include the GroupLense [45] and Ringo [51] projects.

While personalized and content-based systems rely on matching individual user profiles to items, CF also uses community data to provide recommendations. In recent years, recommender systems have moved into the center of our living rooms as part of services for streaming of movies and music, such as Netflix

1. ebay.com
2. amazon.com
3. netflix.com

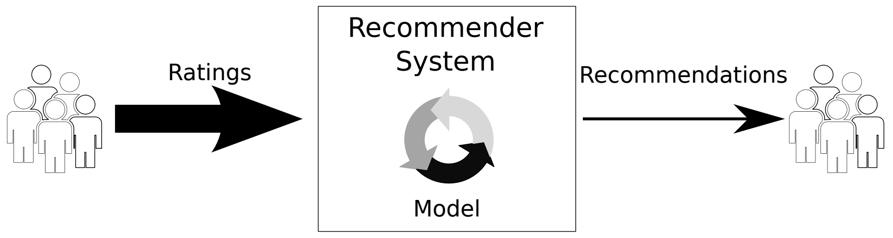

Figure 1.1: Concept of a recommender system.

and Spotify.[4]

In a recommender system users rate certain items (e.g., movies for Netflix). The recommender system uses these ratings in combination with various machine learning techniques to compute a model that, provided a users rating history, computes recommendations. One technique, for example, maps each user and item to the same feature space and provides recommendations based on the distance between users and items in said feature space. Users receive these recommendations and continue to provide additional ratings, thereby causing the model to be updated and improving the recommendations further. As the number of users, items, and rated items increases over the lifetime of the system, the recommendation stage will have to be able to adapt to updates in order to provide the user with up-to-date recommendations. Figure 1.1 illustrates the general concept of a recommender system. The input to a recommender system is a stream of large amounts of data (ratings), which needs to be filtered to provide users with the relevant data, the best recommendations. The filtering rules are dependent on the ratings and will continuously change due to new input. The users providing the ratings can be, but are not necessarily the same users that receive the recommendations, as CF systems utilizes community data to provide recommendations.

1.1 Parallelization Using GPUs

Matrix factorization is a machine learning technique commonly used in CF recommender systems, which often uses a stochastic gradient descent (SGD) approach. The popularity of matrix factorization recommender systems has risen in recent years—especially due to their performance in the Netflix Prize, in which they were an essential part of the winning entry [32]. Matrix factorization recommender systems are now part of, among others, the Netflix recommendation system [33]. Generally, they are considered one of the best

4. spotify.com

standalone approaches in CF, but are in practice often combined with other models to achieve even better performance [37].

Another method that also emerged as part of the Netflix Prize competition is the Restricted Boltzmann Machine (RBM) approach to CF [46]. RBMs have been shown to provide rating accuracies similar to matrix factorization recommender systems, but have been observed to find different patterns in the data [46]. This observation led to highly accurate hybrid systems, where both RBMs and matrix factorization techniques are combined [32].

To cope with the rapid increase in the number of available ratings and the constant stream of new, additional, ratings, it is essential to consider parallel approaches to the problem of recommender systems. Many techniques have been proposed to solve this problem, as the accuracy of recommender systems highly relies on the number of ratings used [27]. Faster, parallel execution will allow the combination of different algorithms, whose results can be combined to achieve better results. Various approaches have been suggested to parallelize the recommendation task, both on distributed and on shared memory systems [44, 55, 60, 65].

Besides accuracy and computational performance, the number of training parameters is another important factor that needs to be considered when choosing a machine learning technique for recommender systems, because many of these techniques require parameters to be specified prior to execution. For simpler configuration of the recommender system, it is desirable that the algorithms have as few training parameters as possible.

This thesis focuses on the middle part of Figure 1.1—the recommender system—and investigates the effects of parallelizing it using NVIDIA's Compute Unified Device Architecture (CUDA) framework for graphics processing units (GPUs).

The computational capability of GPUs has increased significantly in recent years compared to Central Processing Units (CPUs), as illustrated in Figure 1.2, making them an important part of high-performance computing. This has led to the integration of GPUs in a large fraction of the top supercomputers [38]. Even though potentially huge speedups can be achieved, not all problems are suited to parallelization on GPUs.

The task of training recommender system models is a computational expensive task and parallelizing them efficiently on GPUs requires an understanding of their properties, such as memory layout and accesses, and how these translate to the GPU architecture. GPUs have previously found application in CF recommender systems, like those approximating the Singular Value Decomposition (SVD) [14], the RBM [6], and various user-based and clustering

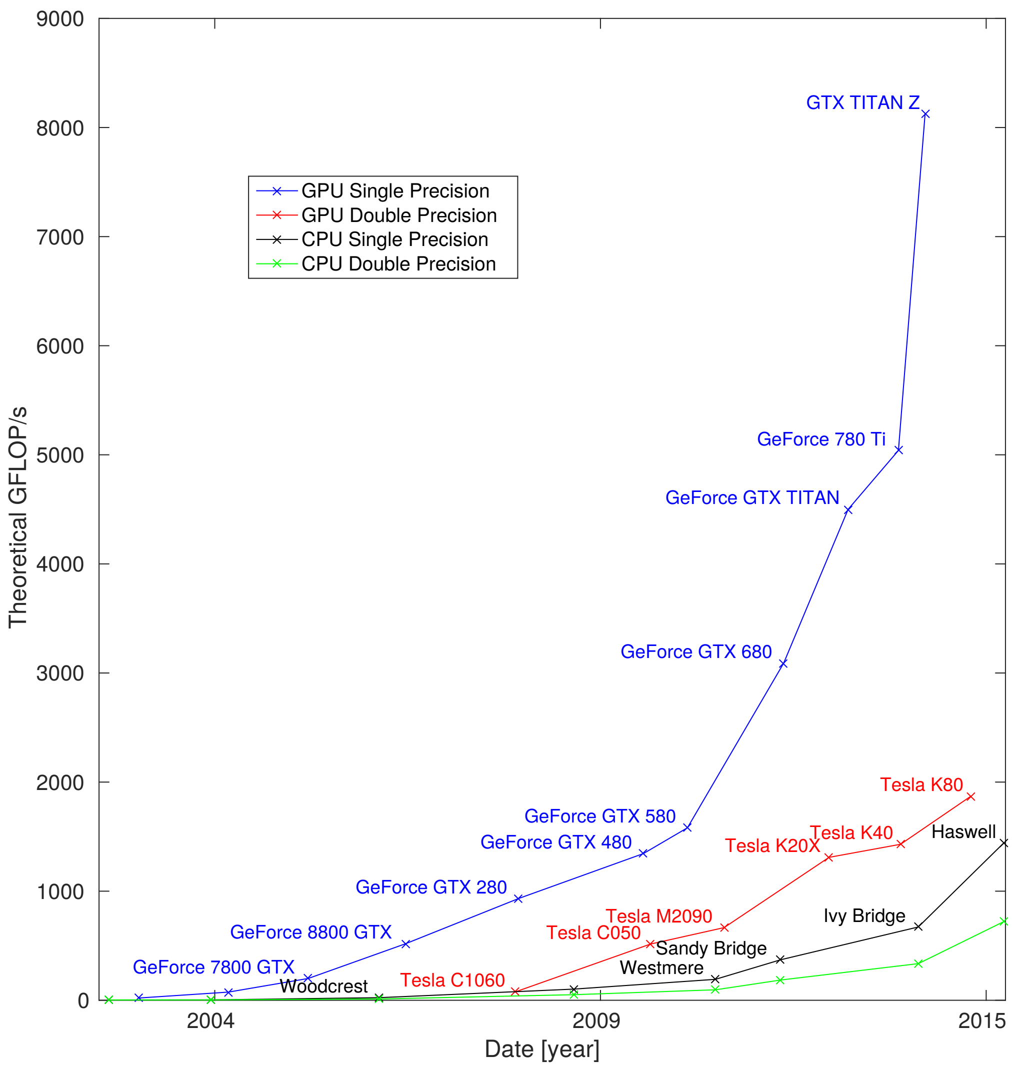

Figure 1.2: Theoretical floating point operations per second (FLOP/s) for various NVIDIA GPUs and Intel CPUs. See Appendix A for more information.

models [29, 57, 62].

1.2 Problem Definition

In this thesis we consider the problem of parallelizing recommender systems on GPUs using the Alternating-Least-Squares with Weighted-λ-Regularization (ALS-WR) algorithm, which has previously been shown to have great efficiency potential on CPUs [37]. The Alternating Least Squares (ALS) algorithm has also the advantage over the SGD and RBM algorithms that it requires fewer parameters to be specified. Although the partial computations of the ALS are inherently parallel, the operations involved are generally quite complex and because of this they are more time consuming to run on the lower clock frequency of the GPU. The ALS-WR algorithm has, to the author's knowledge, not previously been fully implemented using GPUs.

Our thesis is that:

The ALS-WR algorithm can be parallelized efficiently on GPUs by careful orchestration of the GPU single instruction, multiple threads (SIMT) architecture and memory layout.

To support our thesis, we study two related algorithms: SGD and RBM. The SGD algorithm is similarly to the ALS used in matrix factorization approaches to recommender systems, while RBMs are used in model-based systems. We evaluate our method by comparing it to the SGD and RBM approaches, which have been effectively implemented using CUDA.

1.3 Contributions

The contributions of this work are:

- A description of the main approaches to CF.
- The design and implementation of a ALS-WR matrix factorization algorithm on CUDA. Our main focus is on recommender systems, but matrix factorization is also applicable to other contexts, such as computer vision and document clustering.
- An experimental evaluation of the ALS-WR on CUDA by comparing it to CUDA implementations of the SGD and RBM.

1.4 Methodology

The discipline of computer science can, according to the final report of the ACM Task Force on the Core of Computer Science [7], be divided into three major fundamental paradigms: *theory*, *abstraction*, and *design*.

Theory is based on the idea that a problem description is developed and a hypothesis of the relationships within the problem is made. The defined relationships are then proven or disproved, and the results are analyzed.

Abstraction is the second paradigm and describes an experimental scientific method. Similar to the *theory* paradigm, *abstraction* starts off by developing a hypothesis. However, the hypothesis is, unlike in the theory scenario, not proven mathematically, but instead challenged by the use of experiments.

Design has its roots in engineering, where system requirements are defined, and systems are designed, implemented, and tested. The various stages in all three paradigms are usually performed multiple times before completion in an iterative fashion.

As part of this thesis, certain aspects of all three paradigms have been encountered. In the initial stages of the work much focus was put on understanding the existing foundation and theory of the field. Existing related work was implemented to analyze their advantages and drawbacks and based on this foundation, we designed and implemented our own system. The thesis was then experimentally analyzed based on the design and our algorithm was compared to other existing systems.

1.5 Context

This thesis is written as part of the EONS (Efficient Execution of Large Workloads on Elastic Heterogeneous Resources) project at the University of Tromsø—The Arctic University of Norway. The centers investigates parallel programming and parallel processing in the context of future distributed large-scale heterogeneous systems and aims to develop a programming model for big-data workloads, using a combination of compiler, operating system, and high-level scheduler.

1.6 Structure of the Thesis

The thesis is structured in 7 chapters including the introduction.

Chapter 2 provides an introduction to recommender systems and is divided into four different parts. The first part provides the reader with a general overview over the three main approaches to CF recommender systems. The second part focuses on the matrix factorization approach to recommender systems. It introduces the theory of SGD and ALS and includes descriptions of related parallelization attempts. In the third part the RBM and its model-based approach to recommender systems is discussed. To conclude the chapter a taxonomy is presented that provides the reader with an overview of how the methods presented relate to each other.

Chapter 3 reviews some of the most important principles of parallel programming using CUDA, such as the general programming model and the memory model. It also presents the motivation of using GPUs and CUDA over CPUs.

Chapter 4 presents the design and implementation of two of the implemented recommender system algorithms. It illustrates the SGD and RBM algorithms and elaborates on how they were parallelized using CUDA.

Chapter 5 describes the design and implementation of the CUDA ALS-WR implementation. It presents the standard ALS-WR algorithm and details how it was modified to run efficiently on GPUs.

Chapter 6 presents the results of the various algorithms and compares their performance to each other. It also covers how the various algorithm parameters are affected by changes in e.g. dataset sparsity.

Chapter 7 summarizes the results, and proposes future work to extend this thesis.

2 Recommender Systems

The field of recommender systems can be divided into three major categories:

1. Non-personalized recommender systems that mainly rely on aggregated opinions, such as average scores, or on product association (e.g. Amazon's "people who bought this, often bought this together with it").

2. Content-based recommender systems that use a user's profile information and the user's history to provide recommendations.

3. CF systems that additionally include the opinions of other users to provide good recommendations [41].

Hybrid approaches exist that combine multiple of these techniques [5].

This chapter gives a brief introduction to the different categories of CF recommender systems and covers the basic principles of matrix factorization and model-based recommender systems.

2.1 Rating Matrix

The rating matrix is a common data abstraction used for CF recommender systems, where the element in the matrix at position (i, j) will be k if a user i

$$\begin{array}{c}\text{Rating matrix}\\ \begin{array}{c|cccc|} & v_1 & v_2 & v_3 & v_4 \\ u_1 & 1 & 5 & ? & 2 \\ u_2 & ? & 1 & ? & 5 \\ u_3 & 4 & 2 & 5 & ? \\ u_4 & 2 & ? & 1 & 2 \end{array}\end{array} \xrightarrow{\text{Prediction}} \begin{array}{c}\text{Recommendation matrix}\\ \begin{array}{|cccc|c} v_1 & v_2 & v_3 & v_4 & \\ 1 & 5 & 1 & 2 & u_1 \\ 4 & 1 & 4 & 5 & u_2 \\ 4 & 2 & 5 & 5 & u_3 \\ 2 & 2 & 1 & 2 & u_4 \end{array}\end{array}$$

Figure 2.1: Illustration of the rating matrix and the prediction step.

has given item j a rating k. Figure 2.1 displays a rating matrix on the left hand side, where the missing entries are represented by a question mark. It can be seen that the first user, u_1, has given the second item, v_2, a rating of 5. Using the rating matrix as an input, the recommender systems task is to compute accurate predictions for the missing ratings, which can then be used to provide users with recommendations from the recommendation matrix.

2.2 Collaborative Filtering

The systems considered in this thesis belong to the class of CF recommender systems, which find their application in most of the successful recommendation systems found on the Web [48]. They are based on the assumption that people who agreed on item ratings in the past, are likely to agree on item ratings that have only been rated by one of the users.

CF recommender systems can be subdivided into three types, (i) neighborhood-based, (ii) model-based, and (iii) dimensionality reduction approaches [35].

2.2.1 Neighborhood-Based

Neighborhood based CF systems provide recommendations by grouping users based on some similarity metric using their historic taste profiles. There are two main classes of neighborhood based systems, (i) the user-based neighborhood approach, which predicts an items rating for a given user based on how similar users have rated the item and the (ii) item-based neighborhood approach, which assess how similar a new item is to the previously rated items.

These approaches are based on two different assumptions. The user-based approach can be used if the user preferences (their tastes) are either relatively stable for individual users or if a users taste moves in sync with the taste of the other users. In contrast to the user-based approach, the item-based approach assumes that the relationships between the individual items are stable but not necessarily the users taste.

There are drawbacks with both of these approaches. To provide the user with a single recommendation, user-based systems need to compute the correlation between a given user and all m other users. This can be computational expensive for large m. For n items in the recommender system, the recommendation process has therefore a complexity of $O(nm)$ for a given user. Since a recommender system, usually, has to provide ratings to all its users, its complexity grows to $O(m^2n)$. Further, because users generally only rate very few of the available items, it can be difficult to define good neighborhoods. Some of these problems are solved by item-based methods, however, item-based methods are being considered to lack in serendipity [12]. This means that users are rarely pleasantly surprised by receiving useful, but unexpected recommendations.

2.2.2 Model-Based

Model-based systems are another commonly used approach to CF recommender systems. In these systems, recommendation models are trained using the available user, item, and rating information. The models can then be used to predict a rating for a given user-item pair. These models can be generalized as

$$f(p_i, q_j) \rightarrow R_{ij},\ i = 1, 2, \ldots m,\ j = 1, 2, \ldots, n\ , \tag{2.1}$$

where p_i is a model parameter for a user i and q_j is a model parameter for item j [52]. Here, m and n denote the number of users and items, respectively. As can be seen from this equation, a set with model parameters p_i and q_j needs to be found that provided the mapping function f return the correct ratings R. Various model based approaches have been used as part of recommender systems, including clustering [30], Bayesian hierarchical models [63], and RBMs [46].

2.2.3 Dimensionality Reduction

Another approach to CF is the dimensionality reduction or matrix factorization approach, which in recent years has grown in popularity [33, 36]. This method is motivated by the observation that the rating matrix is an overfit representation of the users tastes, as ratings can depend on external factors, such as a users mood. The dimensionality reduction approach achieves a more compact representation of the users tastes, by modeling the recommendation problem as a matrix completion problem given a set of observed elements in a matrix.

There are, however, infinitely many possible matrices that agree with the observed elements. This means that assumptions have to be made regarding the observed and unobserved elements in the matrix. In recommender systems the most common assumption is that of low-rank,[1] which assumes that the complete rating matrix is of low-rank.

2.3 Matrix Factorization Recommender Systems

The matrix completion problem involves fitting a low-rank model that preserves the key aspects of the original incomplete rating matrix [37], which can be used to estimate the missing entries. This implies that all users and items are mapped to the same low dimensional latent[2] feature space with dimension k, where each user and item is represented by k features and similarities can be found using inner products. This means that each user has a feature vector representing the users taste and each item has a feature vector representing what taste the item belongs to. Figure 2.2 illustrates the concept for a two dimensional feature space. Items that are close to a user in the feature space indicate that the users taste corresponds well to the items taste category. This means that the user is likely to enjoy the movie, which therefore should receive a high recommendation score. For example, the girl with the round face in Figure 2.2 is likely to enjoy the movie Titanic, which should therefore receive a high recommendation, whereas she is not likely to enjoy the movie Goodfellas. The mapping for recommender systems is commonly based on SVD-dimensionality reduction.

2.3.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is a technique in linear algebra that has found great applicability in the field of IF, and has also been used to solve information retrieval problems as part of the Latent Semantic Analysis technique [11]. The SVD was initially discovered by the mathematicians Eugenio Beltrami, who in 1873 published the first article on the topic, and Camille Jordan, who independently arrived at the same but a more complete solution in

1. Rank is a term used in linear algebra describing the number of linearly independent rows and columns in a matrix. In our case, a rating matrix of low-rank has many user and item feature vectors that are linear combinations of each other.
2. Features do not have explicit semantic interpretations.

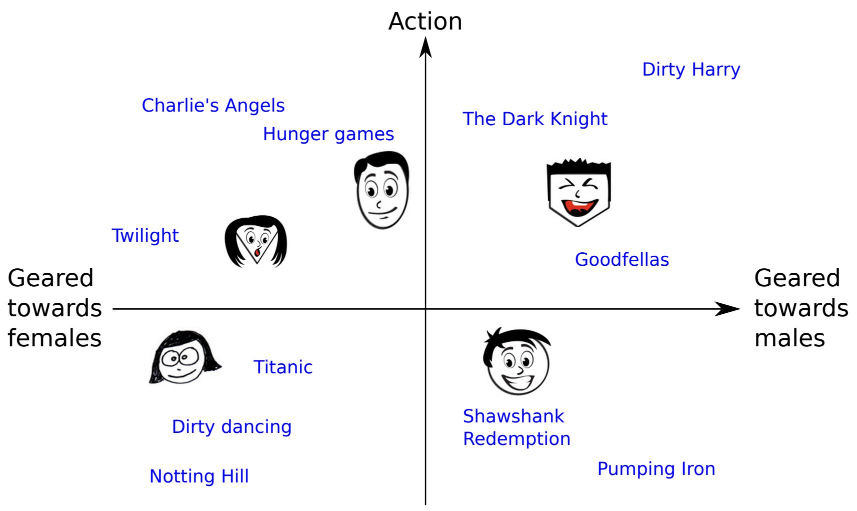

Figure 2.2: A 2-dimensional feature space for matrix factorization recommender systems.

the following year [53]. The method is a common technique used as part of dimensionality reduction, as it transforms correlated variables into uncorrelated variables that better represent the variance in the data. SVD ensures ordering of the dimensions by the amount of variance in the data that the dimensions represent. It is therefore often used to take an overfit representation of high dimensional data and express it in a lower dimensional space by retaining as much as possible of the variance in the data.

Given an $m \times n$ matrix R that we want to decompose, we have to find the $m \times d$ and $n \times d$ orthogonal matrices U and V respectively, such that

$$R = USV^T \ , \tag{2.2}$$

where S is a diagonal matrix of size $d \times d$, containing eigenvalues of RR^T.

The drawback with this simple expression, however, is that solving it for U and V has a computational complexity of $O(m^2n + n^3)$ [20]. Another drawback of this method, in regards to recommender systems, is that a complete matrix is required. Hence, all elements (ratings) in the matrix need to be known, which would make the task of recommendation irrelevant.

To solve the issue of not being able to compute the SVD due to a non-complete rating matrix, multiple methods have been suggested in the literature. One alternative is based on imputation, where the missing ratings are replaced with meaningful substitute values, like the users average rating as done by Sarwar et al. [49]. Another alternative, which in recent years has become the most used approach, is to compute an approximation of the SVD by making use only of the known ratings, such that

$$R \approx UV^T \ , \tag{2.3}$$

where the resulting matrices U and V will not be orthogonal due to it being an approximation and not a true SVD [13].

The matrices U and V are commonly found by utilizing the sum of squared error cost function

$$L(U,V) = \frac{1}{2}\|R - U^T V\|^2 \tag{2.4}$$

and minimizing it with respect to U and V. The cost function is sometimes referred to as loss function.

