CUDA-accelerated Boolean Matrix Factorization

Adrian Lamoth

05.02.2018

Johannes Gutenberg-Universität Mainz

Institut für Informatik
Research group High Performance Computing

Bachelor thesis

CUDA-accelerated Boolean Matrix Factorization

Adrian Lamoth

1. Reviewer Prof. Dr. Bertil Schmidt
High Performance Computing
Johannes Gutenberg-Universität Mainz

2. Reviewer Univ.-Prof. Dr. Stefan Kramer
Data Mining
Johannes Gutenberg-Universität Mainz

Supervisor Dr. Christian Hundt

05.02.2018

Adrian Lamoth
CUDA-accelerated Boolean Matrix Factorization
Bachelor thesis, 05.02.2018
Reviewers: Prof. Dr. Bertil Schmidt and Univ.-Prof. Dr. Stefan Kramer
Supervisor: Dr. Christian Hundt

Johannes Gutenberg-Universität Mainz
Research group High Performance Computing
Institut für Informatik
Saarstraße 21
55122 Mainz

Abstract

Boolean Matrix Factorization (BMF) is commonly used in the field of data analysis. The algorithm decomposes a matrix $C(m \times n)$ into two smaller matrices $A(m \times k)$ and $B(k \times n)$ with k being a user set dimension. Therefore, the resulting factorization can either be exact $(AB = C)$ or an approximation $(AB \approx C)$. To solve this problem in a high parallel fashion, **CuBin** (**Cu**da-accelerated **Bin**ary Matrix Factorization) is introduced. In contrast to other state-of-the-art algorithms, the purposed one consists of alternatively changing rows and columns of A and B randomly with the help of thousands of small threads best suited for the CUDA programming model. The parallelization over thousands of entries enable full usage of all available cores on a modern NVIDIA GPU. Additionally, modelling up to 32 entries of given binary matrices A, B and C as a single integer data type results in faster and fewer accesses to the data. With this discrete approach, the memory requirements while executing are lowered substantially. $CuBin$ is evaluated against other state-of-the-art matrix multiplication algorithms. Experiments on multiple real-world data sets show competitive results with only a portion of computation time used. $CuBin$ proves to be a good compromise between low run times and a decent factorization result for the decomposition of large Boolean matrices.

Acknowledgement

I want to thank all the persons that supported me and helped me on my way to coding and writing this bachelor thesis. Thanks to Prof. Dr. Bertil Schmidt for giving me the opportunity to write my thesis in his department about my favourite topic in university, parallelism on CPU and GPU and David Krüger for the usage of his GTX 1080 I used for my evaluation. A very special gratitude goes out to Katharina Dost who gave me an unpublished paper I learned a lot from. Special thanks to Dr. Christian Hundt for continuously giving new fresh ideas and topics during frequent and (almost always) very long talks. It was always fantastic to discuss issues not only related to this work but about everything that is happening in regard to HPC, NVIDIA and programming. Thanks for the passion these talks gave me to never stop asking questions.

Contents

1 Introduction

1.1 Motivation and Problem Statement

Matrix factorization of large low-rank matrices is a commonly used algorithm, primary in the fields of data analysis. The goal is to decompose a very large and sparse matrix C into two small and more dense matrices A and B using a dimension parameter k so A is an $m \times k$ and B is an $k \times n$ matrix. In contrast to regular matrix decomposition algorithms, namely *LU decomposition* or *eigendecomposition*, the resulting factor matrices own different dimensions, thus efficiently saving space. In addition these factors are used to find patterns for improvements of a current state. As a result, this factorization can either be exact $(AB = C)$ or an approximation $(AB \approx C)$. While the approximation should be as good as possible, it is also desired to keep information about inherent structures of the given matrix.

This work will focus on the decomposition of Boolean matrices where entries are either 1 or 0 (true/false). Large binary matrices are often found in real-world applications considering the relations of every element of two sets can be expressed with them. Take the following example: A streaming platform is used by thousands of users, each one with access to over a hundred thousand movies. By creating a matrix with these two dimensions, a relation between every user and all movies can be created, for example showing if a specific movie has already been watched or rated. To implement some kind of *recommender system*, a set of users and movies is taken to create groups of users with similar movie preferences and movies being watched by similar people. The reconstruction of a large matrix by using two smaller ones creates overlapping groups, each representing similar movies with similar users. Users could get recommendations for movies which they have not seen yet but are likely to find appealing.