As recommender system problems commonly involve very sparse datasets this is often represented as a sum over all the observed rating values [35]. This leads to

$$L(U,V) = \min_{U,V} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}(R_{ij} - U_i^T V_j)^2 \ , \tag{2.5}$$

where

$$I_{ij} = \begin{cases} 1, & \text{if } u_i \text{ has rated } v_j \\ 0, & \text{else .} \end{cases}$$

The matrix completion problem can therefore be stated as

$$(U^*, V^*) = \min_{U,V} L(U,V) \ , \tag{2.6}$$

where $U^{*T}V^*$ correspond to the complete versions of the rating matrix. A missing rating for user i for item j can be predicted as $U_i^{*T} V_j^*$ [55].

2.3.2 Approaches to Solving the Matrix Completion Problem

Several approaches have been suggested to solve the low-rank matrix factorization problem, with two of the most popular ones being gradient-descent and alternating-minimization based [37]. Even though these approaches do not produce a true SVD of rating matrix R, they have performed well in predicting unseen ratings [32].

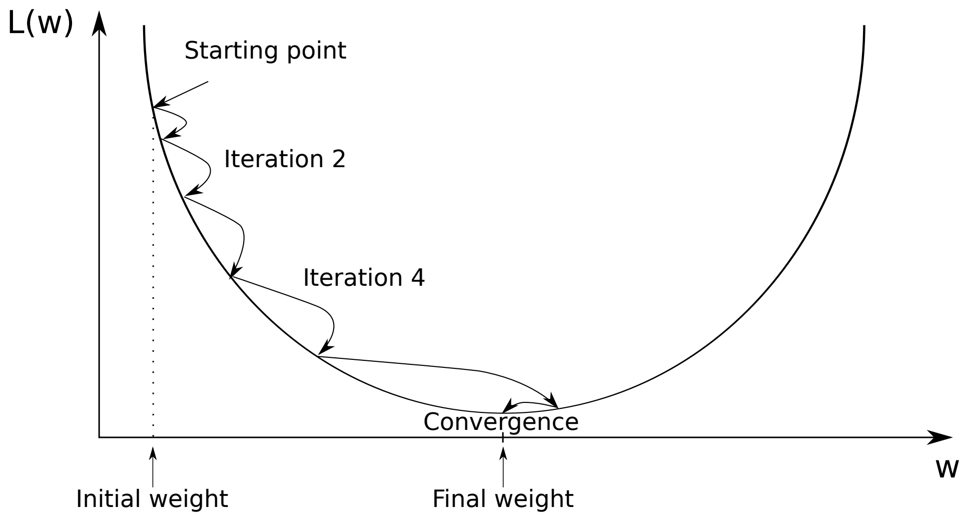

Figure 2.3: Example of how an ideal gradient descent for a convex cost function behaves.

Gradient-descent

Gradient descent methods are iterative optimization algorithms that have been frequently used in the field of machine learning to minimize differentiable cost functions of various algorithms. The standard gradient descent approach computes the gradient of the cost function for the current weight parameters (U and V in the matrix factorization recommender system scenario) and does a step towards the cost functions direction of largest decrease in the parameter space. Figure 2.3 illustrates the gradient descent principle for an ideal convex cost function with one weight-variable. However, cost functions are often more complex in practice and contain not only a single global minimum, but also local minima that the gradient descent can converge to. To decrease the risk of getting stuck in local minima, different techniques like utilizing a momentum factor or an on-line gradient descent variant have been suggested [56].

Two additional problems that are often encountered with gradient descent optimization (and most learning algorithms), are overtraining and overfitting. Overtraining occurs when the weights (free parameters) adapt to peculiarities in a specific training set due to excessive training on the training dataset, whereas overfitting occurs when the number of free parameters is too large, such that the model adapts to details of the trainings set. Both these issues result in a model that is non-generalizable, causing it to perform poorly for unknown (new) feature vectors (data points).

The cost function for the SVD approximation requires a sum over the whole dataset. This means that it is possible to perform gradient descent updates by considering the whole dataset and by performing an update step only after performing a complete sweep through the dataset.

The on-line variant of the gradient descent, the stochastic gradient descent (SGD), offers an alternative approach to finding an optimal solution by only considering a single training point when performing weight updates. This means that the gradients and updates are only computed from the cost function at a specific point instead of based on the global cost function. The intuition behind this approach is that multiple small steps (one for each presented training point) are performed. Each of these steps decreases the global cost based on only knowing one training point. This contradicts with the intuition of the standard gradient descent where large steps are taken towards decreasing the global cost function. Even though the convergence does not follow a straight line, it will converge to a local or global minimum if a sensible learning rate is chosen. Tips on choosing a sensible learning rate can be found in Bottou, 2012 [3].

The SGD generally requires more steps to achieve convergence compared to the standard gradient decent, but since each step calculation only requires one training point, it often leads to a quicker convergence compared to the standard gradient descent algorithm. Additionally, as the update calculation is slightly varying from one training point to another, the SGD is less prone to getting stuck in local minima and will often be able to escape into more optimal regions. The general update equation for gradient descent based algorithms for one weight variable is

$$w_{t+1} = w_t - \mu \frac{\partial L}{\partial w} \, , \tag{2.7}$$

where μ is the learning rate and $\frac{\partial L}{\partial w}$ the partial derivative of the cost function $L(\cdot)$ with respect to the weight w.

For the recommender system scenario, the cost function L depends on two weight parameters U and V. Using Equation 2.5 and adding a L2 regularization term[3] to it to avoid overfitting yields the cost function

$$L(U,V) = \min_{U,V} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}(R_{ij} - U_i^T V_j)^2 + \frac{\lambda_1}{2}||U||_F^2 + \frac{\lambda_2}{2}||V||_F^2 \,, \tag{2.8}$$

where λ_1 and λ_2 are regularization parameters and $||\cdot||_F^2$ the Frobenius norm [35].

Differentiating the cost function in Equation 2.8 with respect to U and V to find an expression for the SGD update equations leads to

$$\frac{\partial \mathcal{L}}{\partial U_i} = \sum_{j=1}^{n} I_{ij}(U_i^T V_j - R_{ij})V_j + \lambda_1 U_i \tag{2.9}$$

and

$$\frac{\partial \mathcal{L}}{\partial V_j} = \sum_{i=1}^{m} I_{ij}(U_i^T V_j - R_{ij})U_i + \lambda_2 V_j \,. \tag{2.10}$$

For a single training point $z(i,j)$, the update for the model (U, V) can therefore be computed by using equation 2.9 and 2.10 and the update can be applied using equation 2.7 such that

$$U_i^{(t+1)} = U_i^{(t)} - \mu \frac{\partial \mathcal{L}}{\partial U_i} \tag{2.11}$$

and

$$V_j^{(t+1)} = V_j^{(t)} - \mu \frac{\partial \mathcal{L}}{\partial V_j} \,, \tag{2.12}$$

where μ is the learning rate and the exponents (t) and $(t+1)$ indicate the t^{th} and $(t+1)^{\text{th}}$ iteration, respectively. It can be seen that each training point requires the update of both the whole vector U_i and the whole vector V_j. Algorithm 2.1 illustrates the SGD algorithm using pseudo code.

Choosing the correct learning rate μ is a crucial part, as it has a big impact on the convergence speed of the gradient descent based methods. The learning

3. The regularizer trades off variance vs. bias. Bias is the error due to classifiers having a preconceived notion of what data should look like, whereas a high variance refers to the fact that a model is not restricted enough. Not restricting the model enough leads to overfitting to training data. L2 regularization restricts the parameters to a sphere in the parameter space around the origin, introducing some bias to reduce variance [22].

Algorithm 2.1 – The SGD algorithm.

```
while not converged do
  for each training point $z(i,j)$ do
    update $U_i$ using Equation 2.11
    update $V_j$ using Equation 2.12
  end
end
```

rate is commonly chosen to decrease with the iteration count to guarantee convergence. However, it can be difficult to chose this function, as a slow decrease of the learning rate will result in a slow decrease in the variance of the weight estimate, whereas a quick decrease will require more updates and iterations to converge to a local or global minima [3].

Alternating Least Squares

The cost function for the matrix completion problem in Equation 2.5 is not convex. However, the Alternating Least Squares (ALS) approach is utilizing the fact that fixing one of the parameter matrices U or V will result in a convex, least-squares problem. By alternating which parameter to fix and optimizing the sub-problems of the matrix completion problem, the overall matrix completion problem can be solved iteratively. Using equation 2.5 as a starting point and using weighted-λ-regularization, as was done by Zhou et al. [64], yields

$$\begin{aligned} L(U,V) = \min_{U,V} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}(R_{ij} - U_i^T V_j)^2 \\ + \lambda(\sum_i n_{u_i}||u_i||^2 + \sum_j n_{v_j}||v_j||^2)\ . \end{aligned} \tag{2.13}$$

The global minima of the least-squares sub-problems can then be derived analytically by solving

$$\frac{\partial L}{\partial u_i} = 0 \tag{2.14}$$

and

$$\frac{\partial L}{\partial v_j} = 0\ , \tag{2.15}$$

which yields the update equations

$$u_i = (V_{I_i} V_{I_i}^T + \lambda n_{u_i} E)^{-1} V_{I_i} R^T(i, I_i) \tag{2.16}$$

(2.17)

and

$$v_j = (U_{I_j} U_{I_j}^T + \lambda n_{v_j} E)^{-1} U_{I_j} R^T(i, I_j) \ . \tag{2.18}$$

Given the weight parameter matrix U of user feature vectors, U_{I_j} denotes the sub-matrix of U that contains only the user feature vectors of users that have rated item j. Similarly, given the weight parameter matrix V of item taste vectors, V_{I_i} denotes the sub-matrix of V that contains only the items that the user i has rated. R is the rating matrix, E is the $dim \times dim$ identity matrix and n_{u_i} and n_{v_j} are the number of items a user i has rated and the number of users who have rated a given item j, respectively. dim is the dimensionality of the latent feature space that the users and items are mapped to. A full ALS update for both U and V will in this thesis be referred to as an epoch.

Comparing the Gradient-Descent and Alternating Minimization Approach

Both SGD and ALS are approaches that can be applied to the matrix completion problem, however, there are certain differences between the two methods. The SGD algorithm is computationally less expensive than the ALS, as it does not require solving the least-square problems [61]. Makari 2014 [37], however, states that, for small numbers of dim (< 50), the computational overhead is "acceptable". On the other hand, the ALS algorithm is superior to the SGD with regards to the number of parameters that need to be set, as it does not require the specification of a learning rate.

2.3.3 Parallelizing Matrix Factorization Recommender Systems

Multiple solutions have been proposed to parallelize and/or distribute the computation of the SGD algorithm. Unfortunately, the generic SGD algorithm is not embarrassingly parallel, which makes it difficult to scale to very large amounts of data, as locks or other forms of synchronization primitives have to be used. The reason for the difficulties is that the weights learned in each iteration will depend on the weights learned in the previous iteration. Also, the updates computed for one element in the matrix will lead to an update in the whole weight vector for that column and row. For example, consider a case

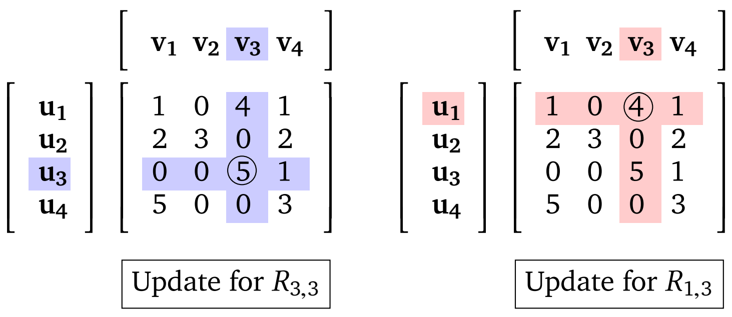

Figure 2.4: Parallelization problems with the SGD.

with four users u_1, ... u_4, and four items v_1, ... v_4, as illustrated in Figure 2.4. The Figure shows two update calculations, one for the rating $R_{3,3}$ and one for the rating $R_{1,3}$. The circle in each of the two illustrations represents the rating that is used for the update. It can be seen that for the first update the third item feature vector (v_3) and the third user feature vector (u_3) are being updated. However, in the second update the third item feature vector (v_3) is updated as well. This means that these two updates can not be performed in parallel in a consistent manner.

Attempts to parallelize SGD have relied on either the inherent structure in the matrix factorization problem or certain properties of the matrix factorization. For instance, the stratified stochastic gradient descent (SSGD) [15] algorithm uses the structure of the matrix factorization problem to solve the parallelization issue and provides the foundation of many of the parallelized approaches. The SSGD is based on the idea that the total cost function can be seen as a sum over losses of smaller areas (stratum losses), which together make up the whole matrix. The losses are found using loss (cost) functions $L_1(w), \dots, L_n(w)$ such that

$$L(w) = w_1 L_1(w) + w_2 L_2(w) + \dots + w_n L_n(w) \, . \tag{2.19}$$

It has been proven that, by carefully minimizing these stratum losses, the algorithm will converge [15]. Based on this idea, the distributed stochastic gradient descent (DSGD) was developed. It utilizes the following theorem, which was proposed and proved by Gemulla et al. [15]:

Theorem 1. *Two training points $z_1 = (u_1, v_1) \in Z$ and $z_2 = (u_2, v_2) \in Z$ are interchangeable with respect to any loss function L having summation form if they share neither row nor column, i.e., $u_1 \neq u_2$ and $v_1 \neq v_2$.*

DSGD subdivides the rating matrix into $d \times d$ blocks of size $m/d \times n/d$, where

m and n refers to the row and columns of the matrix. It then utilizes the fact that blocks, which are diagonal from each other (share no row or column) are interchangeable according to Theorem 1 and schedules the blocks that are independent from each other together.

The SGD algorithm Jellyfish [43] is also based on the idea of the SSGD and partitions the rating matrix into independent blocks. However, unlike in the SSGD the data is reshuffled each epoch to randomize the order and achieve faster convergence. To avoid the full cost of reshuffling the algorithm relies on overlapping the reshuffling and SGD computation. This means that multiple copies of data need to be kept and thus the algorithm has high memory requirements. Additionally reshuffling is expensive. Another algorithm that has been proposed by Recht et al. in 2011 [44] is Hogwild, a lock-free algorithm, which is described in more detail in Section 4.1.

Parallelization approaches using CUDA exist for the SGD algorithm. Kato and Hosino [28] take the SGD and parallelize it using CUDA. However, unlike in the sequential version, their algorithm is not updating U and V for each training sample, but instead processes U and V separately. It first updates U for all training samples and then V for all training samples. This procedure is illustrated using pseudo code in Algorithm 2.2. They claim that a speedup factor of 20 over a sequential CPU implementation was achieved for a randomly generated dataset.

Algorithm 2.2 – The modified SGD algorithm suggested by Kato and Hosino [28].

```
while not converged do
  for each training point $z(i,j)$ do
    update $U_i$ using Equation 2.11 in parallel
  end
  synchronize
  for each training point $z(i,j)$ do
    update $V_j$ using Equation 2.12 in parallel
  end
  synchronize
end
```

The parallelization of ALS is commonly based on the idea that each of the least square problems can be computed independently, as updates to U do not affect the updates to V [64]. Thus, multiple threads can calculate updates for U (or V) in parallel. Using this observation, it is possible to extend ALS to a distributed shared-nothing algorithm [55], where every node is responsible for updating parts of the feature matrices in parallel. Other algorithms based

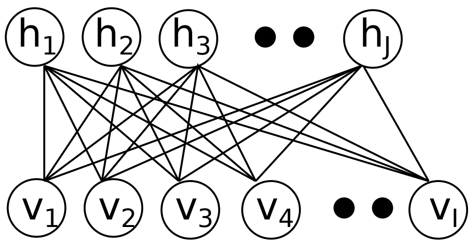

Figure 2.5: A Restricted Boltzmann Machine with J hidden and I visible units.

on the concept of alternating minimizations have been suggested based on a coordinate descent approach to the matrix factorization problem [60].

2.4 Model-Based Recommender Systems

The most popular model-based approach to recommender systems is the Restricted Boltzmann Machine (RBM), which is considered to be on par with the matrix factorization recommender systems and thereby also one of the best standalone approaches to the recommender system task [46]. Besides recommender systems, RBMs have been used to model complicated, high-dimensional data and have found application in modeling of speech and human motion [39, 54].

2.4.1 Restricted Boltzmann Machines

RBMs can be viewed as a two layer stochastic, energy based neural network[4] where each of the neurons (or units) in the hidden and visible layer are connected with no intra-layer connections. In graph theory this is referred to as a complete bipartite graph [34], and is illustrated in Figure 2.5. The units in the hidden layer correspond to stochastic binary feature detectors, whereas the visible units represent the observed binary states (input data).

The RBM is based on an energy model, where the joint configuration of the visible and hidden units has an energy of

4. A neural network is a common machine learning technique that is inspired by the brain and the brains structural constituents, the neurons [23].

$$E(\boldsymbol{v},\boldsymbol{h}) = -\sum_{i\in\text{visible}} a_i v_i - \sum_{j\in\text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}\ , \tag{2.20}$$

where v_i corresponds to the binary state of the visible unit i, and h_j corresponds to the binary state of the hidden unit j. w_{ij} corresponds to the weight connecting the two units and a_i and b_j correspond to the visible and hidden biases, respectively [25].

The probability for every pair of v_i and h_j is given by the following energy function, which is based on the Boltzmann distribution in statistical mechanics [50],

$$p(\boldsymbol{v},\boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v},\boldsymbol{h})} \tag{2.21}$$

with the partition function Z, sum of all Boltzmann factors,

$$Z = \sum_{v,h} e^{-E(\boldsymbol{v},\boldsymbol{h})} \tag{2.22}$$

turning the Boltzmann factor into a probability.

This means, that the probability for the visible vector v is given as

$$p(\boldsymbol{v}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})}\ . \tag{2.23}$$

From this it is possible to see that the probability for a given visible vector can be increased by either lowering the energy for the given vector, or increasing the energy for the other visible vectors. Training of RBMs is usually performed using stochastic steepest ascent,[5] where the derivative of the log probability with respect to the weight is given as

5. Stochastic steepest ascent is similar to statistic gradient descent. However, in the stochastic steepest ascent, a cost function is maximized by following the positive gradient to a local/global maxima.

$$\frac{\partial \log p(\boldsymbol{v})}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \;, \tag{2.24}$$

leading to the update rule

$$\Delta w_{ij} = \mu \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \right) \;, \tag{2.25}$$

where μ is the learning rate, $\langle \cdot \rangle_{\text{data}}$ corresponds to the expectation under the distribution of the data, and $\langle \cdot \rangle_{\text{model}}$ to the expectation under the distribution of the model.

An unbiased sample for the first term $\langle v_i h_j \rangle_{\text{data}}$ can be easily computed as there are no intra-layer connections, which means that for a given training vector $\boldsymbol{v}$ it is possible to compute the binary state of the hidden units as

$$p(h_j = 1 | \boldsymbol{v}) = \sigma \left(b_j + \sum_i v_i w_{ij} \right) \;, \tag{2.26}$$

where $\sigma(\cdot)$ is the logistic sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \;. \tag{2.27}$$

However, finding an unbiased sample for the second term, $\langle v_i h_j \rangle_{\text{model}}$, is more difficult, as it would require running a Monte Carlo Markov Chain to convergence, with Gibbs sampling being the transition operator of the chain as illustrated in Figure 2.6 [24].The conditional probabilities of the hidden units are given as

$$p(v_i = 1 | \boldsymbol{h}) = \sigma \left(a_i + \sum_j h_j w_{ij} \right) \;. \tag{2.28}$$

As can be seen in the figure, finding a perfect sample for $\langle \cdot \rangle_{\text{model}}$ would be very time consuming and inefficient. Geoffrey Hinton presented a simplified learning rule, where the distribution of the first reconstruction $\langle \cdot \rangle_{\text{recon}}$ is used instead of the model statistics [24]. Even though this simplified learning rule only roughly approximates the actual gradient of the log probability, it has been shown to perform well in many application scenarios [46]. Increasing the number of Gibbs sampling steps before collecting the statistics yields more exact updates, and is therefore usually increased as the training progresses, leading to

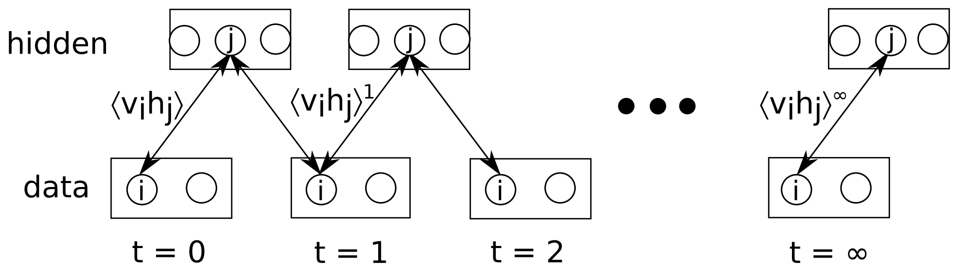

Figure 2.6: The figure illustrates the Gibbs sampling for RBMs until convergence. To compute $\langle v_i h_j \rangle_{\text{model}}$ the statistics of the model need to be found. This can be done by performing Gibbs sampling until convergence $\langle v_i h_j \rangle^{\infty}$. This is, however, a very time consuming procedure.

$$\Delta w_{ij} = \mu \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{T}} \right) \ , \tag{2.29}$$

where T is the number of steps the Gibbs sampling was performed.

Restricted Boltzmann Machines for CF

RBMs were first introduced to the task of CF under the Netflix Prize [46], and ended up being a crucial part of the winning entry [32]. Initially, the use of RBMs seems to be unintuitive and inefficient, as each user has only rated a small number of items. Representing each item by a visible unit would yield a huge RBM with each training vector being very sparse. Instead of doing this, Salakhutdinov et al. [46] suggests that each user has a small private RBM with the visible units corresponding only to the actual ratings of the user. Each personal RBM has only one training vector, but as all models use and update the same weights and biases the learning will contribute to improving the "global" model.

In recommender system scenarios, the modeling of the visible units is slightly different to the original RBM. Here, the input vector $\boldsymbol{v}$ is replaced by a $K \times m$ observed binary indicator matrix, where element v_i^k is set to one if the user has rated item i with rating k. K and m are the number of different possible ratings and the number of rated items, respectively. Figure 2.7 illustrates this new model design for a scenario with $K = 5$ (e.g., 1 to 5 star ratings).

Instead of Equation 2.20, the energy of the joint configuration of the visible and hidden units for the modified model is given as

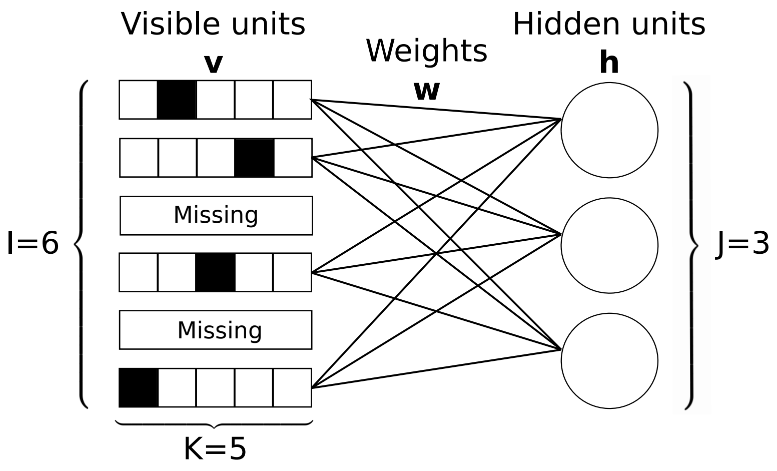

Figure 2.7: Illustration of the Restricted Boltzmann Machine design to the CF problem.

$$E(\boldsymbol{V},\boldsymbol{h}) = -\sum_{i\in\text{visible}}\sum_{k=1}^{K} a_i^k v_i^k - \sum_{j\in\text{hidden}} b_j h_j - \sum_{i,j}\sum_{k=1}^{k} v_i^k h_j W_{ij}^k \ . \qquad (2.30)$$

The Restricted Boltzmann Machine for Collaborative Filtering (RBM-CF) makes use of a conditional multinomial distribution to model the columns of V such that

$$p\left(v_i^k = 1|\boldsymbol{h}\right) = \frac{\exp\left(a_i^k + \sum_j h_j W_{ij}^k\right)}{\sum_{l=1}^{K}\exp\left(a_i^l + \sum_j h_j W_{ij}^l\right)} \qquad (2.31)$$

and a conditional Bernoulli distribution for modeling of the hidden units such that

$$p\left(h_j = 1|\boldsymbol{V}\right) = \sigma\left(b_j + \sum_i\sum_{k=1}^{K} v_i^k w_{ij}^k\right) \ . \qquad (2.32)$$

The weight matrix in a CF recommender system is a three dimensional matrix, where one dimension is used to represent the different rating options. The update rule for the weight matrix is similar to Equation 2.29

$$\Delta W_{ij}^k = \mu\left(\langle v_i^k h_j\rangle_{\text{data}} - \langle v_i^k h_j\rangle_{\text{T}}\right) \ . \qquad (2.33)$$

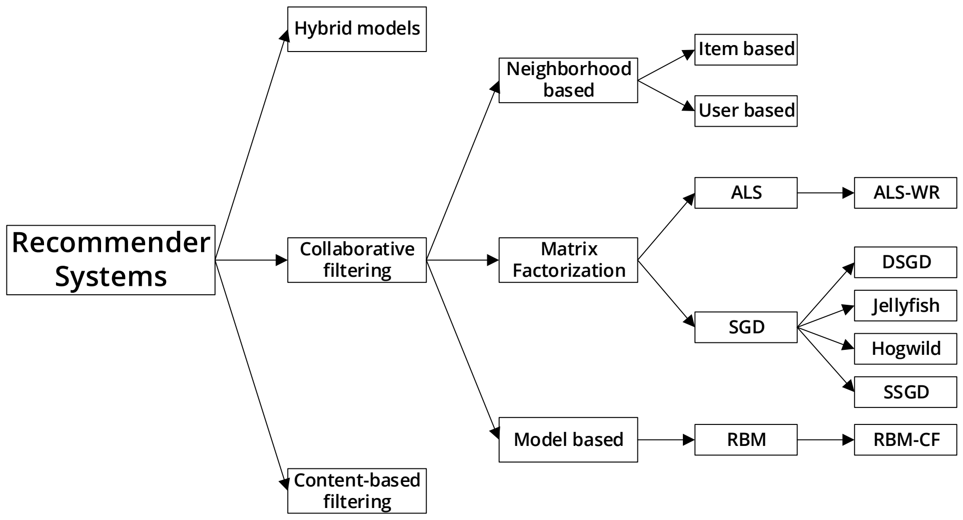

Figure 2.8: A taxonomy of the recommender systems discussed in this section. Only the systems that have been explicitly discussed are displayed in the hierarchy.

Testing is performed by clamping the observed ratings $\boldsymbol{V}$ on the visible units, computing $p(h_j = 1|\boldsymbol{V})$ for all hidden units and then computing a reconstruction using $p(v_i^k = 1|\boldsymbol{h})$. The prediction can then be found as

$$\sum_{k=1}^{K} p\left(v_i^k = 1|\boldsymbol{h}\right) \times k \; . \tag{2.34}$$

Even though training the RBM-CFs can take some time, the above equation shows that the prediction of new ratings given the observed ratings can be performed in $O(J)$ where J is the number of hidden units $\boldsymbol{h}$.

2.5 CF Recommender System Taxonomy

This chapter has given a general description of the main techniques for CF recommender systems and has provided the reader with a more detailed analysis of the three approaches used in this thesis: the SGD, the ALS and the RBM-CF algorithms. We have devised a taxonomy of the various recommender systems that have been explicitly mentioned in this chapter, which is shown in Figure 2.8. It illustrates how the various systems and recommender system classes are related to each other.

Chapter 1 introduced the incentive to use parallel programming techniques for recommender systems. The next chapter will focus on why GPUs are ideal for this parallelization and provide the reader with a basic understanding of how CUDA, NVIDIA's parallel programming model for GPUs, can be used.

3 Design principles for efficient CUDA implementations

The use of GPUs for computational intensive tasks has in recent years experienced a rapid increase in popularity [16]. This chapter explains the motivation for using GPUs over conventional CPUs in the context of recommender systems and introduces the reader to the CUDA programming model.

3.1 Motivation

Ever since the release of the first commercial microprocessor (Intel 4004) in 1971 [2], general-purpose CPUs have historically been the method of choice for performing computations, and their performance has continuously improved. During the major part of the CPUs development the main focus was on supporting sequential workloads, leading to an optimization for sequential execution workloads at the cost of parallel execution.

Many techniques have been used to optimize CPUs for sequential execution. These include caching, out of order processing, and branch prediction. This added complexity actually makes up most of the circuitry on the CPU, thereby

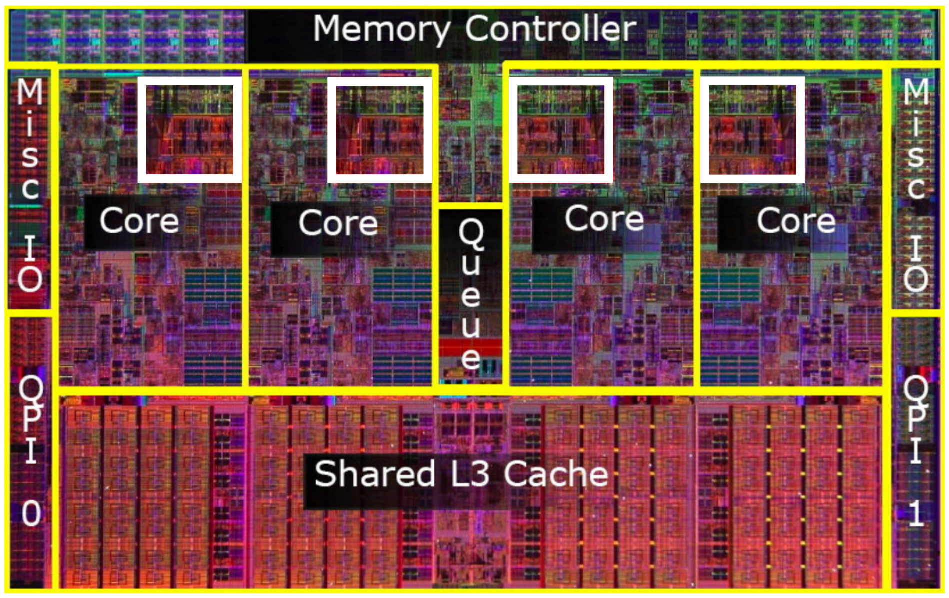

Figure 3.1: Illustration of Intel's Core i7 processor (Codename Bloomfield). The white boxes are used to illustrate the portion of the chip used for execution units. The figure is based on the one by Glaskowsky, 2009 [17].

occupying space which otherwise could be used for additional execution units. This is illustrated in Figure 3.1, which shows a Core i7 processor where the area occupied by execution units on the chip is highlighted by white boxes. It can be seen that these only make up a small fraction of the total chip area.

Additionally, in recent years performance improvements have delivered diminishing returns. This is due to challenges such as the memory wall, the instruction-level parallelism (ILP) wall, and the power wall [4]. The memory wall refers to the increasing gap between the CPU and memory speeds, which leads to memory being a performance bottleneck unless larger caches are introduced. The ILP wall is used to refer to the increased difficulty of parallelizing instructions in a single thread, unless the complexity is increased by, for instance, making use of more aggressive branch predictions. The power wall refers to the fact that increasing clock frequencies of processors leads to an increase in energy consumption and thereby an increase in production of heat. All these challenges have led to an increase in processor complexity.

GPU computing was introduced to the consumer market in 2001 with the release of the NVIDIA's GeForce 3, making use of pixel shaders, which to a limited extend where programmable [17]. Over the following years the programmability was extended leading up to the GeForce 7800, which was released in 2005. It used three types of programmable engines for the various stages of the graphics processing pipeline [17].

The drawback of this architecture was that three types of programmable engines had to be controlled, and the throughput in the various stages of the pipeline had to be balanced carefully. In 2006 a new architecture was released as part of the GeForce 8800 that used a unified shader architecture and removed the pipelined architecture. In the unified shader architecture each shader core can perform all shader tasks in the pipeline, thereby removing the need of pipeline stage balancing. Together with the GeForce 8800, CUDA—a parallel programming model for NVIDIA's GPUs—was released, making GPU programming more accessible to consumers [47].

Further development has in the past years led to improved architectures, with the current state of the art being the Maxwell architecture [10]. The Kepler architecture [9]—which has been used as part of this thesis, and which was the state of the art in GPU design prior to the Maxwell architecture—is shown in Figure 3.2. The figure shows that the GPU, to a large extent, is composed of computational units (dark orange boxes) with a small amount of support elements. Unlike the CPU (Figure 3.1), which has been optimized for sequential executions with a large amount of conditional branches (control-flow intensive), most of the GPU chip area and consumed power is used for arithmetic work. This makes the GPUs ideal for workloads that can use multiple threads and that are computationally intensive.

The potential for speedup of matrix computation on GPUs compared to CPUs, is well illustrated in an experiment, where an element-wise cube is taken of a square matrix. This is an operation requiring many multiplications that are independent for each element in the matrix and is an embarrassingly parallel operation. Figure 3.3 displays the computation time for running the experiment with different sized matrices on both the CPU and the GPU. It shows that the GPU performs and scales a lot better than the CPU. However, in this plot the time spent for transferring the matrix to the global memory of the GPU has not been included. It only displays the ideal effect that can be achieved if many operations are chained on the GPU so that the computation/communication ration becomes large.

Accounting for the memory transfer time, the observed speedup is much lower, due to the fact that the amount of data that needs to be transferred increases proportionally to the square of the matrix dimension. This can be seen in Figure 3.4.

This experiment leads us to our first design principle.

Design Principle 1. *Minimize host to device communication.*

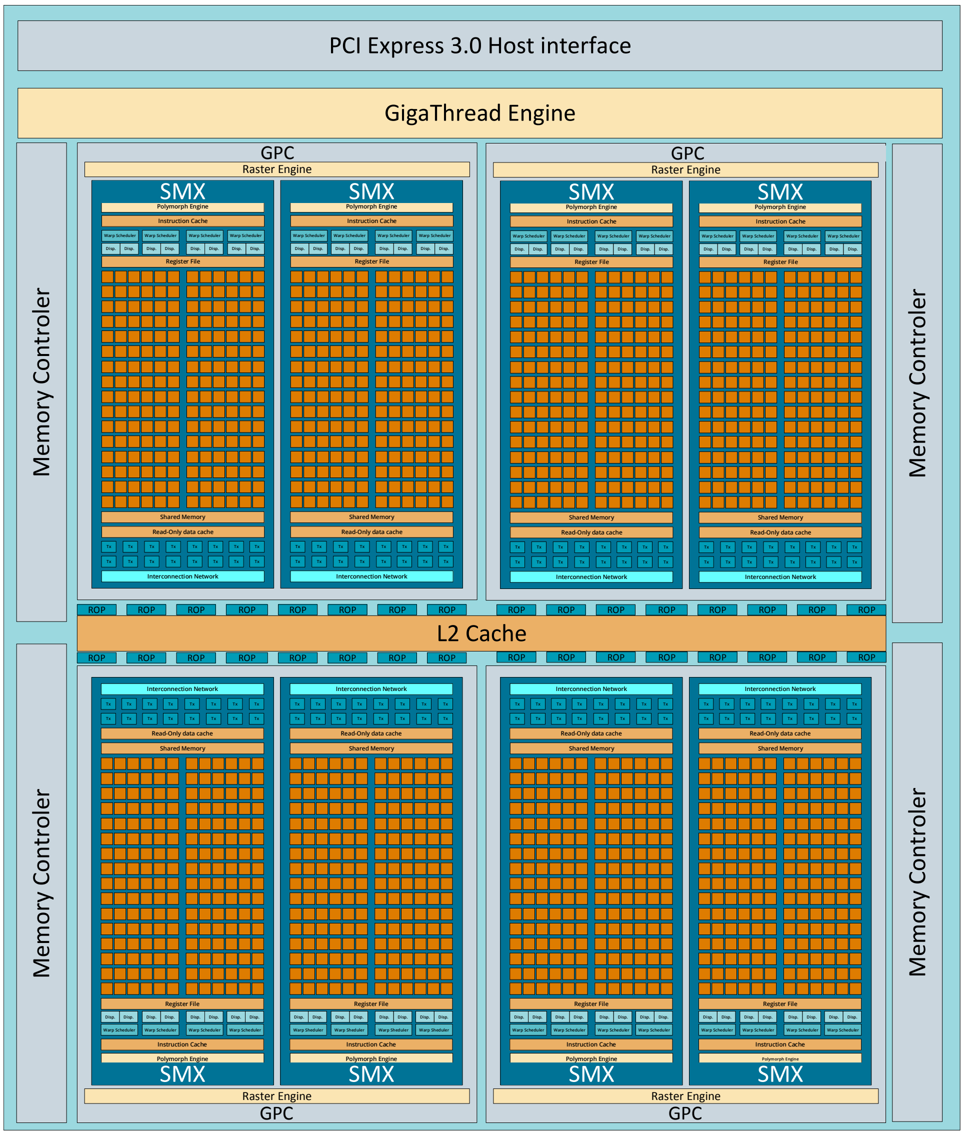

Figure 3.2: Example of a Kepler architecture. It can be seen that the fraction occupied by computational units (dark orange boxes) is much larger than for a modern CPU.

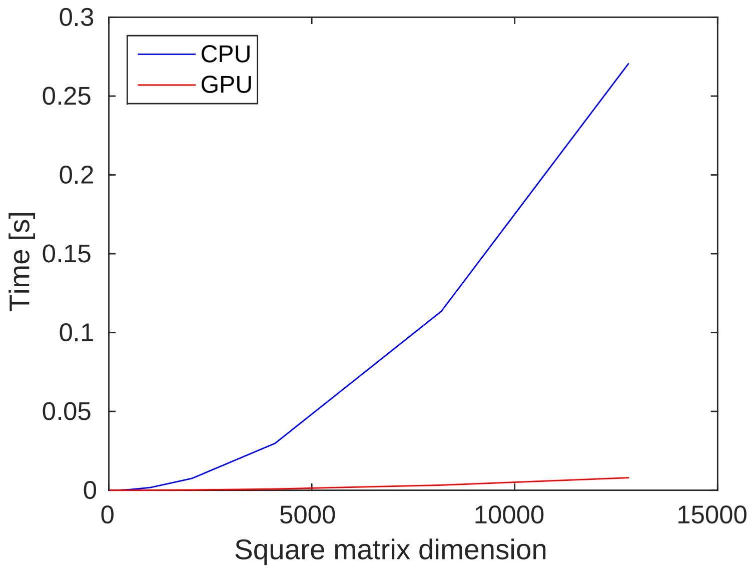

Figure 3.3: The graph illustrates the difference in computational performance, which can be achieved for certain workloads by an GPU over an CPU. The example chosen was an elementwise matrix operation, and only the computation part, excluding the memory transfer, is timed.

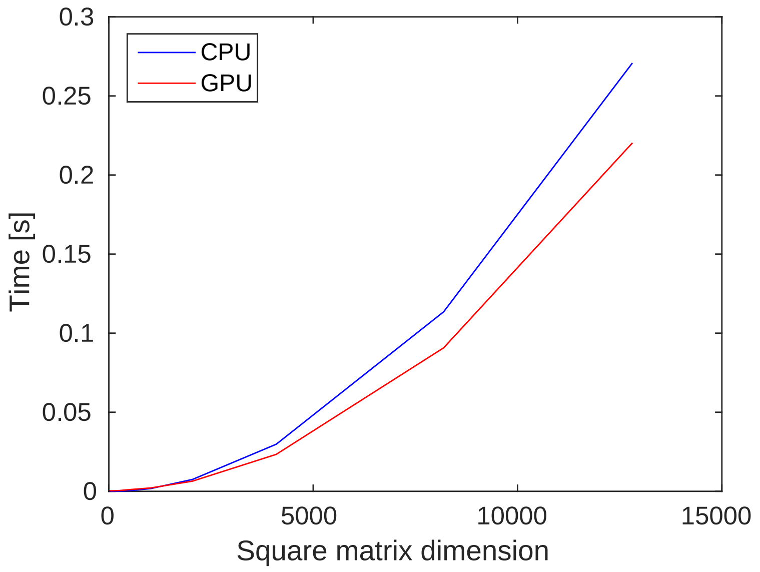

Figure 3.4: The graph shows the difference between the CPU and GPU for an elementwise matrix operation. The time for memory transfer as well as the computation is measured. It is based on the same experiment as Figure 3.3.

In this and the following chapters important aspects related to CUDA programming for, but not limited to, recommender systems will be highlighted as design principles.

3.2 CUDA Programming Model

Modern NVIDIA GPUs consist of processing units called Streaming Multiprocessors (SM), each one consisting of multiple Scalar Processors (SP). CPUs can use these by initializing kernels—functions that run massively parallel on the GPU and that are executed by many threads. GPU threads, unlike CPU threads, are extremely lightweight and the overhead of creating and switching between threads is very small. With the development of improved GPU architectures the CUDA programming model has also improved. The features supported by a GPU are indicated by the *compute capability*, its version number [8].

CUDA threads are organized in thread blocks, where each thread block can be one-, two-, or three-dimensional, and where each thread has a unique index vector describing its location in the thread block. Thread blocks are organized in a similar way (one-, two-, or three-dimensionally) in a grid with a unique index identifying each block [8]. When initializing a kernel, the grid layout—as well as the thread block layout—has to be chosen to fit the problem.

CUDA uses a single instruction, multiple threads (SIMT) architecture, where each SM runs one or more thread blocks. Each thread block contains a multiple of 32 threads called a warp, the minimum unit of execution for modern GPUs. Threads in a warp execute in lockstep regardless of divergent branching. This means that if not all 32 threads agree on the execution path, some of the threads will be idle during parts of the computation and the warps full efficiency will not be achieved.

Putting this in respect to the GPU architecture presented in Figure 3.2 it is possible to see that the GPU is made up of eight Kepler Generation Streaming Multiprocessorss (SMXs) divided up into four Graphics Processing Clusters (GPCs). A more detailed view of the SMX is illustrated in Figure 3.5. Besides the SPs, the SMXs also consist of four warp schedulers, eight dispatch units, sixteen texture units, and various types of memory. The memory model of the GPU will be explained in the next section.

In the CUDA programming model the CPU, or *host* using CUDA terminology, and the GPU (*device* in CUDA terminology) are assumed to have separate memory spaces. The communication between them is performed using the global device memory, which is persistent through kernel launches. The host therefore

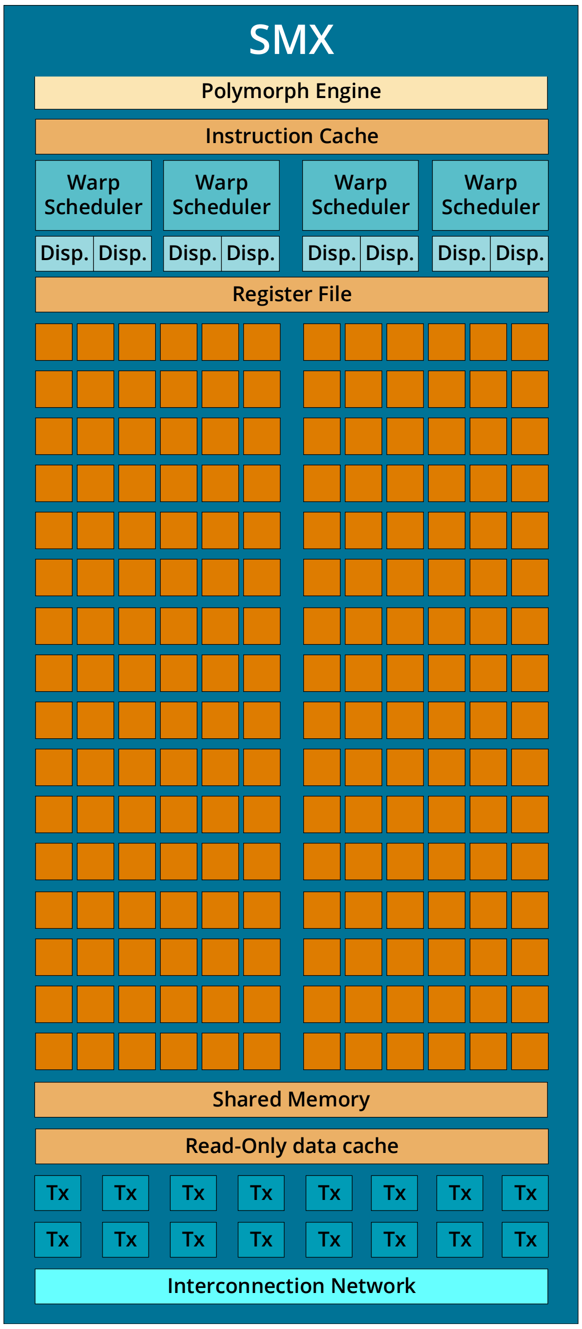

Figure 3.5: The SMXs in the Kepler GPU architecture.