This technique can also be applied to improve shops by helping them grouping their products better, physically or also as recommendations in an online shop. Only storing the two factor matrices additionally leads to better memory utilization if the dimensions of A and B are small and the original dimensions are enormous. To sum it up, the intention is to minimize the reconstruction error by finding the best possible matrices A and B approximating C or also called "ground truth".

While there is a lot of theoretical work on binary matrices [21, 22, 18], especially by Pauli Mietten [28, 33, 32, 31] and Seung *et al.* [25, 26], there are not many direct implementations for very large matrices, especially ones written for the execution on a GPU. Thus, the **Cu**da-accelerated **Bin**ary Matrix Factorization algorithm **CuBin** is introduced. The purposed algorithm is easy to understand, competitive and scalable. With the help of thousands of cores on the graphics card, leading to massive parallelism perfectly suited for large matrices, a solution is found providing comparable reconstruction errors and low execution time.

In the first chapter, the basics and terminology are clarified. Next, $CuBin$ is theoretically explained and reasoned. After discussing the NVIDIA GPU programming model in-depth to fully understand the hardware and software mechanics, the implementation is introduced. Different ways of achieving the best possible solution in a highly parallel fashion are discussed. After explaining each step, a corresponding CPU implementation is shown. In the end, $CuBin$ is evaluated and compared to itself and other state-of-the-art algorithms and potential future work is discussed.

1.2 Matrix multiplication

To effectively implement the matrix-matrix and matrix-vector multiplication, we first take a look at the regular multiplication algorithm. For two given matrices $A(m \times k)$ and $B(k \times n)$ the result C will be a $m \times n$ matrix. Every entry $c_{i,j}$ with $0 \leq i < m$ and $0 \leq j < n$ is calculated by:

$$c_{i,j} = \sum_{l=0}^{k-1} a_{i,l} \cdot b_{l,j} \tag{1.1}$$

This exhibits the fact that the calculation of one entry is dependant on exactly one row of A and one column of B. Corresponding entries of this row and column get multiplied and summed up yielding the result $c_{i,j}$.

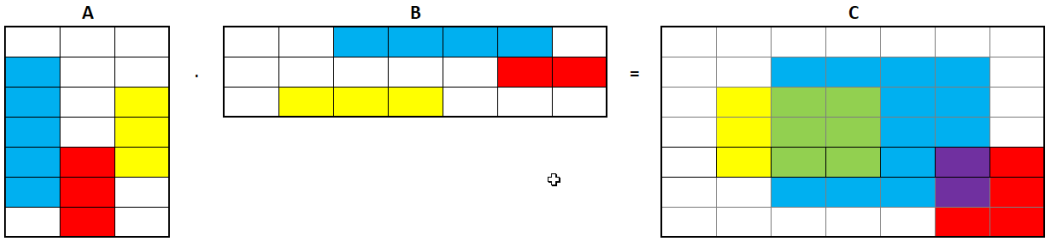

Fig. 1.1: Matrix-matrix multiplication with colors, overlapping colors added up.

For Boolean matrices $A, B \in \{0,1\}^{m \times n}$, wrong results could emerge with entries greater than 1, illustrated by the green and purple color in figure 1.1. Simply adding a $modulo$ 2, so $(1 \oplus 1) \equiv 0 \ (modulo\ 2)$ would also be incorrect considering $1 \oplus 1$

(or $true \oplus true$) should equal 1, not 0. With the use of **and/or** conjunctions the matrix-matrix calculation can be rewritten:

$$(A \circ B)_{i,j} = c_{i,j} = \bigvee_{l=0}^{k-1} a_{i,k} \wedge b_{k,j} \tag{1.2}$$

The difference between the regular matrix multiplication algorithm and the modified one can easily be shown by an exemplary calculation of one entry by taking one row from matrix A and one column from B:

$$\begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1 = 2 \, (regular) \tag{1.3}$$

$$\begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = (1 \wedge 1) \vee (1 \wedge 0) \vee (0 \wedge 1) \vee (1 \wedge 1) = 1 \, (boolean) \tag{1.4}$$

As desired, the outcome with equation 1.2 will always be 0 or 1.