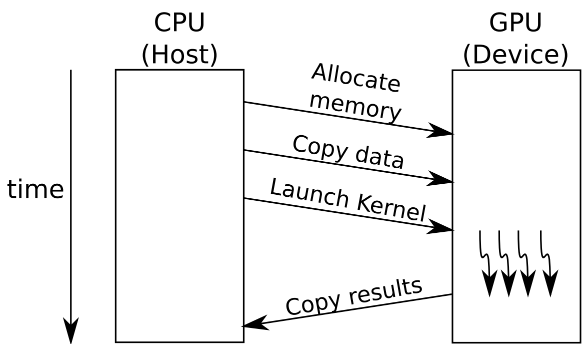

Figure 3.6: Illustration of a common program flow in a CUDA program.

has to allocate memory on the device prior to copying the input data to the GPUs memory space. The kernel can then be launched by the CPU. Once the computation has been completed the CPU is responsible for copying the results back to the CPUs memory space. The control flow in a typical CUDA program is illustrated in Figure 3.6. Code Listing 3.1 illustrates the invocation of a kernel function of name *kernel* in C. The kernel is invoked on a grid containing of $10 \times 10 \times 10$ blocks, where each block contains $16 \times 16 \times 2$ threads, with the parameters A and B.

Code Listing 3.1 – Kernel invocation in C.

```
int invokeKernel(A, B)
{
  dim3 threadsPerBlock(16,16,2);
  dim3 blocksPerGrid(10,10,10);
  kernel<<<blocksPerGrid, threadsPerBlock>>>(A, B);
}
```

3.3 CUDA Memory Model

The CUDA memory hierarchy shown in Figure 3.7 illustrates various memory spaces that CUDA threads might access during their execution. Each thread has access to three main different levels of memory: local memory, shared memory and global memory. Local memory is private to each specific thread and cannot be accessed by other threads. It consists of on-chip registers and a small amount of off-chip memory. Threads in a block, in addition to their individual local memory, also have access to shared memory, which is visible to all the threads in the block. The largest and slowest memory level is the global memory, which all threads can access [8]. It can be used by all threads in the grid to communicate with threads on different SMs or store data that all

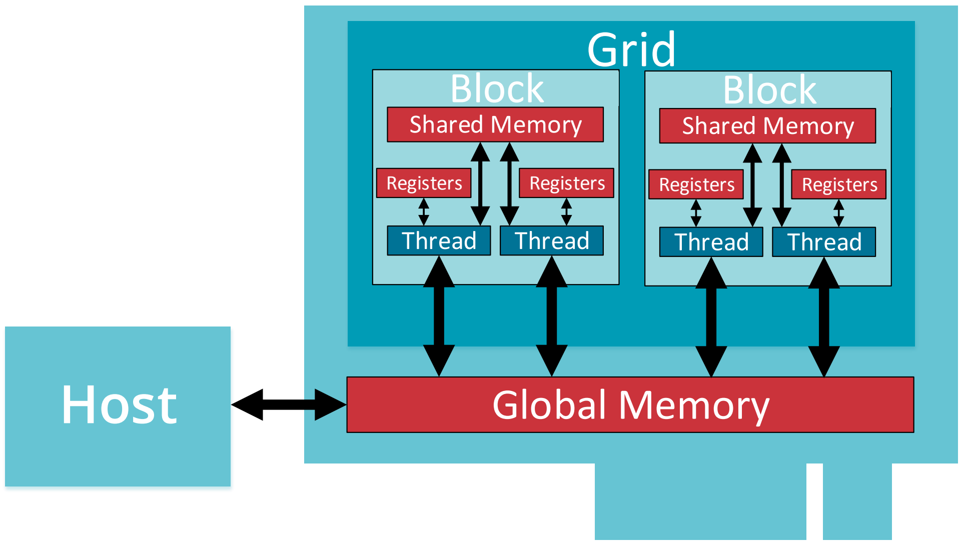

Figure 3.7: Illustration of the memory hierarchy of CUDA GPUs.

threads need to access. Code Listing 3.2 illustrates how global device memory can be allocated in C and how data can be transfered from the host memory to the device and back again.

The specific CUDA architecture, where threads are divided into warps, has to also be considered when performing memory accesses. Global reads and writes by threads in a warp are grouped into as few transactions as possible by the device in an effort to minimize bandwidth usage. If one thread accesses global memory and only needs to read/write a small subset of the data, this will be represented as one transaction. However, if other threads need to access other parts of the same memory area, the operation can be performed as one transaction. A kernel is therefore most efficient when threads read and write to sequential memory locations. Such a contiguous access is called a *coalesced memory access* and is illustrated in Figure 3.8, where the memory chunk that a thread can read from global memory in one transaction is represented by the top array. Early CUDA devices with compute capability less than 2.0 allowed transaction to be coalesced per half-warp (16 threads), whereas newer devices allow larger memory accesses, such that a whole warp (32 threads) can perform one transaction [8].

Figure 3.9 illustrates a different memory access pattern, where the access is not performed sequentially, but all threads still access the same parts of the large chunk of memory. Early devices did not support this form of coalescence and had to issue separate transactions for each of the threads. Current devices, however, support coalesced access even for non sequential memory accesses.

Code Listing 3.2 – Allocating global device memory.

```
int main()
{
  // Size of a 10 element vector
  size_t sizeA = 10 * sizeof(float);

  // Allocate memory on host for a 10 element vector
  float* A = (float*)malloc(sizeA);

  populateVector(&A)

  // Allocate memory on device
  float* A_d;
  cudaMalloc(&A_d, sizeA)

  // Copy vector A to device
  cudaMemcpy(A_d, A, sizeA, cudaMemcpyHostToDevice)

  // Invoke kernel
  dim3 threadsPerBlock(16,16,2);
  dim3 blocksPerGrid(10,10,10);
  kernel<<<blocksPerGrid, threadsPerBlock>>>(A_d);

  // Copy result back to host into array A
  cudaMemcpy(A, A_d, sizeA, cudaMemcpyDeviceToHost);

  // Free memory on device
  cudaFree(A_d);
}
```

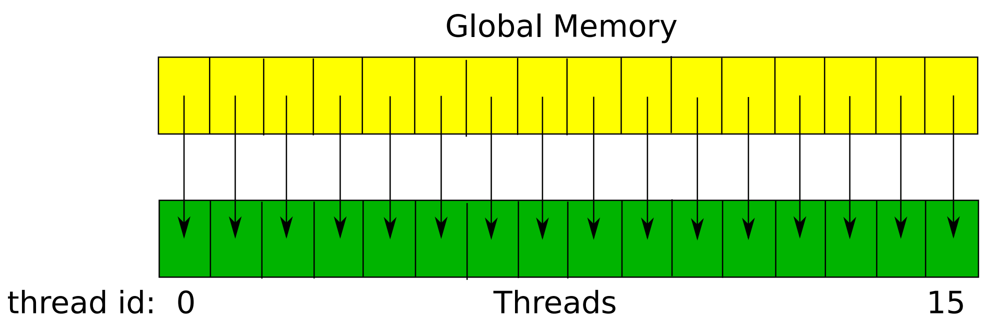

Figure 3.8: Coalesced memory access illustration.

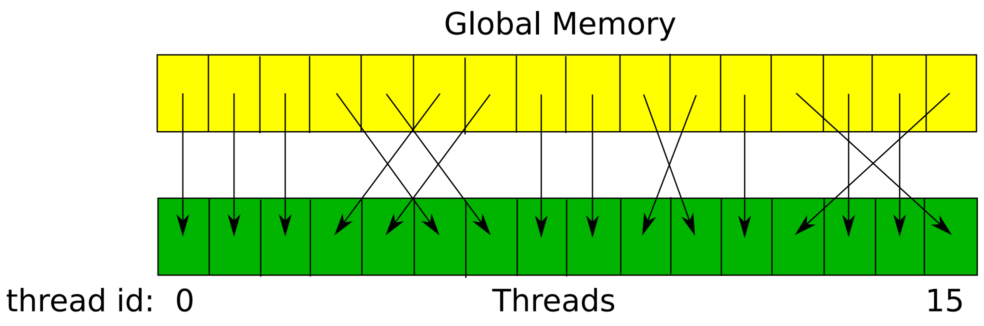

Figure 3.9: Coalesced but not sequential memory access.

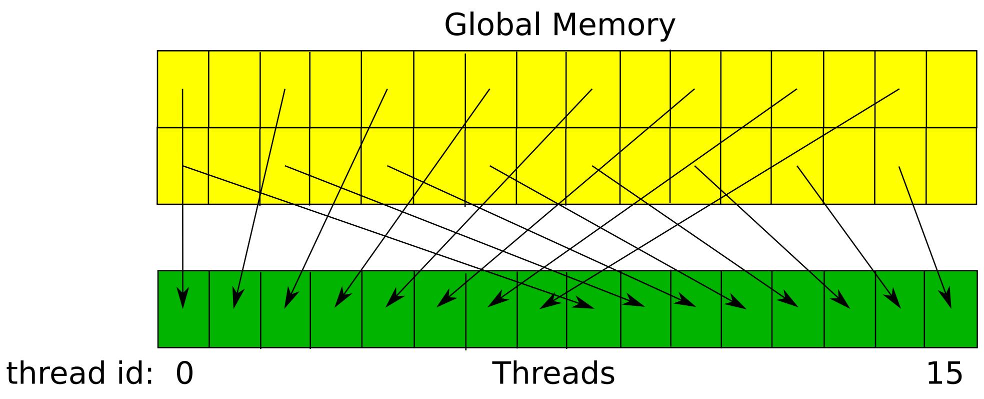

Figure 3.10: Strided, non-coalesced, memory access.

Another memory access pattern is displayed in Figure 3.10, where every adjacent thread is accessing every other array location. This is commonly referred to as strided memory access. It can be seen that in the example figure, two memory transactions need to be performed instead of just the one in the previous examples. A larger stride leads to more memory transactions and thereby to worse performance.

The effect of using coalesced vs. non-coalesced memory access can be illustrated with a similar experiment as the one in Section 3.1. Here, the performance for coalesced memory access is compared with the performance when using the strided memory access pattern of the kernel in Code Listing 3.3.

The input array *a* represents the matrix *A* in row-major order and the threads in each warp will therefore access all the elements in a coalesced fashion as warps in a grid are also in row-major order.

In Code Listing 3.4 (line 12), the memory access pattern has been changed to a strided approach, where the warps in the grid and the elements in memory are still in row-major order, but the elements are accessed in column-major

Code Listing 3.3 – Kernel computing the element wise power of 3 (Coalesced).

```
__global__ void kernel(float *a, float *res, int dim)
{
  // Get x and y threadId in grid
  int x = (blockIdx.x * blockDim.x) + threadIdx.x;
  int y = (blockIdx.y * blockDim.y) + threadIdx.y;

  // Check that threadId's element inside matrix
  if(x > dim || y > dim)
    return;

  // Calculate position in matrix array a
  int idx = y*dim+x;

  res[idx] = a[idx]*a[idx]*a[idx];
}
```

Code Listing 3.4 – Kernel computing the element wise power of 3 (Strided).

```
__global__ void kernel(float *a, float *res, int dim)
{
  // Get x and y threadId in grid
  int x = (blockIdx.x * blockDim.x) + threadIdx.x;
  int y = (blockIdx.y * blockDim.y) + threadIdx.y;

  // Check that threadId's element inside matrix
  if(x > dim || y > dim)
    return;

  // Calculate position in matrix array a
  int idx = x*dim+y;

  res[idx] = a[idx]*a[idx]*a[idx];
}
```

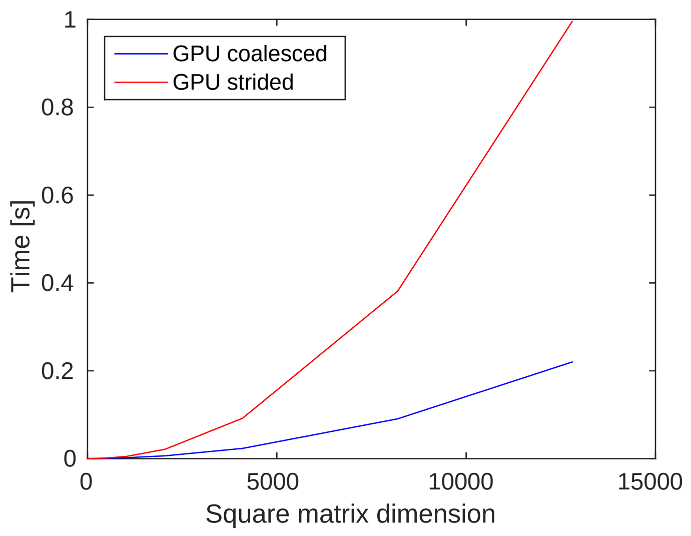

Figure 3.11: Graph showing the performance differences for coalesced vs. strided memory access for an elementwise matrix operation.

order.

The difference in performance of the two example kernels can be seen in Figure 3.11. The time measurements include both the cost of memory transfer from host to device, device to host, and computation. The figure illustrates that the strided approach has a poorer performance than the coalesced one. This leads us to our second design principle.

Design Principle 2. *Coalesce memory accesses, especially for global memory accesses.*

3.4 Summary

This chapter introduced the CUDA programming model and illustrated the computational advantage that can be achieved using GPUs over CPUs in an experiment. The CUDA memory model was presented, and the importance of achieving coalesced memory accesses was shown experimentally. Chapters 4 and 5 use the techniques presented in this chapter and show how the SGD, RBM, and ALS-WR algorithms where implemented using CUDA.

4 Hogwild and RBM: Design and implementation

In this chapter we describe key aspects of implementing CF recommender systems on NVIDIA GPUs. We explore both the matrix factorization and the model-based classes of the taxonomy in Section 2.5. The implementation of two algorithms, one in each class, is described. The Hogwild algorithm was chosen to represent the matrix factorization class, whereas the CUDA RBM approach was chosen to represent the model-based class of recommender systems.

Generally, the high-level architecture of all the recommender systems included in this and the following chapter consists of two components: a host and a CUDA device, as illustrated in Figure 4.1. The host is responsible for loading the training data, converting it into the correct format, and performing sequential operations. The CUDA device performs most of the computational operations required to train the model. Given a set of ratings, the recommendation model produces recommendations for the missing values in the rating matrix.

4.1 Hogwild

The design in this section is based on the work done by Recht et al., 2011 [44], who proposed the Hogwild approach to parallelizing the SGD. Hogwild is a lock-

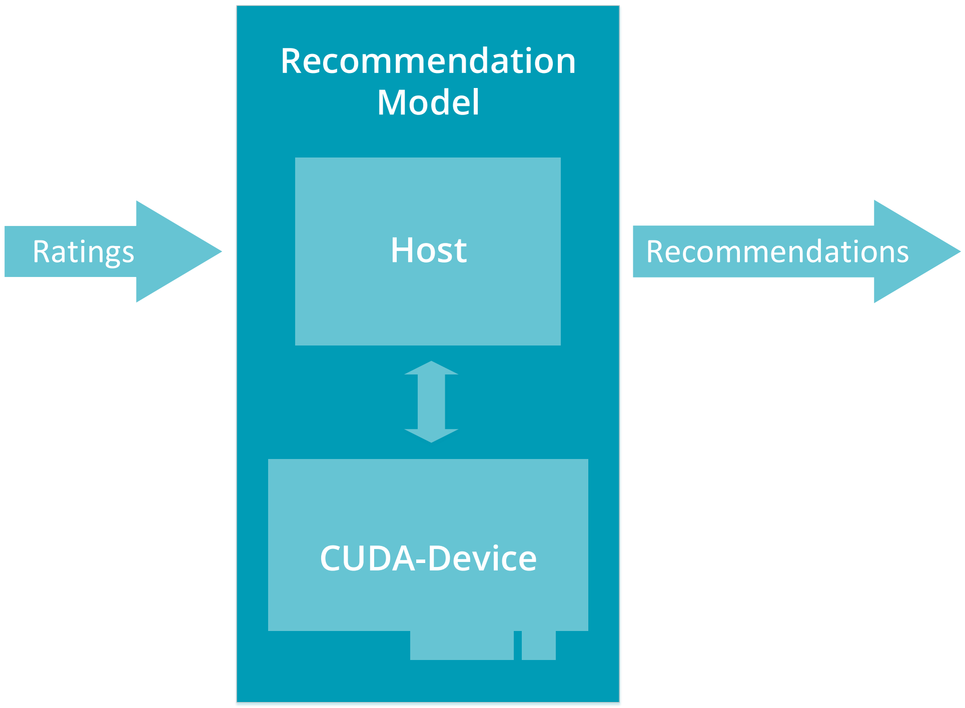

Figure 4.1: The general architecture of the CUDA based recommender systems.

free approach for parallelizing the SGD on shared memory multi-core systems. It is based on the idea that, given a sparse matrix, the SGD will converge even without the use of locking, which means that processes are allowed to update the weights at will and may overwrite progress made by others while doing so. Due to the overwriting occurrences some of the progress will be lost during the training process, which means that usually more update steps are required to achieve convergence. However, as more weight updates can be performed in a given timespan, convergence speedups are observed as the parallelism increases. A more detailed analysis of the scalability and performance of the Hogwild algorithm on CUDA can be found in our earlier work [26].

The pseudo code for the learning algorithm for the SGD is displayed in Algorithm 4.1. It illustrates the iterative nature of the algorithm, which in the CUDA implementation has been performed running massively parallel (Hogwild), by parallelizing the learning phase (line 2-10).

4.1.1 Parallelization

The learning phase is parallelized by calculating the feature vector updates and performing the update on the CUDA-device. Figure 4.2, illustrates how the

Algorithm 4.1 – The Hogwild SGD algorithm.

```
initialize random weights
for each epoch do
  for each training rating $r_{i,j}$ do
    $u_i$ <- load feature vector for user
    $v_j$ <- load feature vector for item
    $\Delta u_i = (u_i^T v_j - r_{i,j}) v_j + \lambda_1 u_i$
    $\Delta v_j = (u_i^T v_j - r_{i,j}) u_i + \lambda_2 v_j$
    $u_i = u_i - \mu \Delta u_i$
    $v_j = v_j - \mu \Delta v_j$
  end
end
```

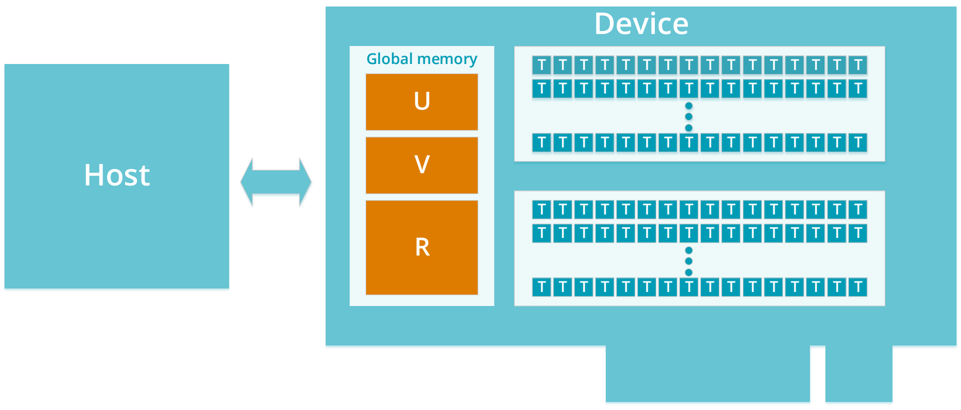

Figure 4.2: Abstract view of the Hogwild CUDA implementation.

matrices U, V, and R are stored in global memory. The figure also shows the grid-layout for the CUDA threads, by illustrating two blocks in the grid. Each row in the blocks is responsible for performing a single training update.

As the rating matrix in recommender systems is usually very sparse, the matrix is stored in the Coordinate List (COO) matrix format, where the representation consists of three arrays—one representing the rating value, one the row index, and one the column index. For example, the rating matrix

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

will in the COO matrix format be represented as

$$\begin{aligned} \text{data} &= [\ 1\ \ 1\ \ 4\ \ 3\ \ 5\ \ 1\ \ 1\ \ 2\ \ 3\ \ 5\ ] \\ \text{row} &= [\ 0\ \ 0\ \ 0\ \ 1\ \ 2\ \ 2\ \ 2\ \ 3\ \ 4\ \ 4\ ] \\ \text{col} &= [\ 0\ \ 1\ \ 2\ \ 1\ \ 0\ \ 1\ \ 2\ \ 2\ \ 0\ \ 2\ ] \end{aligned}$$

The sparseness observation for recommender systems, leads to our third design principle.

Design Principle 3. *Utilize sparse data representations to make efficient use of the limited device memory and reduce the data transfer overhead.*

As argued in Section 3.3, it is important to consider the layout of data in memory when performing CUDA computations. Figure 4.2 illustrates that the data stored in global memory consists of the user-matrix U, the item-matrix V, and the rating matrix R. Accesses to user- and item-matrices corresponds to reading a given user's or a given item's feature vector. To enable coalescent reads and writes, the matrices must be stored vector-wise in memory.

Another important factor to consider when reading the feature vectors from global memory, is that reads and writes must be performed as a transaction in order to avoid inconsistent reads and writes. As CUDA memory transactions are performed per warp, it is only possible to guarantees consistent global reads and writes for feature vectors up to 32 dimensions. For a larger number of dimensions, locking would be required. However, as the number of dimensions is often chosen to be reasonably small to avoid overfitting, this issue does not often occur.

4.2 RBM-CF

Another promising approach to collaborative filtering is the RBM-CF algorithm proposed by Salakhutdinov et al. in 2007 [46] as described in this thesis in Section 2.4. Additionally, the work presented in this section is based on the work done by Cai et al., 2012 [6], who proposed a parallel implementation of the RBM-CF algorithm on GPUs.

As seen in Section 2.4, the conditional probabilities of the visible and hidden units are given as

$$p(v_i^k = 1|\boldsymbol{h}) = \frac{e^{a_i^k + \sum_j h_j W_{ij}^k}}{\sum_{l=1}^{K} e^{a_i^l + \sum_j h_j W_{ij}^l}} \tag{4.1}$$

and

$$p(h_j = 1|\boldsymbol{V}) = \sigma(b_j + \sum_i \sum_{k=1}^{K} v_i^k w_{ij}^k) \,, \qquad (4.2)$$

and the weight update is computed as

$$\Delta W_{ij}^k = \mu(\langle v_i^k h_j \rangle_{\text{data}} - \langle v_i^k h_j \rangle_{\text{T}}). \qquad (4.3)$$

Additionally, the visible bias updates are computed as

$$\Delta a_i = \mu(\langle v_i \rangle_{\text{data}} - \langle v_i \rangle_{\text{T}}) \qquad (4.4)$$

and the hidden bias updates as

$$\Delta b_j = \mu(\langle h_j \rangle_{\text{data}} - \langle h_j \rangle_{\text{T}}). \qquad (4.5)$$

Using this underlying model, the general algorithm will be introduced in Section 4.2.1 before Section 4.2.2 illustrates how it was parallelized using CUDA.

4.2.1 Algorithm

The learning algorithm for the RBM can best be illustrated by use of pseudo code, which can be found in Algorithm 4.2.

It illustrates that the learning is performed iteratively on mini-batches, where each mini-batch corresponds to the vectors of rated movies for multiple users. For each of the mini-batches the hidden units are computed using Equation 4.2, before the n^{th} reconstruction is found using Gibbs sampling and utilizing equations 4.1 and 4.2. The final step is to use the statistics found by performing the Gibbs sampling to update the weight and bias terms using equations 4.3, 4.4, and 4.5. The most time consuming calculations consist of the matrix multiplications in lines 6, 12, 15, and 18 ($\overline{h}^k$,$\overline{h}^{k'}$, $\overline{v}^{k'}$ and ΔW^k) of Algorithm 4.2. These have speedup potential utilizing GPUs [6]. Table 4.1 provides an overview over the matrix multiplication operations.

Algorithm 4.2 – The RBM-CF algorithm.

```
initialize random bias and weights
for each epoch do
  for each mini-batch do
    construct $\overline{v}^1$ to $\overline{v}^K$
    for k=1 to K do
      $\overline{h}^k = \mathrm{tr}(W^k)\overline{v}^k + \overline{b}$
    end
    $\overline{h} = \sigma(\sum_{k=1}^{K} \overline{h}^k)$
    $\overline{h}' = \overline{h}$
    for each Gibbs sampling step do
      for k=1 to K do
        $\overline{v}^{k'} = W^k \overline{h}' + \overline{a}^k$
      end
      $\overline{v}^{k'} = exp(\overline{v}^{k'}) / \sum_{k=1}^{K} exp(\overline{v}^{k'})$
      compute $\overline{h}'$ as above using $\overline{v}^{k'}$
    end
    for k=1 to K do
      $\Delta W^k = \overline{v}^k \mathrm{tr}(\overline{h}) - \overline{v}^{k'} \mathrm{tr}(\overline{h}')$
      $W^k = W^k + \mu \times \mathrm{avg}(\Delta W^k)$
      update $\overline{a}^k$
    end
    update $\overline{b}$
  end
end
```

	Left		Right		Product	
	Symbol	Dim	Symbol	Dim	Symbol	Dim
Hidden Update	$\mathrm{tr}(W^k)$	$J \times I$	$\overline{v}^k$	$I \times M$	$\overline{h}^k$	$J \times M$
Visible Update	W^k	$I \times J$	$\overline{h}^{k'}$	$J \times M$	$\overline{v}^{k'}$	$I \times M$
Weight Update	$\overline{v}^k$	$I \times M$	$\mathrm{tr}(\overline{h}^{k'})$	$M \times J$	ΔW^k	$I \times J$

Table 4.1: Overview over the three matrix operations of the RBM-CF.

4.2.2 Parallelization

To improve the parallelization of this algorithm even further, the sparsity of the visible units $\overline{v}^k$ should be taken into consideration according to Design Principle 3. To do this, $\overline{v}^k$ is represented in a sparse format, which consists of a mixture of three commonly used sparseness formats. This section will describe the process of constructing $\overline{v}^k$.

Data format

For a set of M users in a batch with I items—potentially rated per user—the rating matrix is an $I \times M$ matrix. For three users in a batch with five items in the system, the rating matrix might look like

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

Using this example, the first part of Figure 4.3 illustrates how the rating vector for user 1 (first column) can be converted to a binary representation, where each column corresponds to one item in the original rating vector and each row denotes what rating a user has given to the item. Items that have not received a rating are represented by an x. Using this representation to find v^1, the binary rating vector for all items that received a rating with $k = 1$ can be produced by selecting the first row (highlighted in the figure). The $I \times M$ binary batch matrix $\overline{v}^1$ is found by repeating this for all users.

However, as previously noted, these matrices ($\overline{v}^k$), will be very sparse and should therefore be represented as sparse matrices. This is done by using a

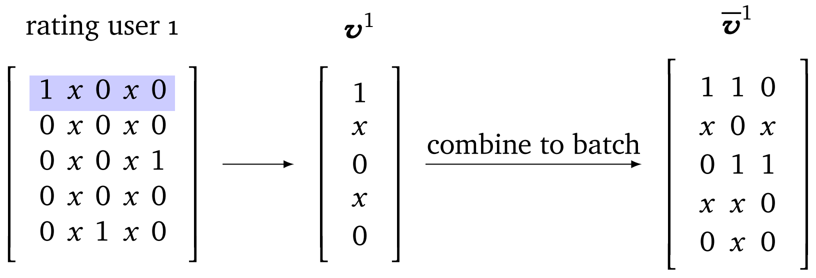

Figure 4.3: Illustration of the batch dataset generation.

sparse matrix format that combines the advantages of the Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and the COO matrix formats. The joint sparse matrix format is used to support coalesced reads and writes as part of the three different matrix multiplications and is motivated by Design Principle 2. The CSR and CSC formats both consist of three arrays each: a data array, an index array and an index pointer array. The example rating matrix would, in the CSR format, be represented as

$$\begin{aligned}\text{data} &= [\ 1\ \ 1\ \ 4\ \ 3\ \ 5\ \ 1\ \ 1\ \ 2\ \ 3\ \ 5\ ]\\ \text{indices} &= [\ 0\ \ 1\ \ 2\ \ 1\ \ 0\ \ 1\ \ 2\ \ 2\ \ 0\ \ 2\ ]\\ \text{indptr} &= [\ 0\ \ 3\ \ 4\ \ 7\ \ 8\ \ 10\ ]\end{aligned}$$

and in the CSC format as

$$\begin{aligned}\text{data} &= [\ 1\ \ 5\ \ 3\ \ 1\ \ 3\ \ 1\ \ 4\ \ 1\ \ 2\ \ 5\ ]\\ \text{indices} &= [\ 0\ \ 2\ \ 4\ \ 0\ \ 1\ \ 2\ \ 0\ \ 2\ \ 3\ \ 4\ ]\\ \text{indptr} &= [\ 0\ \ 3\ \ 6\ \ 10\ ]\end{aligned}$$

The process of computing this sparse format for the previous example of $\overline{v}^1$ is illustrated in Figure 4.4. As the data array is being updated as part of the visible unit update computation, it is advantageous to only represent the data (binary format of rating values) in one form. This avoids the need of keeping multiple arrays consistent. To be able to access the data in CSR matrix format, which is required when computing the weight updates, it is necessary to represent the data using a mapping array. This array maps the data array of the CSR matrix format to the data array of the CSC format. As can be seen in Figure 4.4, the sparse format does include all the 0-values in addition to

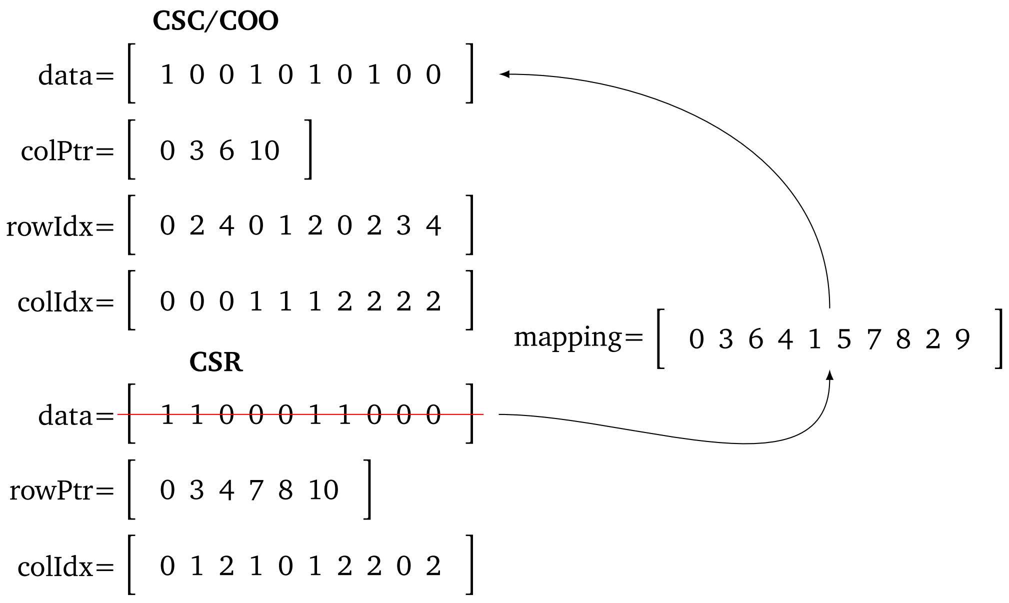
CSC/COO
data= [1 0 0 1 0 1 0 1 0 0]
colPtr= [0 3 6 10]
rowIdx= [0 2 4 0 1 2 0 2 3 4]
colIdx= [0 0 0 1 1 1 2 2 2 2]
mapping= [0 3 6 4 1 5 7 8 2 9]
CSR
data= [~~1 1 0 0 0 1 1 0 0 0~~]
rowPtr= [0 3 4 7 8 10]
colIdx= [0 1 2 1 0 1 2 2 0 2]

Figure 4.4: Figure that shows the conversion of the batch dataset matrix to the sparse matrix format.

1-values (not the x-values). This is done to be able to use the row/column indices, row/column pointers, and mapping arrays for all $\overline{v}^k$. An additional benefit is that the indices and arrays do not have to be changed when computing the reconstructions during the Gibbs sampling procedure. To increase the computation/communication ratio according to Design Principle 1, the author made the design choice to store the row, column and mapping indices on the device during batch generation and keep them there until the model has been updated for the batch. This avoids unnecessary data transfer.

Hidden units update

The $\overline{h}^k = \mathrm{tr}(W^k)\overline{v}^k + \overline{b}$ part of the hidden unit update has been parallelized using CUDA and the CSC part of the sparse matrix representation of $\overline{v}^k$. To achieve the parallelization of the matrix multiplication, each column in the result matrix is represented by a thread block, with each thread being responsible for exactly one hidden node in the column. By leveraging the CSC format in this manner, the entries of $\overline{v}^k$ are read coalesced. Because $\overline{v}^k$ is restricted to being either 1 or 0, the inner product computation of W^k and $\overline{v}^k$ can be expressed as a summation for increased performance.

Visible units update

The $\overline{v}^k = W^k\overline{h}+\overline{a}$ expression of the visible unit update has been parallelized using CUDA. In this case $\overline{v}^k$ was represented by the COO format and one thread block is assigned to each missing entry in $\overline{v}^k$. The threads in the block perform the inner product computation between W_i^k and $\overline{h_j}$, and parallel reduction is used to sum up the individual thread results to efficiently perform the computation for large numbers of hidden units. As for the hidden units, the multiplication is expressed as a summation since $\overline{h_j}$ can only be 1 or 0.

Weight update

The third matrix operation computes $\Delta W^k = \overline{v}^k \text{tr}(\overline{h})$ and $\Delta W^k = \overline{v}^{k'} \text{tr}(\overline{h}')$ by using the CSR format together with the mapping information present in the mapping array. One thread block is used to compute each row of the product matrix ΔW^k, with threads computing elements in the row. Unlike the visible and hidden unit update scenario, it can not be assumed for the weight update that one of the matrices that is getting multiplied contains only ones and zeros. This means that the optimization where the multiplication is transformed into an addition cannot be performed.

Data transfer

To increase the performance of the algorithm the author considered storing and performing the weight updates (line 18 and 19 in Algorithm 4.2.1) on the device to avoid the transfer overhead in all three update computations (Design Principle 1). The drawback of this approach is that for large datasets and large batches the k weight matrices can be too large to store on the device, which might be why it was not suggested by Cai et al. [6]. However, for the datasets used in this thesis the weight matrices where small enough and the difference in performance was investigated. Both versions of the algorithm were run for one epoch. The dataset used, is a movie dataset from Douban.[1] It has roughly 130,000 users, which where run in batches of 100. A detailed overview over the dataset can be found in Section 6.1. Figure 4.5 illustrates the results of those test runs. By storing and keeping the weight matrices in device memory, the total time for one epoch was halved compared to the original implementation. Similar improvements can be observed for each of the different updates. We note, that for datasets that are small enough to keep the weight matrices in memory a considerably speedup can be achieved for the RBM-CF.

1. douban.com

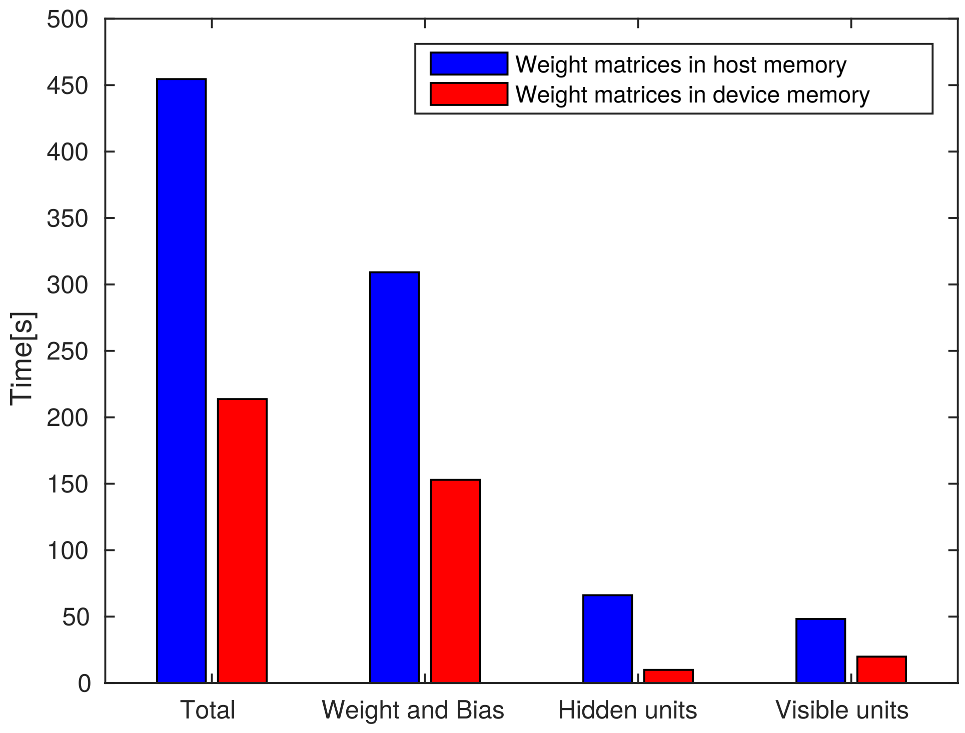

Figure 4.5: Performance gain by keeping the weight matrices in device memory. The time it takes to perform one epoch is measured for both implementations to illustrate the effect of keeping the weight matrices in global memory. Besides the total time, the time for the three update computations was measured.

4.3 Summary

This chapter provided the reader with an understanding of how the SGD and RBM algorithms were implemented and optimized using CUDA. In Chapter 6, we will evaluate the two algorithms and compare the results with the Alternating-Least-Squares with Weighted-λ-Regularization on CUDA (ALS-CUDA) implementation, which is presented in the next chapter.

5 ALS-CUDA: Design and implementation

This chapter describes the design and implementation choices that where made to implement the Alternating-Least-Squares with Weighted-λ-Regularization on CUDA (ALS-CUDA) algorithm. The ALS-CUDA algorithm is based on the ALS-WR algorithm discussed in Section 2.3.2, in particular the two equations

$$u_i = (V_{I_i} V_{I_i}^T + \lambda n_{u_i} E)^{-1} V_{I_i} R^T(i, I_i) \tag{5.1}$$

and

$$v_j = (U_{I_j} U_{I_j}^T + \lambda n_{v_j} E)^{-1} U_{I_j} R^T(i, I_j) \ , \tag{5.2}$$

where u_i and v_j refer to the i^{th} and j^{th} row and column in the result matrix, R, respectively. Given the weight parameter matrix U of user feature vectors, U_{I_j} denotes the sub-matrix of U that contains only the user feature vectors of users that have rated item j. Similarly, given the weight parameter matrix V of item taste vectors, V_{I_i} denotes the sub-matrix of V that contains only the items that the user i has rated. E is the $dim \times dim$ identity matrix, and n_{u_i} and n_{v_j} are the number of items a user i has rated, and the number of users who have rated a given item j, respectively. dim is the dimensionality of the latent feature space that the users and items are mapped to.

The pseudo code of the ALS-WR algorithm can be seen in Algorithm 5.1. Each epoch consist of an U and V update, where the equations 5.1 and 5.2 are used to compute them. Each of the individual update computations includes the construction of the sub-matrices U_{I_j} and V_{I_i}.

The data format used to represent the rating matrix is a combination of the CSC and CSR matrix representation format. Unlike in the RBM-CF algorithm, no mapping array was used to represent the two data arrays (see Section 4.2.2), as the data in the rating matrix is constant during training. The data format was chosen to allow efficient construction of the sub-matrices by using the CSR index pointer array to find the number of ratings for a given user and to retrieve I_i. Using the CSC index pointer array the number of ratings for a given item can be found and the vector I_j can be retrieved efficiently (Design Principle 3).

The matrices U and V, and the sparse rating matrix R are transferred to the device before the first update calculation and are updated in device memory. This means, that no data has to be transfered between the updates, causing a high computation/communication ratio (Design Principle 1).

The update equations 5.1 and 5.2 can be divided up into two main operations. The first one is the computation of the matrix multiplication and the addition of the regularization term ($V_{I_i}V_{I_i}^T + \lambda n_{u_i}E$ and $U_{I_j}U_{I_j}^T + \lambda n_{v_j}E$). Two main challenges arise when implementing the matrix multiplication, which can be solved by making use of a dynamic matrix multiplication. This is discussed in Section 5.1.

Once the first main operation has been performed and returned a result matrix A, the second operation is the computation of the expression $A^{-1}V_{I_i}R^T(i, I_i)$ or $A^{-1}U_{I_j}R^T(i, I_j)$. As matrix inversion is very costly, the operation is performed by solving the expression as a linear system of equations, which is discussed in Section 5.2.

5.1 Dynamic Batch Matrix Multiplication

One problem for the CUDA implementation is the fact that for the $V_{I_i}V_{I_i}^T$ and $U_{I_j}U_{I_j}^T$ matrix multiplications, the sub-matrices V_{I_i} and U_{I_j} are of varying size, but have to be performed for each column in V and each row in U. The ability to perform matrix multiplications in parallel is critical for the performance of the algorithm. Hence, a kernel must be designed that can perform all multiplications in one batch. The second design issue with the matrix multiplication

Algorithm 5.1 – The ALS-WR algorithm.

```
initialize random weights
for each epoch do
  for each user $i$ in U do
    $V_{I_i}$ <- construct sub-matrix for user $i$
    $u_i = (V_{I_i} V_{I_i}^T + \lambda n_{u_i} E)^{-1} V_{I_i} R^T(i, I_i)$
  for each item $j$ in V do
    $U_{I_j}$ <- construct sub-matrix for item $j$
    $v_j = (U_{I_j} U_{I_j}^T + \lambda n_{v_j} E)^{-1} U_{I_j} R^T(i, I_j)$
  end
end
```

arises because the size of the matrices V_{I_i} and U_{I_j} are dependent on the number of users who have rated an item j, or on the number of items that a given user i has rated. Since CUDA does not support dynamic allocation of memory during kernel executions, the varying size of the matrices can be a problem. To circumvent this limitation, we exploit mathematical properties of the two matrix multiplications. As the matrix multiplication involves a matrix being multiplied by its transpose, the resulting matrix will be a square matrix of $dim \times dim$ dimensions, where dim is the dimension of the user and item feature vectors. The other advantage of ALS is that the result of the multiplication will be a symmetric matrix B such that

$$B = B^T. \tag{5.3}$$

This means that the result matrix can be fully represented by $dim \times (dim + 1)/2$ instead of $dim \times dim$ elements. As dim is known, the required space in global memory can be allocated prior to kernel execution. This assumption of symmetry is still valid after the regularization term $\lambda n_{u_i} E$ is added, as it, due to the identity matrix E, only modifies the values on the diagonal.