1.3 Error measurement

When multiplied, the factors A and B return a matrix C'. The most common way to determine the reconstruction error, also called distance, is by either measuring it with the help of the Hamming distance or the Euclidean distance. On matrices, they are defined as follows:

$$d_{hamming}(C, C') := |\{i \in \{1 \dots n\}, j \in \{1 \dots m\} \mid c_{i,j} \neq {c'}_{i,j}\}| \tag{1.5}$$

$$d^2_{euclidean}(C, C') := \sum_{i,j} c^2_{i,j} - {c'}^2_{i,j} \tag{1.6}$$

Formula 1.5 represents the number of indices where corresponding entries $c_{i,j}$ and $c'_{i,j}$ are different. The squared Euclidean distance sums up all squared difference between each entry of both matrices. When working with discrete Boolean matrices, there is no difference between the two methods. Each entry $c_{i,j}$ which is not equal to

$c'_{i,j}$ would increase the error by exactly 1, leading to following formula being used:

$$d_{bool}(C, C') := \sum_{i,j} |c_{i,j} - c'_{i,j}| \tag{1.7}$$

To summarize, the goal of finding the best suitable matrices for A and B is reached by minimizing the error, calculated with 1.7, between the reconstructed matrix $C' = AB$ and the given matrix C.

1.4 Matrix rank

Generally said, the rank of a matrix describes the maximum number of linear independent vectors. This definition also holds true in Boolean matrix algebra, even though the rank itself may differ: For some matrices the Boolean rank can be up to twice the rank and for some the rank goes down to the logarithm of the normal rank [28].

By calculating the rank *r* of $C \in \{0,1\}^{m \times n}$, a perfect factorization $(A \circ B) = C$ can be found if $A \in \{0,1\}^{m \times r}$ and $B \in \{0,1\}^{r \times n}$. For most data mining applications, a dimension factor k, usually much smaller than $rank(C)$, is given in advance considering calculating the rank in advance is very computation-intensive for large matrices. Additionally, the resulting rank could be way too high, so a potential factorization would not be worth the time. This is why in practice, the rank is neither known nor used most of the time.

A good example showing the difference between a Boolean and a normal factorization is shown by Miettinen [28]:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \tag{1.8}$$

$$= \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \tag{1.9}$$

$$\approx \begin{pmatrix} 1/2 & 1/\sqrt{2} \\ 1/\sqrt{2} & 0 \\ 1/2 & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}+1}{2} & \frac{\sqrt{2}+2}{2} & \frac{\sqrt{2}+1}{2} \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \end{pmatrix} \tag{1.10}$$

For a given matrix, the Boolean rank-2 factorization shown in equation 1.9 is exact with using the equation 1.2 for each entry. With the regular, non-binary computation seen in equation 1.10, no exact rank-2 decomposition can be found. The Boolean rank in this example would be two, while the normal rank is one greater.

1.5 Non-negative Matrix Factorization (NMF)

The problem of finding the best possible solution for a given matrix C and a factor $k \in \mathbb{N}_{>0}$ is proven to be NP-hard while additionally generalizing k-means clustering [45, 48, 1]. As a result, there is no algorithm that always finds the best possible solution. A minimization of $|C - AB|$ is not expected to be a global minimum, only a local one due to a bumpy optimization landscape [26, 29].

Bipartite Graphs

As explained in the motivation statement, $m \times n$ matrices can be seen as relations between two sets of size m and n. This fact can be illustrated using an unweighted, undirected and bipartite graph $G = (V \cup U, E)$ with $m + n$ vertices. With V owning m vertices and U owning n, an entry $c_{i,j}$ equals 1 iff $(v_i, u_j) \in E$, observable in figure 1.2.

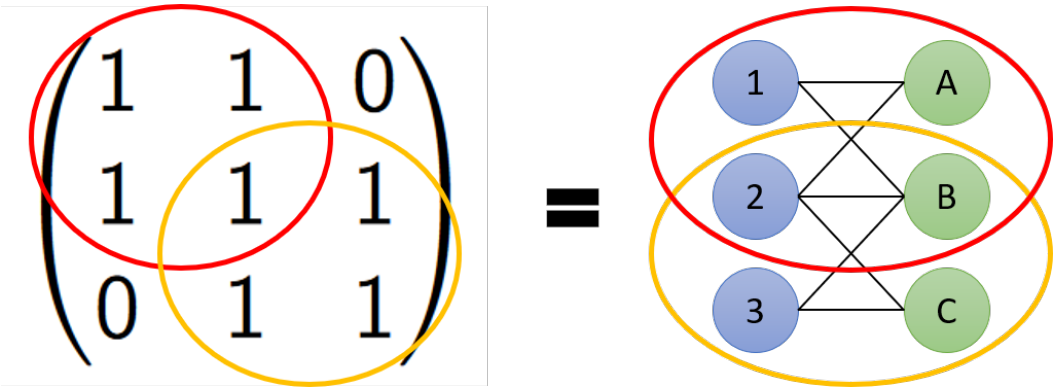

Fig. 1.2: Binary matrix and its corresponding graph.

By definition, a *biclique* is a graph where each vertex on the left side is connected to each one on the right side. These subgraphs can been seen in the red and yellow circles in figure 1.2. When performing an exact factorization with the factor k, k complete bicliques can be found in the graph, covering all edges with at least one.

The first biclique, colored in red, contains the values $(1, 1, 0)^T$ and $(1, 1, 0)^T$ (all vertices in the red circle equal 1, all the others 0) which is exactly the first column and first row (transposed) of the exact factorization in equation 1.9. The yellow biclique

is displayed by $(0,1,1)^T$ and $(0,1,1)^T$ which are equal to the second row/column. The two factor matrices thus create two overlapping blocks when reconstructing, leading to an exact factorization.

As a result, an approximate Boolean matrix factorization with a given factor k produces k quasi-bicliques which cover most but not all edges. More information about the relation between matrices and sets, graphs and more can be read in [29].

Related work

The most common method to approach a factorization is by initializing A and B to step-wise update them until a desirable state is found or no improvements are to be made. Often cited and used is the algorithm of Lee and Seung [25, 26]. The multiplicative update rule, described as a "good compromise between speed and ease" is defined with the following rule:
For given ground truth C, initialize A and B positive:

$$A_{i,k} \leftarrow A_{i,k} \frac{(CB^T)_{i,k}}{(ABB^T)_{i,k}} \tag{1.11}$$

$$B_{k,j} \leftarrow B_{k,j} \frac{(A^TC)_{k,j}}{(A^TAB)_{k,j}} \tag{1.12}$$

with the multiplications done on an element by element basis. This updating proceeds until both A and B are stable.

Another considerable way to disentangle the problem is called *gradient descent*, which takes the euclidean loss function 1.6 described earlier and differentiates along both A and B, resulting in the gradient, to then "walk along" the descent using a small step size until a local minimum is found.
To receive the gradient, first the error function is taken:

$$d_{ij}^2 = (c_{ij} - c'_{ij})^2 = (c_{ij} - \sum_{l=0}^{k-1} a_{ik}b_{kj})^2 \tag{1.13}$$

After differentiating with respect to A and B, an update function can be written and a step size α is set. Its value determines the speed a minimum is approached with. When set too high, the minimum might never be reached since it might be computing around it. When set too low, the update progress might be very slow, especially with large matrices. A possible implementation on python and more in-depth information can be found at [27]. The use of a greedy rank-one downdating algorithm has also

proven to provide good results by Bergmann *et al.* at the analysis of genome-wide expression data [2].

All these algorithms were primary built for regular, non-binary NMF. Updating corresponding rules for the Boolean case can be very complex and not comprehensible but they roughly follow the same course of action: initialize two matrices and update them until an abort condition is met. Zhang *et al.* published a work with two algorithms suited for binary matrices, the first one based on a penalty function similar to the multiplicative update rule and the other one using the gradient approach [50].

While both rely on alternating updates, Miettinen also purposed another procedure [29]: Find submatrices which are as dense as possible (quasi-biclique) and for a given k, choose the best ones to create the two factor matrices, illustrated in figure 1.2.