Using the matrix A to represent either V_{I_i} (or U_{I_j}), where V_{I_i} (or U_{I_j}) is a $dim \times n_{u_i}$ (or $dim \times n_{v_j}$) matrix, the matrix multiplication can be expressed as

$$\begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \times \begin{pmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$$

In global memory, the matrices V and U are stored feature-vector wise as displayed in Figure 5.1.

This memory layout means that coalesced memory accesses cannot be achieved by performing the standard matrix multiplication, which can incur large over-

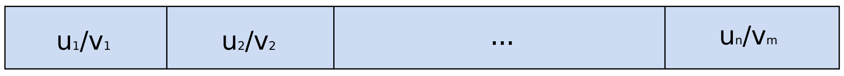

Figure 5.1: Memory layout for the ALS-CUDA algorithm. Both weight matrices are stored vector-wise.

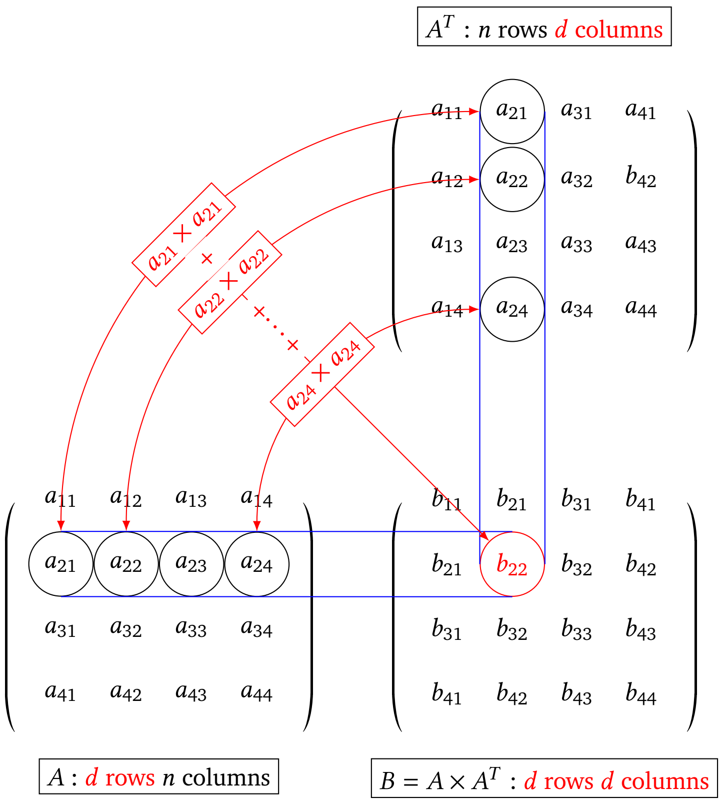
A^T : n rows d columns
$a_{21} \times a_{21}$
+
$a_{22} \times a_{22}$
+ ... +
$a_{24} \times a_{24}$
A : d rows n columns
$B = A \times A^T$: d rows d columns

Figure 5.2: Illustration of the matrix multiplication for diagonal entries when a matrix is multiplied with its transpose.

heads (Design Principle 2). Figure 5.2 displays the process of matrix multiplication for the diagonal entries and Figure 5.3 for the non-diagonal elements. These figures illustrate how an entry in the result matrix B can be found, and the effect of the symmetry. It can be seen that in the scenario where A is in row-major order, reading rows in A can be performed coalesced (which corresponds to a column in A^T). However, as matrix A in the ALS-CUDA scenario needs to be stored in column-major order (one column represents one user/item feature vector), this cannot be done as easily.

To utilize coalesced reads, the implementation presented here uses a different pattern to compute the matrix multiplications. A column (size *dim*) is read coalesced from global memory (either from V or U), and is placed in shared memory to allow all threads in the warp (and block) fast access. The element that each thread read is stored in the threads local memory.

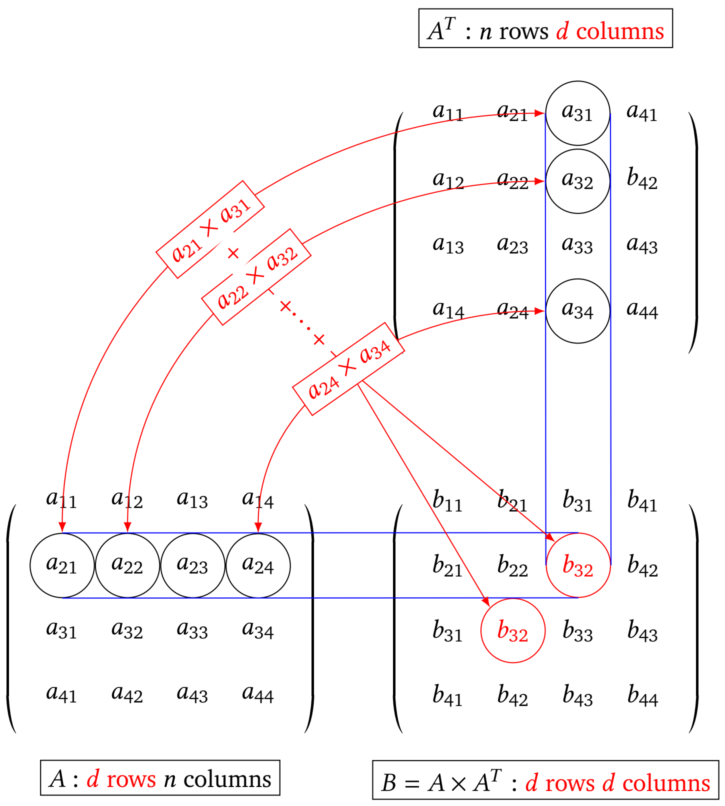
A^T : n rows d columns
$a_{21} \times a_{31}$
+
$a_{22} \times a_{32}$
+ . . . +
$a_{24} \times a_{34}$
A : d rows n columns
$B = A \times A^T$: d rows d columns

Figure 5.3: Illustration of the matrix multiplication for off-diagonal entries when a matrix is multiplied with its transpose.

Using the data column we can compute one part (term) of each of the elements in result matrix B. This is done by initially multiplying each of the values in local memory with themselves, which yields a summation term for the diagonal entries, as illustrated in Figure 5.4. Using the rest of the elements from the column, which are now stored in shared memory, each thread can compute the other combinations of its local memory entry and all the others. This way the summation term for the non-diagonal terms can be computed as illustrated in Figure 5.5.

As the result matrix is symmetric, only the lower (or upper) half of the non-diagonal elements need to be computed explicitly. This means that not all permutations, but all *combinations* have to be found, resulting in $\binom{dim}{2}$ combinations for non-diagonal terms and *dim* for for the diagonal terms. As dim

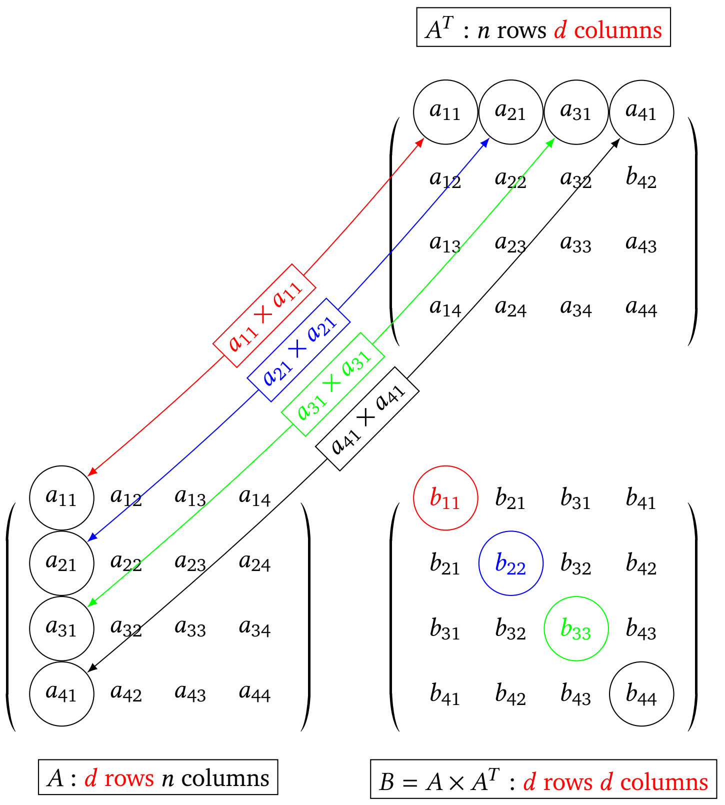
A^T : n rows d columns
$a_{11} \times a_{11}$
$a_{21} \times a_{21}$
$a_{31} \times a_{31}$
$a_{41} \times a_{41}$
A : d rows n columns
$B = A \times A^T$: d rows d columns

Figure 5.4: Finding the first part of the diagonal entries for the modified matrix multiplication procedure. The second column in matrix A will yield another term for each of the diagonal entries in B. All four terms are found by repeating the procedure for all four columns in A.

threads are performing the calculation, each thread is performing at most

$$\begin{aligned}\left\lceil \frac{\left(\binom{dim}{2} + dim\right)}{dim} \right\rceil &= \left\lceil \frac{\left(\frac{dim!}{2!(dim-2)!}\right)}{dim} \right\rceil + 1 \\ &= \left\lceil \left(\frac{(dim-1)!}{2!(dim-2)!}\right) \right\rceil + 1 \\ &= \left\lceil \frac{dim+1}{2} \right\rceil \end{aligned} \tag{5.4}$$

calculations per column. If $\binom{dim}{2} + dim$ is not divisible by dim some of the threads will require one operation less than stated in the equation. This means that for a column of size dim, each thread performs $O(dim)$ operations.

Once all combinations have been evaluated and the results have been written to the correct place in shared memory, the next column is loaded and the same

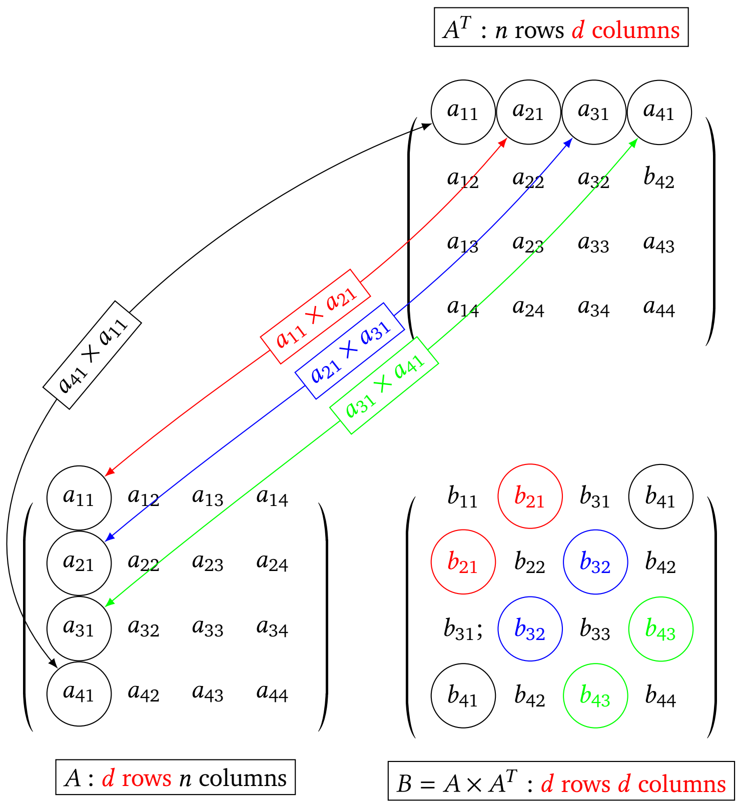
A^T : n rows d columns
$a_{41} \times a_{11}$
$a_{11} \times a_{21}$
$a_{21} \times a_{31}$
$a_{31} \times a_{41}$
A : d rows n columns
$B = A \times A^T$: d rows d columns

Figure 5.5: Finding the first part of some of the off-diagonal entries for the modified matrix multiplication procedure. The figure illustrates the first combinations of the first column in A. Computing all the combinations will yield one term for each of the off-diagonal elements. Repeating it for all four columns will yield four terms for all off-diagonal terms, which when added together yield the result matrix.

process is performed, thus enabling coalesced reads for all the memory accesses. This process is repeated n_{u_i} or n_{v_j} times for V_{I_i} and U_{I_j} respectively to compute the whole matrix.

However, to combine the products, e.g. $a_{21} \times a_{31}$ in Figure 5.5, we need to find the correct update position in the result matrix B. The correct row position can be found by using the current thread id, which corresponds to the row number in matrix A. In our example, this would be 2. The correct column position can be found by the shared memory array location that the current thread reads the second product term from. This corresponds to the column in A^T, which in our example is 3. Ensuring that for the symmetric entries only the elements with a row number lower than their column number are stored, the index in the symmetric row-major matrix can be found as

$$idx = (row - 1)dim + col - \frac{(row-1)row}{2} - 1\,, \tag{5.5}$$

where symmetric elements are in the form that $col \geq row$.

A pseudo code for the dynamic batch matrix multiplication is illustrated in Algorithm 5.2.

Algorithm 5.2 – The dynamic batch matrix multiplication.

```
// Read rowIdx in matrix A (in range [0,dim-1])
rowIdx = getThreadId()
for each column in sub-matrix A do
  // Read threads first product term
  threadValue = column[rowIdx]
  // Calculate maximum number of operations per thread
     using Equation 5.4
  nrOps = getNrOps(dim)
  for operationIdx in range(nrOps) do
    // Check if the current combination has to be
       calculated, handles the case of $\binom{dim}{2} + dim$ not
       being divisible $dim$
    if(combinationNr < totalCombinationNr)
      columnIdx = (operationIdx + rowIdx) \% dim
      updateTerm = threadValue * column[columnIdx]
      //Find index in symmetric matrix array using
         Equation 5.5
      idx = getSymIdx(min(row,col) + 1, max(row,col) +
         1, dim)
      symMatrix[idx] += updateTerm
  end
end
```

5.2 Solving the Linear System of Equations

Once the vector multiplication ($V_{I_i}R^T(i, I_i)$ and $U_{I_j}R^T(i, I_j)$) and the matrix multiplication has been performed, it is possible to view the least square problem as a linear equations system of type $Ax = b$, where b refers to the vector and A to the summation of the matrix multiplication and the regularization term. Many methods exist for solving such a system, for example Gauss-Jordan [1]. However, using the fact that the matrix multiplication will always result in a symmetric matrix it is possible to half the operations required by utilizing a

LDL^T decomposition. The faster Cholesky [42] decomposition could not be used, as the matrix is not guaranteed to be positive definite, which is the assumption that the Cholesky decomposition bases itself on.

5.2.1 LDL$^{\mathrm{T}}$ Decomposition

In the LDL^T decomposition, a matrix A gets decomposed into a lower matrix L and a diagonal matrix D such that

$$A = LDL^T \tag{5.6}$$

with L's diagonal terms being

$$L_{ii} = 1. \tag{5.7}$$

For a square matrix with dimension $dim = 3$, the decomposition would be

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{pmatrix} \times \begin{pmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{pmatrix} \times \begin{pmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{pmatrix}$$

where the entries for the D matrix can be found by

$$D_{jj} = A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2 D_{kk} \tag{5.8}$$

and the entries for L by

$$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} D_{kk} L_{jk}. \tag{5.9}$$

Once the decomposition has been performed, the system of linear equations can be solved by a forward substitution followed by a backward substitution.

5.2.2 Forward Substitution

Once the LDL^T decomposition has been performed the linear system can be written as

$$LDL^T x = b \ , \tag{5.10}$$

where matrix A has been replaced by its decomposition.

Replacing $DL^T x$ with a new vector variable z, it is possible to rewrite equation 5.10 as the new system of linear equations

$$Lz = b \ . \tag{5.11}$$

As L is a lower matrix, solving for z becomes a trivial task and can be done by

$$z_i = b_i - \sum_{k=1}^{i-1} L_{ik} z_k \ . \tag{5.12}$$

To find x, z needs to be solved with respect to x, which is referred to as the backward substitution step.

5.2.3 Backward Substitution

Utilizing the fact that D is a diagonal matrix, and L^T an upper matrix, it is possible to solve

$$z = DL^T x \tag{5.13}$$

with respect to x by scaling z by D. As D is a diagonal matrix, the result vector y can be found as

$$y_i = \frac{z_i}{D_{ii}} \tag{5.14}$$

with

$$L^T x = y \ . \tag{5.15}$$

As L^T is an upper matrix, the vector x can be found by

$$x_i = y_i - \sum_{k=i+1}^{N} L_{ki} x_k \text{ for } N, N-1, \ldots, 1 \ , \tag{5.16}$$

solving the original linear set of equations and resulting in the updated U_i or V_j feature vector.

Algorithm 5.3 – The ALS-CUDA algorithm.

```
for each row in U (or V) do

  // Use CSR format of rating matrix to find start and
      end of $I_i$ (or CSC format to find $I_j$)
  startIdx = getStartRowIndices()
  endIdx = getEndRowIndices()

  // Find number of ratings for user (or item), which
      corresponds to number of columns in matrix $A$
  nrColumns = endIdx-startIdx

  symMatrixSize = dim * (dim + 1) / 2

  // Compute symmetric matrix
  sparseMatrix = matrixMultiplication(nrColumns,
      ratingMatrix)

  // Solve the system of linear equations
  L,D = LDLTDecomposition(sparseMatrix, dim,
      symMatrixSize)
  z = forwardSubstitution(L, b, dim, symMatrixSize)
  x = backwardSubstitution(L, z, D, dim, symMatrixSize)

  // Update row in U (or V)
  row = x
end
```

5.3 Complete ALS-CUDA algorithm

Combining the procedure of solving the linear system of equations with the batch matrix multiplication algorithm, the ALS-CUDA algorithm can be expressed as illustrated in Algorithm 5.3. The *matrixMultiplication* function refers to the dynamic batch matrix multiplication algorithm described in Section 5.1 and illustrated in Algorithm 5.2. The functions in lines 16-18 follow the steps described in Section 5.2 and solve the linear system of equations.

5.4 Summary

This chapter introduced the ALS-CUDA algorithm. To optimize the ALS-WR algorithm on CUDA, a dynamic matrix multiplication algorithm was formulated that allows for memory coalesced reads and writes. It was optimized using the symmetry property. Additionally, the process of performing the matrix inversion by solving a linear equation system has been illustrated. The LDL^T decomposition was used to optimize the number of reads/writes required to perform the operation. In Chapter 6, the algorithms presented in this and the previous chapter are evaluated and compared.

6 Evaluation

This chapter describes the experimental setup and the datasets that have been used to evaluate the ALS-CUDA algorithm. The scalability of the ALS-CUDA algorithm is analyzed and its performance is compared to the SGD and RBM-CF implementation.

6.1 Experimental Setup

All experiments were run on a Dell Precision T3610 workstation with the following specifications: Intel Xeon(R) CPU E5-1620 v2 @ 3.70GHz×8, GeForce GTX 770 (1536 CUDA Cores), and 64GB DDR3 1866MHz.

The first dataset used as part of this evaluation is the Douban[1] movie dataset that was produced in [35]. Douban is a Chinese website that was launched in 2005 and allows users to rate and review books, movies and music, whilst also supporting social networking services and providing users with recommendations. In our evaluation, we will only utilize the movie ratings. The dataset consists of 129,490 unique users, 58,541 unique movies, and 16,830,839 ratings between 1 and 5. This corresponds to a sparsity[2] of 0.22% for the rating matrix.

1. douban.com
2. In recommender systems sparsity is the fraction of nonmissing elements over the total number of elements in the rating matrix.

Table 6.2 summarize the statistics of the dataset.

Table 6.1: Statistics of the Douban dataset.

	Minimum	Average	Maximum
Number of Ratings (Users)	1	129.98	6,328
Number of Ratings (Items)	1	287.51	49,504

The second dataset is from the Jester Online Joke recommender system, which is a research project from the UC Berkeley Laboratory for Automation Science and Engineering [19]. Jester was launched in November 1998 and gained popularity after being mentioned in Wired News magazine and on online news sites such as Yahoo, Excite and Netscape News [19]. The data used from the Jester dataset in this thesis includes ratings collected between April 1999 and May 2003, and consists of 100 jokes (items) that have been rated by 73,421 users. In total the dataset contains 4,136,360 ratings and has a sparsity of 56.34% for the rating matrix, which is unusually dense for a recommender system. An overview of the statistics of the Jester dataset can be found in Table 6.2. The ratings in the Jester dataset are continuous values between -10.00 and $+10.00$. To avoid negative and non-integer ratings, the ratings are normalized and discretized to an integer range of 1 to 5.

Table 6.2: Statistics of the Jester dataset.

	Minimum	Average	Maximum
Number of Ratings (Users)	15	56.34	100
Number of Ratings (Items)	18505	41363.60	73413

For our evaluations, the quality of the recommendations are measured using the mean absolute error (MAE) metric. The MAE is defined as

$$MAE = \frac{1}{T} \sum_{i,j \in S} \left| R_{i,j} - \hat{R}_{i,j} \right| \ , \tag{6.1}$$

where T is the total number ratings in the test dataset S. $R_{i,j}$ and $\hat{R}_{i,j}$ denote the actual and the models predicted rating for user i and item j, respectively.

The datasets that we use for our experiments are split into a training set and a test set. The training set consists of 80% of the ratings, and the test set of the remaining 20%.

6.2 Finding suitable free parameters

The learning phases of the presented algorithms in Chapter 4 and 5 rely on a set of learning parameters. For the SGD algorithm, two parameters are required—the learning rate and a regularization parameter—whereas the ALS-CUDA algorithm requires only the regularization parameter. The RBM-CF algorithm has the most parameters of the three algorithms, as it requires a learning rate, the number of hidden nodes, and the batch size to be specified. To compare the algorithms fairly, we must find suitable parameter for all algorithms and for each dataset. This section introduces the parameters for the different algorithms and presents the best parameters for each of the datasets.