1.6 CuBin

While all noted and explained algorithms for NMF contain more or less complex calculations, the approach taken in $CuBin$ is unlike all others. While also locally updating both matrices alternately to minimize the loss function until a minimum is found, there are no real update instructions. With the high level of parallelism the GPU brings to the table, which will be explained in-depth later, it is possible to simply try out updates by randomly flipping entries on the matrices and check if the changes made reduce the error. While this greedy algorithm does not seem very feasible in regard to performance at first, the amount of saved calculations and accesses to the matrices cannot be neglected.

Due to the dependencies of calculating a specific matrix entry and to optimize the calculation, only chunks of full rows and columns will be altered randomly. A rough pseudo code is shown in algorithm 1.

In most literature, the factors matrices are coded continuously and values are initialized and bound in range [0,1]. This is needed considering algorithms doing alternating, calculated updates use a small step size, leading to minimal updates which would be impractical in a discrete fashion. Since $CuBin$ is completely random anyway, it follows a discrete approach. As a more important note, it opens the door for additional speedups of almost 4x due to better memory management as well as less memory accesses, illustrated in detail later.

```
Data: Matrix C to be factorized, factor k;
initialize matrices A and B;
calculate starting error |C-AB|;
while error > threshold and iterations < max_iterations do
    /* alter A                                                    */
    change n random rows of A randomly;
    calculate error_new;
    if error_new < error_old then
        apply changes to A;
        update error;
    else
        discard changes;
    end
    /* do the same for B                                          */
    change n random columns of B randomly;
    calculate error_new;
    if error_new < error_old then
        apply changes to B;
        update error;
    else
        discard changes;
    end
end
```

Algorithm 1: Pseudo code of $CuBin$.

1.6.1 Initialization

Even though algorithms exist for the initialization of A and B [24, 3], most algorithms still use a random one [44] due to the additional time consumed. Using a naive random initialization of A and B, densities of around 50% would emerge, even in a discrete implementation (rounded values). Due to the great sparsity of C, a high starting error would be the consequence.

To counteract this, Dost *et. al.* [20] used the density of the ground truth d_C while considering the factor k to determine the starting density of A and B using the following equation:

$$a_{ij} = \begin{cases} 1 & \text{if } randomNumber \in [0,1] < \sqrt{1 - \sqrt[k]{1 - d_C}} \\ 0 & \text{if } randomNumber \in [0,1] \geq \sqrt{1 - \sqrt[k]{1 - d_C}} \end{cases} \tag{1.14}$$

This is done analogously for matrix B. Another simple method to initialize the matrices consists of initializing them with the same density as the ground truth or alternatively even lower with a density of $d_C/100$. While this does not change the outcome of the optimization by a large margin, the differences on the starting error are still worth noting and measured in the performance chapter.

1.6.2 Error measurement

The biggest advantage of the algorithm lies in its simplicity and space efficiency. Calculating the initial error $|C - AB|$ can be done row-wise or column-wise eliminating the need to store two $(m \times n)$ matrices. In $CuBin$, the rows of matrix A are being iterated and with the use of the full matrix B, one line in AB is constructed. The row-wise error with the corresponding line in C is added to the global error, also eliminating the need to fully store matrix C'.

Even though it still comes down to a full matrix multiplication with a run time complexity of $\mathcal{O}(mnk)$, the space requirements are down to two variables created for a temporary entry and the global error. In addition, this algorithm only needs to be called once in the beginning after the initialization of A and B.

As a result, by changing entries during the updating process, only individual rows and columns have to be reconstructed because both the error measurement and the updating do not need the complete reconstructed matrix.

1.6.3 Optimization

For the updating of matrix A, multiple random rows are altered the same way: A row $a_{i,*}^{old}$ is replicated and then changed by flipping random bits in it, producing $a_{i,*}^{new}$. Both rows are multiplied with B using matrix-vector multiplication. The resulting vectors $(AB)_{i,*}$ and $(A'B)_{i,*}$ are then taken to create two temporary errors $error\_old_i$ and $error\_new_i$, measuring the distance between each one and the given line $c_{i,*}$. The difference is used for updating the global error if $a_{i,*}^{new}$ is better suited than $a_{i,*}^{old}$, being the case when $error\_new_i - error\_old_i < 0$. The counterpart, matrix B, is changed similarly: Columns, noted as $b_{*,j}$, are picked and randomly transformed. Using the multiplication with A, the same error measurement is done using column $c_{*,j}$. Both update steps proceed until one abort condition is met.