6.2.1 Hogwild on CUDA

For the Hogwild algorithm the training samples presented to the algorithm were sampled with replacement from the training set and the weights were initialized randomly by sampling from a uniform distribution over $[0, 1)$. The ideal regularization parameter of the Hogwild algorithm was then found by performing two experiments: one for the Douban, and one for the Jester dataset. Figure 6.1 illustrates the results for the two experiments, where the MAE on the test dataset for a range of parameters is plotted. For each parameter setting the algorithm performed 7.6×10^8 updates and the best achieved MAE for the test dataset was selected. The figure shows that the best regularization parameter, λ, for the Douban dataset lies at 0.021. For the Jester dataset, the best regularization parameter was found to be $\lambda = 0.06$. The two experiments were repeated ten times for different training set realizations and the figure shows the mean MAE over those runs. The standard deviation was evaluated for each regularization parameter and the largest standard deviation was 1×10^{-3} and 3.06×10^{-4} for the Jester and Douban dataset, respectively. The observed small standard deviation indicates that the algorithm is robust to changes in the training dataset.

To find an optimal learning rate parameter for the Hogwild algorithm, we perform a similar experiment as the one for finding the regularization parameter, but this time adjusting the learning rate. The resulting mean MAE for the Douban and Jester dataset can be found in Figure 6.2. As can be seen, the optimal learning rate in Douban is 0.026, while in Jester it is 0.016. The largest standard deviation for this example was 1.5×10^{-3} for the Jester and 5.11×10^{-4} for the Douban dataset.

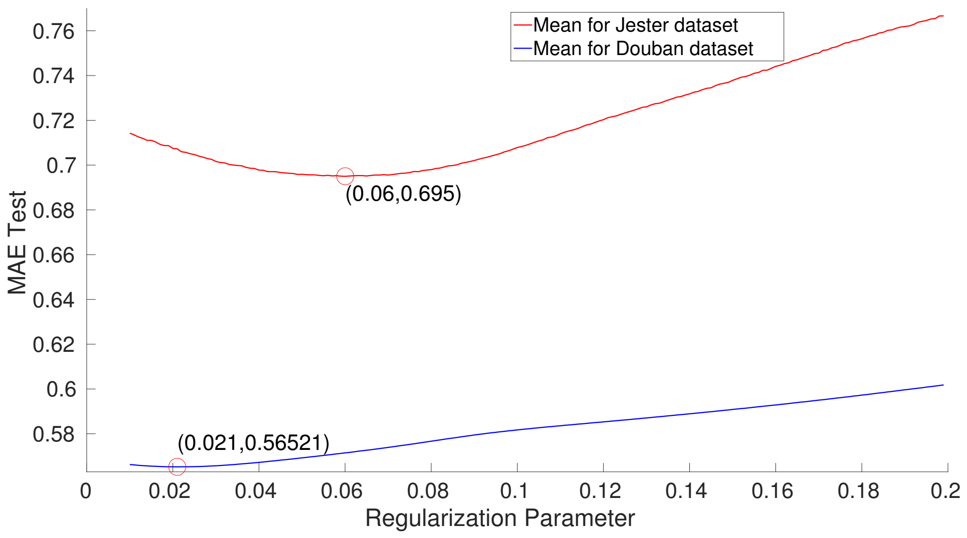

Figure 6.1: MAE for various regularization parameters λ, for the Hogwild algorithm.

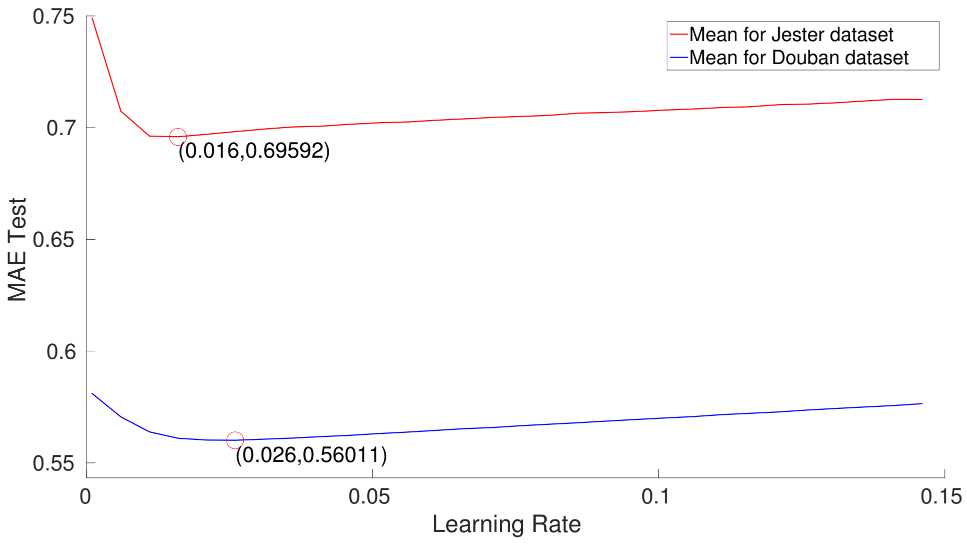

Figure 6.2: MAE for various learning rate parameters for the Hogwild algorithm.

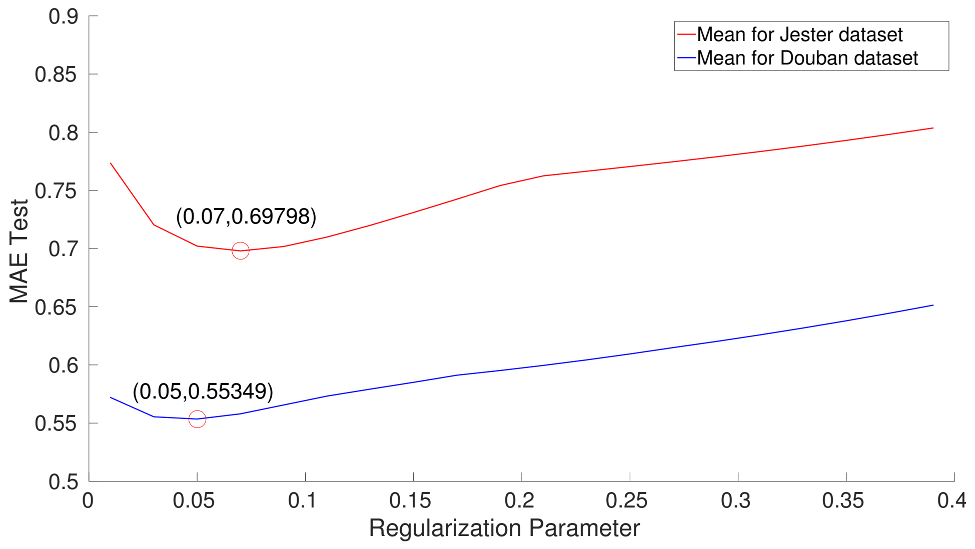

Figure 6.3: MAE for various regularization parameters λ, for the ALS-CUDA algorithm.

6.2.2 ALS-CUDA

Our ALS-CUDA algorithm requires only a regularization coefficient to be predefined. Two experiments have been performed to find the best regularization value. As with our Hogwild experiments, the weights for the ALS-CUDA algorithm were initialized randomly by sampling from a uniform distribution over $[0, 1)$. Figure 6.3 shows the best MAE for the Douban and Jester test dataset after 50 updates for a range of different regularization parameters. It can be seen that the optimal regularization parameters for the Douban and Jester dataset are $\lambda = 0.05$ and 0.07, respectively. Small variations in λ can yield considerable changes in accuracy. The largest standard deviation was found to be 2.12×10^{-4} for the Douban and 7.19×10^{-4} for the Jester dataset.

6.2.3 RBM-CF on CUDA

The RBM-CF requires us to set a learning rate, as well as two additional parameters: the number of hidden units and the batch size. The learning phase of the RBM-CF was found to be considerably slower than the learning phase of the ALS and SGD. To be able to evaluate the learning rate in a feasible way, only parts of the training and test dataset were used to find a good learning rate for the model. Léon Bottou [3] states that learning rates can be mathematically proven to be independent of the training set size for a stochastic gradient descent as long as the subset of data is a good representation of the full dataset. Using this reasoning, we split the rating matrix into a smaller subset, which was

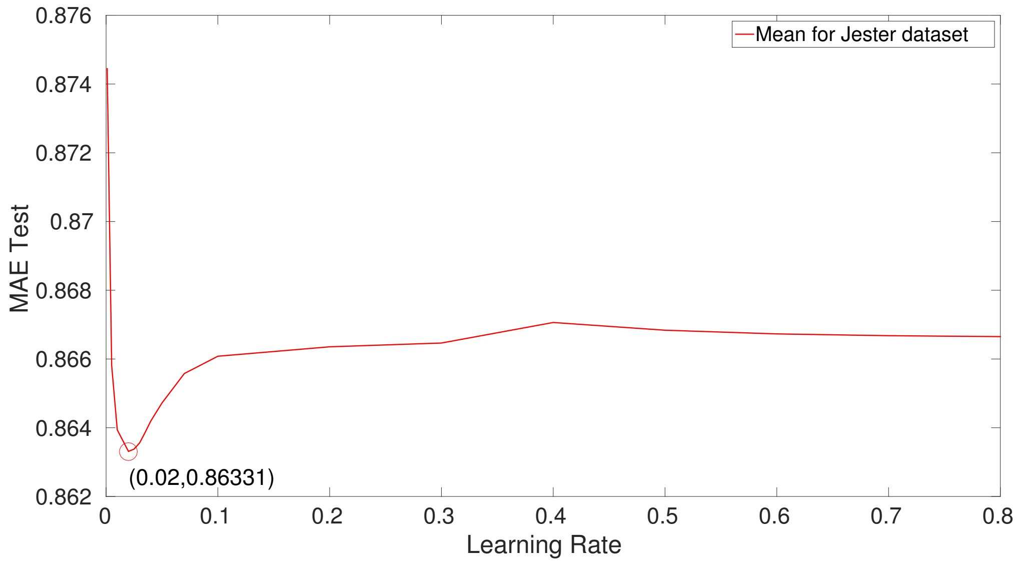

Figure 6.4: MAE for various learning rate parameters for the RBM-CF algorithm.

used for the learning rate experiment in this section. The initial weight values were sampled from a normal distribution with a standard deviation of 0.01 and zero-mean as suggested by Salakhutdinov et al. [46]. Figure 6.4 illustrates that, for the Jester dataset, the most progress towards convergence was made using a learning rate of 0.02. Due to high computational cost, the parameters were only evaluated on the smaller Jester dataset.

The method suggested by Léon Bottou [3] to evaluate the learning rate parameter on a small dataset can, however, not be used when considering the right number of hidden nodes or the batch size, as these are directly dependent on the size of the dataset. Instead, as parameter sweeps were unfeasible, we based our choice of parameters on the works of Cai et al. [6] and Salakhutdinov et al. [46], after confirming their findings in our implementation. The number of users per batch was chosen to be 100, whereas the number of hidden units was set to 128.

6.3 Sparsity Experiments

As part of the evaluation, the effect of dataset sparsity has been analyzed. As mentioned in Chapter 1 the accuracy of a recommender system is assumed to decrease as the number of ratings in the dataset decreases. Further, our intuition tells us that a model is more prone to overfitting as the number of

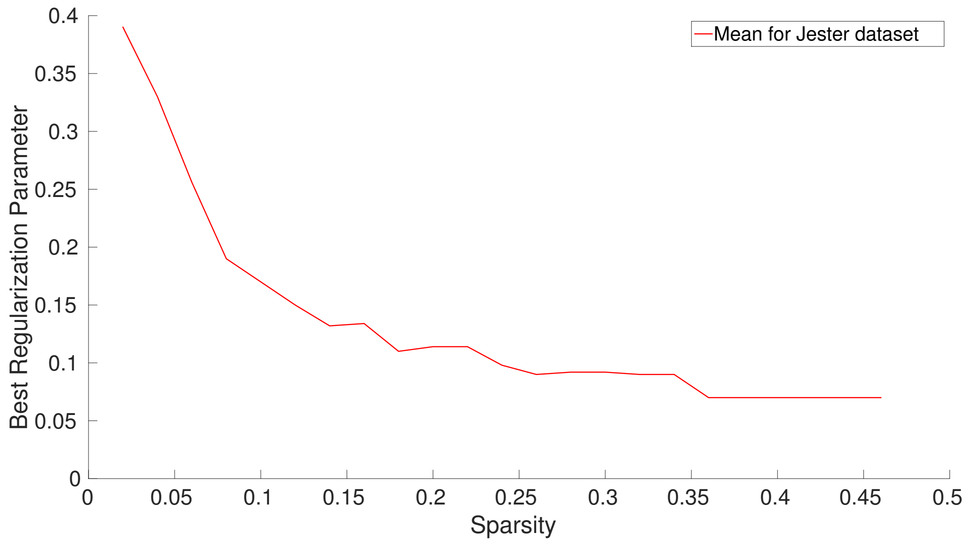

Figure 6.5: Optimal regularization parameter λ, for the ALS-CUDA algorithm for changes in sparsity.

training samples decreases, because each of the samples is presented more often to the model and the variation in ratings is decreased. This experiment focuses on validating these two intuitions.

As described in Section 6.1 the Jester dataset has a sparsity of 56.34%. This means that even after dividing it into a training and test set the dataset will still have a high sparsity percentage. This allows for a sparsity analysis using a real dataset. Figure 6.5 illustrates the effect of changes in sparsity on the regularization parameter. For sparser datasets a higher regularization parameter needs to be chosen, which is as expected, as the reduced number of ratings can quickly lead to overfitting. The largest standard deviation for this experiment was 9.8×10^{-3}.

Taking it a step further and looking at the general accuracy of the model for different levels of sparsity shows that accuracy decreases as the number of ratings in the dataset decrease. The results can be seen in Figure 6.6. They agree with our intuition that more ratings in the dataset will improve the accuracy. From these experiments we can conclude that real-world recommender systems need to be designed to account for changes in sparsity. The largest standard deviation for this experiment was 1.4×10^{-3}.

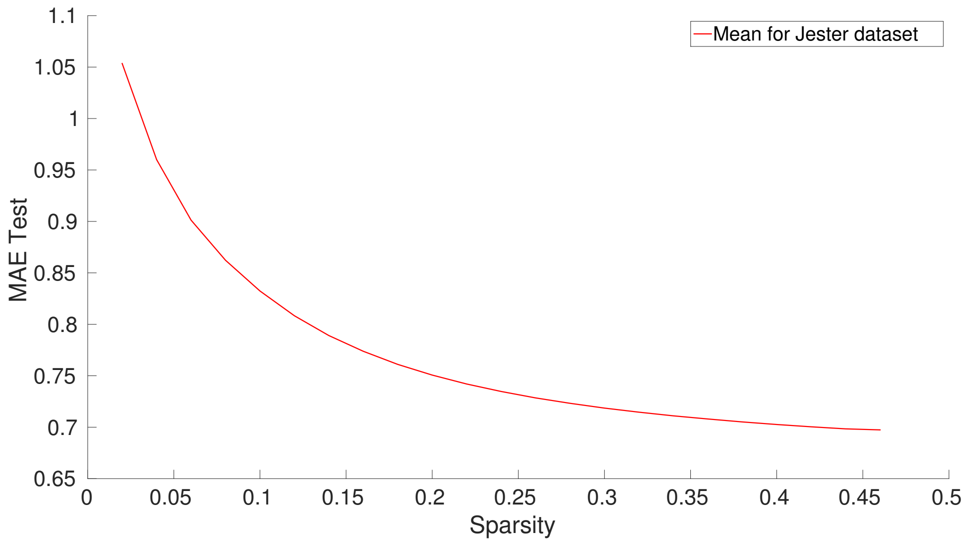

Figure 6.6: MAE on the Jester test dataset for varying sparsity for the ALS-CUDA algorithm.

6.4 ALS-Scalability

Next we evaluate the scalability of the ALS-CUDA algorithm proposed in this thesis. Figure 6.7 shows the scalability of the ALS-CUDA algorithm. The speedup is calculated by measuring the time it takes for the algorithm to reach the threshold of 0.5732 for the Douban dataset for various numbers of CUDA threads, and is normalized by the time it takes one half-warp to do the same. The number of threads is increased in steps of 128 (4 warps, or 1 block in our case). It can be seen that the ALS-CUDA algorithm scales linearly up to 4096 threads, before a significant drop in computational performance can be observed. An analysis of the drop indicates that it is due to the limited amount of shared memory available to each of the 8 SMXs in the Kepler architecture.

Our algorithm requires each half-warp to store its own symmetric matrix for its dynamic batch matrix multiplication. This means that for a 16 dimension matrix factorization, 136 32-bit elements need to be stored per half-warp. Using this information to calculate the shared memory requirements for one block (4 warps) means that the symmetric matrix requires approximately 4.35kB. As the algorithm also requires another 136 32-bit element matrix per half-warp to store the matrix decomposition, as well as additional shared memory for solving the linear system of equation, the total shared memory requirement per block is 10.594kB.

The Kepler architecture supports a maximum amount of 48kB of shared mem-

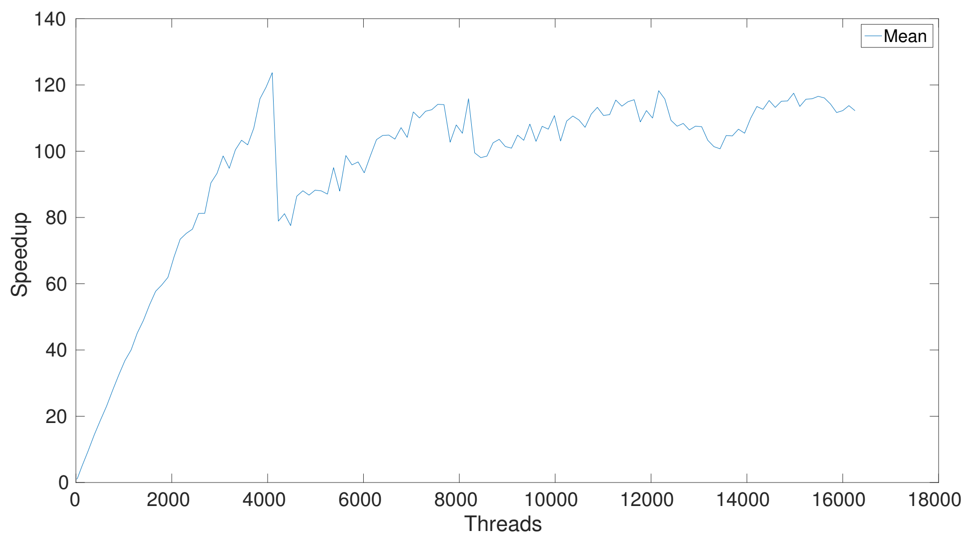

Figure 6.7: Illustrates how the ALS-CUDA-implementation scales with an increase in assigned CUDA-threads.

ory per SMX [40]. This means that the number of blocks that can be resident on each SMX for our algorithm, given a block size of 4 warps, is

$$\left\lfloor \frac{48kB}{10.594kB} \right\rfloor = 4\ . \tag{6.2}$$

This corresponds to 512 resident threads per SMX or, since the device consists of 8 SMXs, 4096 resident threads on the device. This is considerably lower than the 2048 maximum threads per SMX that is stated in the Kepler architecture [40].

The effect of moving the symmetric matrices to global memory (2 symmetric matrices per half-warp), and thereby decreasing the shared memory requirement of each block to approximately $2.09kB$, was tested. Figure 6.8 illustrates the number of blocks that can reside on a SMX for a given amount of shared memory. As can be seen from the figure, the modified algorithm should not be constrained by shared memory requirements, and each SMX should therefore be able to have 16 resident thread blocks with 64 active warps. Figure 6.9 shows the results of scaling the modified ALS-CUDA algorithm, and it can be seen that it scales linearly for a much large number of threads, until 12,288 threads (or 12 blocks per SMX). To support the full amount of 16 blocks per SMX, the number of registers (local memory) per thread would have to be reduced.

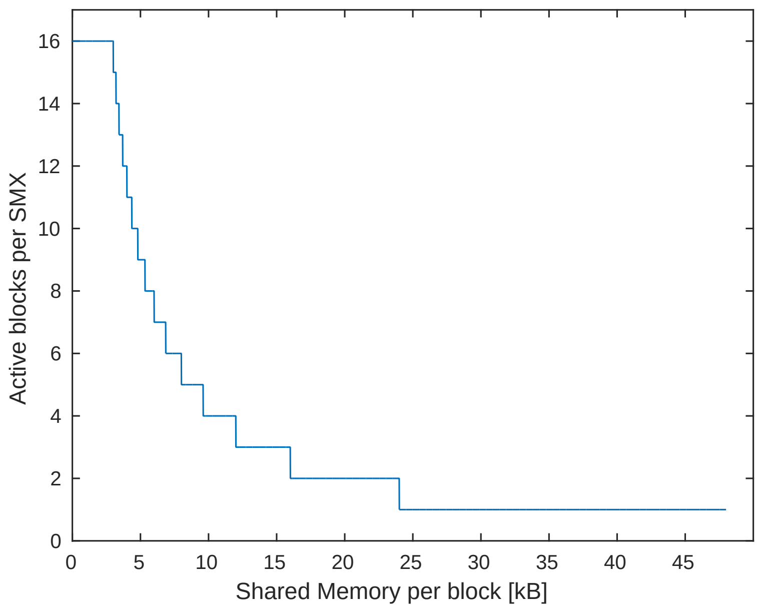

Figure 6.8: Active blocks that can be resident on an SMX for a given amount of shared memory.

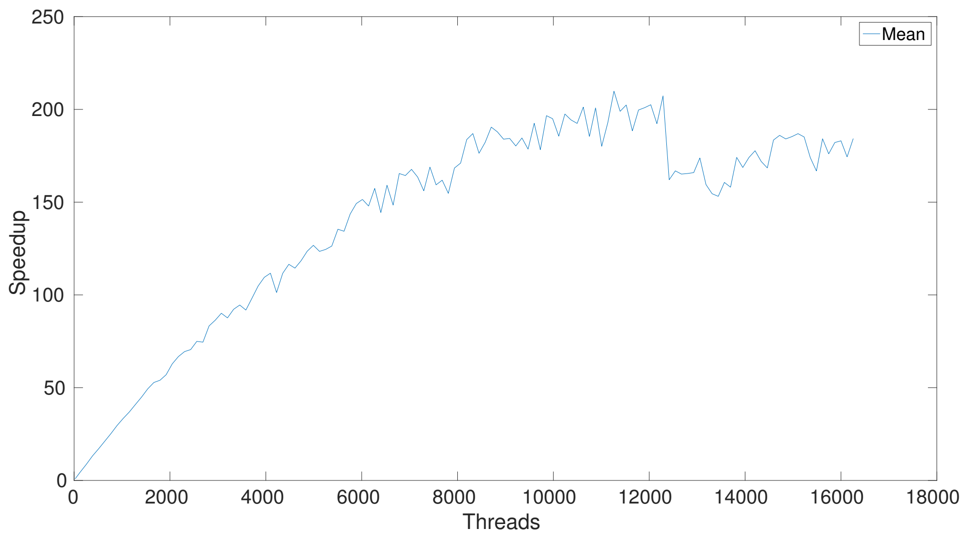

Figure 6.9: Illustrates how the ALS-CUDA global memory implementation scales with an increase in assigned CUDA-threads.

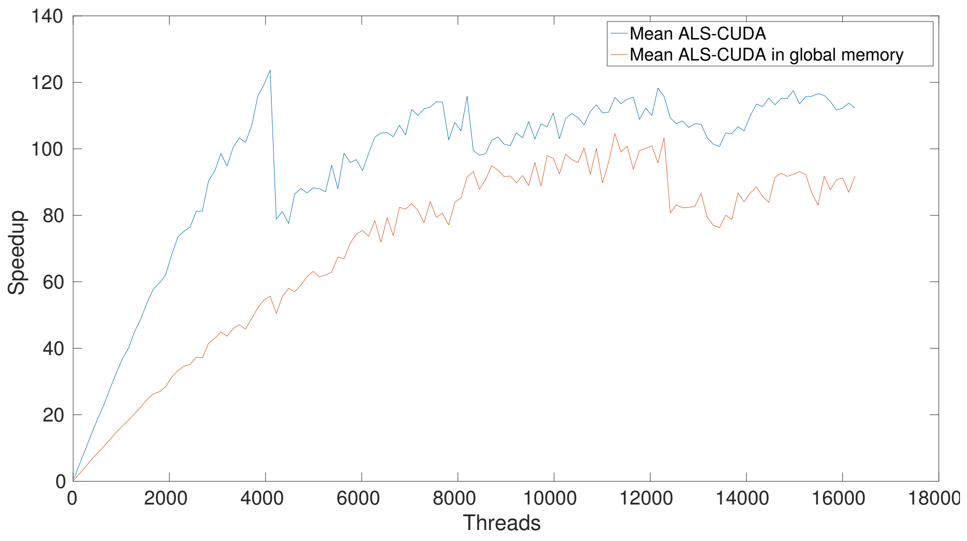

Figure 6.10: Compares the scalability of both the shared memory and the global memory ALS-CUDA algorithm with an increase in CUDA-threads.

The figure shows that increasing the number of threads for the modified ALS-CUDA algorithm, also increases the amount of global memory accesses, which leads to constraints due to memory bandwidth. Figure 6.10 compares the shared memory ALS-CUDA to the global memory version. The speedup for both algorithms normalized by the one half-warp performance of the shared memory implementation is shown. We see that the shared memory version performs generally better due to the utilization of fast shared memory. However, we also see that the performance difference between the two implementations decreases as the number of assigned threads increases.

6.5 Speedup of Matrix Factorization Algorithms

The computational performance of the two matrix factorization algorithms, Hogwild and ALS-CUDA, has been compared. This was done by using the best parameters for each of the algorithms and measuring the time for each of them to reach a certain threshold. For the Douban dataset this threshold was chosen to be 0.5732,[3] whereas the threshold for the Jester dataset was set to 0.71.[4] Figure 6.11 shows the performance of the two algorithms for both datasets for

3. The value was chosen since it was achieved using matrix factorization recommender systems for the Douban dataset in Ma et al.2011 [35].
4. For the Jester dataset the value was hand-chosen, due to a lack of better options.

the sequential CPU and the parallel GPU implementations. To be able to make a fair comparison, the parallel SGD algorithm was measured once for the ideal block/thread configuration of the ALS-CUDA algorithm and once for its own ideal configuration. A scalability experiment, similar to the one in Section 6.4, showed that the best performance of the SGD algorithm is reached when all 8 SMXs have 2048 resident threads. From the figure we can see that significant performance gains are achieved for both datasets when using the GPU. Table 6.3 displays the speedups for the ALS-CUDA, the SGD with ALS-CUDA configuration, and the ideal SGD block/thread configuration for both datasets. For the SGD implementations the configuration is specified as (block/thread).

Table 6.3: Speedup for the ALS-CUDA and SGD algorithm for the Douban and Jester datasets.

	Speedup Factor		
	ALS-CUDA	SGD (32/128)	SGD (128/128)
Douban	175.4	133.9	209.3
Jester	131.3	73.4	225.3

Figure 6.12 provides a close-up of the results for the parallel versions of the SGD and the ALS-CUDA algorithm. We observe that the ALS-CUDA algorithm clearly outperforms the SGD algorithm on both datasets, when the same configuration is chosen. Additionally, it outperforms the ideal SGD configuration on the Jester dataset, but is slightly slower than the SGD on the Douban dataset. The difference in computational performance for the ALS-CUDA algorithm between the two datasets can be explained by the fact that one epoch has to compute updates for each row and each column. A dataset that has a low sparsity will therefore require more row and column update computations compared to a dataset with high sparsity. Including the fact that the ALS algorithm only requires one parameter (the regularization parameter), it provides a good alternative to the SGD in situations where a trade-off between performance and the benefit of having fewer parameters can be made, or in situations where the dataset has a high density.

6.6 Comparing Matrix Factorization Approaches to RBM-CF

We compare the computational performance of the CUDA RBM-CF algorithm to the two matrix factorization recommender systems, SGD and ALS-CUDA.

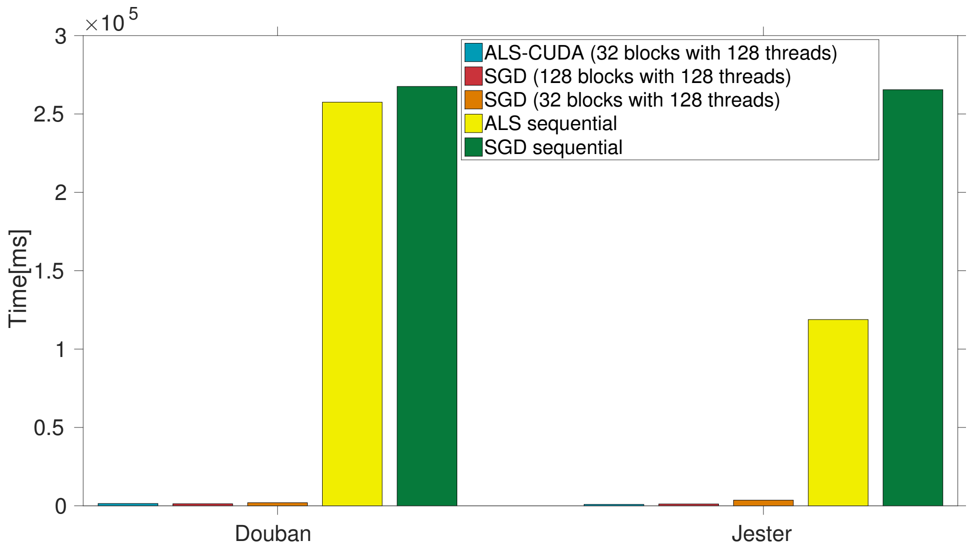

Figure 6.11: Comparison of the sequential and parallel implementations of the two matrix factorization methods (SGD and ALS-CUDA) for the two dataset.

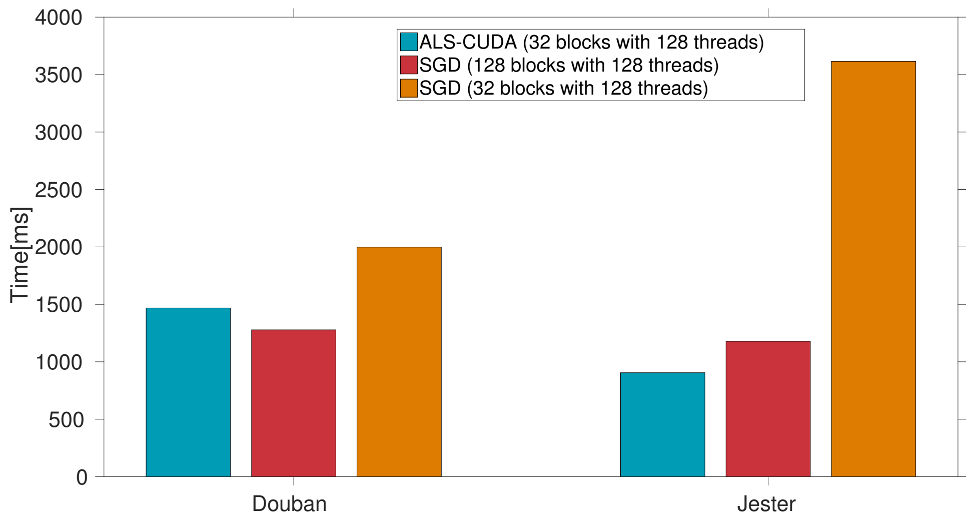

Figure 6.12: Comparison of the parallel implementations of the two matrix factorization methods (SGD and ALS-CUDA) for the two dataset.

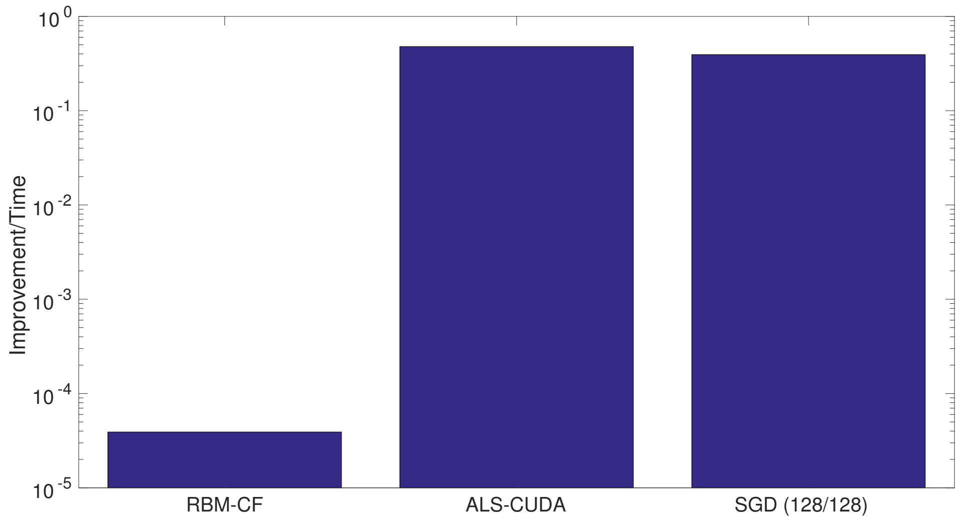

Figure 6.13: Comparing the three algorithms by using a measure of improvement divided by time.

As the RBM-CF algorithm did not reach the threshold for the Jester dataset,[5] the plot compares them using a measure of reduction in MAE over time. This means that the improvement in MAE compared to the initial MAE is measured once each algorithm converges and is divided by the time it took to reach convergence. Figure 6.13 shows that the RBM-CF algorithm is considerably slower than both the SGD and ALS-CUDA algorithms, which agrees with our findings presented in Section 4.2.2.

5. The reason for the RBM-CF algorithm not reaching the threshold could be due to the fact that the whole training was performed using 1 Gibbs sampling step per update, or due to the fact that the number of hidden nodes was not ideal.

7 Concluding Remarks

Recommender systems have become important parts of online services, such as Amazon and Netflix. As these services are scaled to accommodate large numbers of users, items, and ratings, it is essential to understand how the underlying algorithms can be efficiently parallelized.

In this thesis, we have described and analyzed key issues involved when parallelizing recommender systems on CUDA GPUs. We have developed ALS-CUDA, a novel algorithm to solve the matrix completion problem on CUDA and compared its performance to two other recommender system algorithms. The ALS-CUDA algorithm clearly outperforms the RBM implementation, and is only slightly slower than the SGD algorithm on the Douban dataset. It does outperform the SGD algorithm on the Jester dataset, whilst additionally requiring fewer training parameters. The scalability of the ALS-CUDA algorithm was analyzed, and we observe that it scales linearly with the number of CUDA threads until the shared-memory limit of the SMXs is reached.

7.1 Future work

On the performance side, the extensibility of the recommender system problem to GPU-clusters could be investigated. This would help solve the problem of having access to only a limited amount of memory. Especially for larger datasets, implementations of CUDA-aware MPI, such as MVAPICH, could be used [58].

However, distributed shared-nothing algorithms would have to be employed due to the memory not being shared by different nodes. Many common shared-nothing approaches to matrix factorization recommender systems are based on the SSGD algorithm, where the computation of the local losses could be performed using GPUs. This was, however, out of scope for this thesis.

The ALS-CUDA algorithm could be evaluated on more modern GPUs of the Maxwell architecture. With the introduction of compute capability 5.2, the maximum amount of shared memory was doubled (compared to the Kepler architecture) and SMs now offer 96 kB instead of 48 kB [10]. Generally, the trend tends to go towards more shared memory per SM, which the ALS-CUDA algorithm will benefit greatly from. For the modern Maxwell architectures, the ALS-CUDA implementation is expected to support 8 blocks à 4 warps per SM. The Maxwell architecture also allows the implementation of a hybrid between the shared memory and the global memory version, where the dynamic batch matrix multiplication is stored in global memory, but solving the linear system of equations is completely done using shared memory. This hybrid version should be able to support the maximum number of 16 blocks à 4 warps resident per SM, as the Maxwell architecture increases the number of registers per thread by a factor of 4 compared to the Kepler GPU used as part of this thesis.

The ALS-CUDA algorithm has been evaluated in this thesis in the context of recommender systems. However, matrix factorization is applicable to many fields, such as computer vision or document clustering [21, 59]. Analyzing the algorithm in a non-recommender system context could provide further insight into the advantages and disadvantages of the ALS-CUDA in a more general setting.

On the accuracy side, an investigation of the algorithms extensibility could be performed. Especially the matrix factorization methods can be extended to include additional features, which would improve their accuracy. However, adding more features might also introduce additional challenges with regards to the CUDA programming model, that need to be considered. On the other hand, hybrid systems could be tested for real-world applications, where matrix factorization recommender systems are used and combined with RBMs. The effect on accuracy of using stale RBMs combined with matrix factorization systems could be analyzed and might allow for more accurate high-performance systems, in which the RBM model is updated less frequently due to their lack in performance.

A more thorough analysis of when the SGD performs better than the ALS-CUDA algorithm and vice versa is also part of future work and was out of scope for this thesis.

7.2 Conclusion

As part of this thesis we have successfully designed and implemented a novel ALS-WR algorithm on CUDA—ALS-CUDA—and have achieved significant speedup factors of up to 175.4 over a sequential CPU implementation of the ALS-WR algorithm. To design the algorithm, inspiration was taken from both the SGD and RBM algorithms. A comparison of the three algorithms indicates that the ALS-CUDA algorithm outperforms the SGD algorithm for small, dense datasets, whereas the SGD algorithm performs better on sparser datasets. Additionally, our results show that both the SGD and ALS-CUDA algorithms outperform the RBM algorithm.

List of References

[1] Howard Anton and Chris Rorres. *Elementary linear algebra: with supplemental applications*. Wiley, 2011.

[2] William Aspray. The Intel 4004 microprocessor: What constituted invention? *Annals of the History of Computing, IEEE*, 19(3):4–15, 1997.

[3] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[4] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[5] Robin Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.

[6] Xianggao Cai, Zhanpeng Xu, Guoming Lai, Chengwei Wu, and Xiaola Lin. GPU-accelerated restricted boltzmann machine for collaborative filtering. In *Algorithms and Architectures for Parallel Processing*, pages 303–316. Springer, 2012.

[7] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[8] NVIDIA Corporation. *CUDA C Programming Guide*, 7.0 edition, March 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[9] NVIDIA Corporation. Tuning CUDA Applications for Kepler. http://docs.nvidia.com/cuda/kepler-tuning-guide/, 2015.

[10] NVIDIA Corporation. Tuning CUDA Applications for Maxwell. http://docs.nvidia.com/cuda/maxwell-tuningguide/, 2015.

[11] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[12] Christian Desrosiers and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender systems handbook*, pages 107–144. Springer, 2011.

[13] Michael D Ekstrand, John T Riedl, and Joseph A Konstan. Collaborative filtering recommender systems. *Foundations and Trends in Human-Computer Interaction*, 4(2):81–173, 2011.

[14] Blake Foster, Sridhar Mahadevan, and Rui Wang. A GPU-based approximate SVD algorithm. In *Parallel Processing and Applied Mathematics*, pages 569–578. Springer, 2012.

[15] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[16] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. GPGPU Processing in CUDA Architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[17] Peter N Glaskowsky. NVIDIA's Fermi: the first complete GPU computing architecture. *White paper*, 2009.

[18] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[19] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[20] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[21] David Guillamet and Jordi Vitria. Classifying faces with nonnegative matrix factorization. In *Proc. 5th Catalan conference for artificial intelligence*, pages 24–31, 2002.

[22] Trevor Hastie, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman,

and R Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[23] Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004), 2004.

[24] Geoffrey Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[25] Geoffrey Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1):926, 2010.

[26] Michael Kampffmeyer. A Lock-Free Parallel Matrix Factorization Recommender System Using CUDA. Capstone Project, December 2014.

[27] Efthalia Karydi and Konstantinos G Margaritis. Parallel and Distributed Collaborative Filtering: A Survey. *arXiv preprint arXiv:1409.2762*, 2014.

[28] Kimikazu Kato and Tikara Hosino. Singular value decomposition for collaborative filtering on a GPU. In *IOP Conference Series: Materials Science and Engineering*, volume 10, page 012017. IOP Publishing, 2010.

[29] Kimikazu Kato and Tikara Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 769–773. IEEE Computer Society, 2010.

[30] Arnd Kohrs and Bernard Merialdo. Clustering for collaborative filtering applications. In *In Computational Intelligence for Modelling, Control & Automation. IOS*. Citeseer, 1999.

[31] Joseph A Konstan and John Riedl. Recommended for you. *Spectrum, IEEE*, 49(10):54–61, 2012.

[32] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81, 2009.

[33] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[34] Duc Le and Emily Mower Provost. Emotion recognition from spontaneous speech using hidden markov models with deep belief networks. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 216–221. IEEE, 2013.

[35] Hao Ma, Dengyong Zhou, Chao Liu, Michael R Lyu, and Irwin King. Recommender systems with social regularization. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 287–296. ACM, 2011.

[36] Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis. Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*, pages 1–31, 2014.

[37] Faraz Makari Manshadi. Scalable optimization algorithms for recommender systems. 2014.

[38] Sparsh Mittal and Jeffrey S Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys (CSUR)*, 47(2):19, 2014.

[39] Abdel-rahman Mohamed and Geoffrey Hinton. Phone recognition using restricted boltzmann machines. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4354–4357. IEEE, 2010.

[40] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110., 2012.

[41] Anil Poriya, Tanvi Bhagat, Neev Patel, and Rekha Sharma. Non-Personalized Recommender Systems and User-based Collaborative Recommender Systems. *International Journal of Applied Information Systems*, 6, 2014.

[42] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[43] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.

[44] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[45] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of

netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[46] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.

[47] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[48] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 158–167. ACM, 2000.

[49] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, DTIC Document, 2000.

[50] Daniel V Schroeder. *An introduction to thermal physics*, volume 60. Addison Wesley New York, 2000.

[51] Upendra Shardanand and Pattie Maes. Social information filtering: algorithms for automating "word of mouth". In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 210–217. ACM Press/Addison-Wesley Publishing Co., 1995.

[52] Yue Shi, Martha Larson, and Alan Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)*, 47(1):3, 2014.

[53] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.

[54] Graham W Taylor, Geoffrey E Hinton, and Sam T Roweis. Modeling human motion using binary latent variables. In *Advances in neural information processing systems*, pages 1345–1352, 2006.

[55] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 655–664. IEEE, 2012.

[56] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, 4th edition, 2008.

[57] Aalap Tripathy, Suneil Mohan, and Rabi Mahapatra. Optimizing a collaborative filtering recommender for many-core processors. In *Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on*, pages 261–268. IEEE, 2012.

[58] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development*, 26(3-4):257–266, 2011.

[59] Wei Xu, Xin Liu, and Yihong Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 267–273. ACM, 2003.

[60] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems.

[61] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, 41(3):793–819, 2014.

[62] Gao Zhanchun and Liang Yuying. Improving the collaborative filtering recommender system by using GPU. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, pages 330–333. IEEE, 2012.

[63] Yi Zhang and Jonathan Koren. Efficient bayesian hierarchical user modeling for recommendation system. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 47–54. ACM, 2007.

[64] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

[65] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen 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.

A

Theoretical floating point operations calculation

This appendix explains the origin of the results presented in Figure 1.2. The theoretical number of floating point operations per second (TFLOP/s) is calculated using

$$\text{TFLOP/s} = C \times f \times \text{FLOP/c} \,, \tag{A.1}$$

where f denotes the frequency of each of the C cores and FLOP/c is the number of single or double precision floating operations that can be performed per cycle by a given architecture. The GPU and CPU information was taken from the specifications provided by NVIDIA[1] and Intel.[2] In cases where both a base and boost clock frequency where available the base clock frequency was chosen.

This can be illustrated using an example. The GeForce 780 Ti has 2880 cores with a base clock frequency of 875 MHz, where each core can perform two

1. nvidia.com
2. ark.intel.com

single floating point operations per cycle leading to

$$\begin{aligned} \text{TFLOP/s} &= 2880 \times 875 \times 10^6 \text{cycles}/s \times 2\text{FLOP/cycles} \\ &= 5040\text{GFLOP}/s. \end{aligned} \tag{A.2}$$