It could potentially make a big difference between strictly alternating updates of single rows and columns of A and B in contrast to first changing multiple lines or even the whole data once before continuing with the other matrix. The impact on the quality and reconstruction error is also measured in the performance chapter.

The correctness of $CuBin$ is an important property to show. Most of the described methods in other works are also based on a greedy algorithm. In both cases, changes to the matrices are done which will reduce the error and therefore walk along the descent of the loss function, the difference being the way and the frequency other algorithms do their updates. While $CuBin$ does not update every step, it accepts the

same changes leading to a walk along the descent to a local or global minimum. It does not matter how many tries are needed, $CuBin$ will eventually find updates until a minimum is reached or an abort condition is met, similar to the other methods.

Metropolis algorithm

To overcome a local minimum in favour of reaching a global one, a lot of improvements are possible. The usage of a *Markov Chain Monte Carlo* method could potentially be worth the additional run time to get a better end result and the **Metropolis** algorithm is really easy to implement in $CuBin$. Instead of discarding all errors greater than 0, an additional probability is calculated: $p_A = min(1, e^{-\frac{\delta E}{kT}})$, where k is the Boltzmann constant (set to 1 in our case) and T represents the temperature of the system. A uniform number in $[0, 1]$ is taken and checked if it is smaller than p_A. The altered line, even if it increases the error and would therefore be discarded, is then applied if this condition is true. The temperature starts high at the beginning, leading to more frequent updates and gets colder the longer the system runs, thus increasing the run time and error curve.

Finding good values for T and how the temperature is decreased over time has proven to be a challenging task. For too high temperature values the error might not improve at all. When set too low, the usage of the Metropolis algorithm becomes irrelevant. The same principle can be applied to the question when and by how exactly the temperature is cooled down. Additionally, it might not even produce better results, but the algorithm itself is worth noting since it could potentially improve outcome by a large margin.

To sum it up, a lot of factors can be taken into consideration which could make a more or less big impact on quality of the result and the run time of all algorithms for this problem, the most prominent being:

- Number of rows and columns changed at once
- How exactly the updating is done
- Coding the matrix as continuous or discrete
- How to initialize the factors A and B
- The given factor k

2 CUDA

2.1 Introduction

As pointed out, $CuBin$ involves a lot of trying out random changes to the matrices which would result in a very poor performance if realized in a non-parallel fashion while a high-parallel approach on the other hand can potentially be competitive to current implementations.

Generally speaking, there are two components in a regular PC suited for parallel computing: the CPU and GPU. Using modern programming language syntax, threads can be created, each one doing separate operations in parallel. As a side note, not every problem can be written in parallel, let alone achieve a better performance this way. There is not always a way to effectively parallelize a given algorithm by breaking it down into parallel chunks due to obstacles like data dependency or order of execution.

With the help of current CPUs, threads can be created which are assigned to one core. Currently, CPUs for personal use mostly come with four, six or eight cores while more expensive ones like AMD Epyc or Intel Xeon Phi can reach up to 72 cores. Meanwhile, the most recent graphics cards from NVIDIA, the GTX 1080 Ti, TITAN XP or TITAN V, contain up to 5,120 cores.

Both GPU and CPU create parallel working threads but the structure is dissimilar: CPU threads are relatively heavy-weight in comparison to their GPU counterpart. They are smarter, more flexible and mostly do completely different things.

This comes down to the "Flynn's taxonomy", a classification of architectures. CPUs normally use the **MIMD** technique to acquire parallelism, meaning "multiple instruction, multiple data" while the GPU architecture gets described as **SIMD**, "single instruction, multiple data" [13]. NVIDIA itself describes its devices as **SIMT**, "single instruction, multiple threads" [9]. As a result, a CPU is best suited for different instructions on multiple data or data streams, while a GPU uses the same instructions for every thread just on different parts of data. In reality, SIMD extensions are also already built into current CPUs which can be used to greatly increase performance depending on the task.

The SIMD approach on GPUs is based on its fundamental and primary task: rendering pictures and coloring pixels on the screen. A screen with the size 1280×1280 can easily be broken down into 6,400 independent 16×16 pixel chunks for example. The calculation of shadow and light can be done on a pixel-by-pixel basis so a parallelization is easily achievable: 6,400 threads each do the same calculation on their chunks and project it onto the screen while all use the same complete image/scene as input data.

The current CUDA SDK version number is 9.1 for compute ability 3.0-7.0 and works on Kepler, Maxwell, Pascal and Volta architecture. For this thesis, CUDA 8.0 will be used. For a good overview of features released and changes implemented by newer compute abilities, refer to the NVIDIA programming guide or CUDA Wikipedia page [9, 4]

2.2 Architecture

Without going into too much detail, I will explain the architecture of the card used for the implementation performance benchmark, the NVIDIA GeForce GTX 1080 with the Pascal architecture. The GTX 1080 holds 20 streaming multiprocessors, also called SM, SMM or SMX. Each SM contains different kinds of memory, cache, schedulers and 128 streaming processors, also called cuda cores [39]. Figure 2.1 shows the composition of the NVIDIA GeForce GTX 1080 GPU.

When executing a function on the GPU, a grid is created and assigned to a graphics card. A grid contains thread blocks or just blocks, each assigned to a single SM. Introduced with the Fermi architecture, multiple blocks of different GPU functions can reside on a single SM [42]. Threads of a block are then executed by cuda cores. The current limits are 1,024 threads per block with a maximum of $2^{31} - 1$ blocks per grid in x dimension and 65,535 in y and z dimension.

The understanding of a **warp** is essential for a good optimization. In all CUDA versions to this date, 32 threads of each SM start at the same program address at the same time. Thus, a block is effectively processed by a sequence of $blocksize/32$ warps. Performance suffers if thread divergences occur and threads of a warp are at different instructions. Using *if-then-else* structures and executing different instructions for threads in a warp will result in performance loss. In this situation threads stall and need to wait on each other until every thread completed all branches of the control flow.

Fig. 2.1: GTX 1080 GPU architecture.

With this in mind, always setting 1,024 threads per block as default and spawning thousands of blocks is not the best idea, especially with the limited size of shared memory and register file size, shared among all blocks and threads residing on a streaming multiprocessor. On the other hand, too little wraps/threads per block are also not sufficient enough for effective *latency hiding*: When instruction arguments are not ready for use, the GPU switches between warps to hide latency. NVIDIA suggests about 512 threads per SM as a rough estimate to hide latency and achieve a sufficient amount of parallelism [43].

Memory

Global memory, the memory with the largest storage, can be accessed and modified by every SM in the GPU. Physically, it resides off-chip, leading to high latency and relatively low throughput. **Shared memory**, sitting on-chip on every SM with a size of 96 KB, is shared by all cores on the unit. Therefore, 1920 KB can be utilized in total. While being much smaller than the 8 GB of available global memory, the usage of shared memory may result in a speedup up to 10x in comparison to off-chip storage [14]. At first, **texture memory** was also used to implement $CuBin$ but was later replaced. The on-chip, read-only storage is commonly used for texture handling

with different access patterns. The storage structure is optimized for 2D locality [9], so accesses on texture (or matrix) entries - which are close in two dimensional space - yield the best performance. Memory types that are not used and thus not explained are local and constant memory.

Global memory is allocated by CPU code. A regular C-pointer is initiated, usually with the prefix $d\_$ to distinguish between host and device arrays. Using the command *cudaMalloc*, which takes a pointer together with the size, global memory on the GPU is allocated. Afterwards, reading/writing from/to this pointer with regular C code is not possible anymore. Instead, data is copied from device to host or the other way around using *cudaMemcpy* specifying source, destination, size and direction. Reversely, the GPU cannot access data allocated on the CPU.

Introduced in CUDA 6.0 for devices with compute ability 3.0 or higher and 64-bit host applications, **Unified Memory** deserves a special mention. While not present in my work, it eliminates the need for specific declared copies between GPU and CPU by defining a memory space readable by both. Data transfers still take place in the background but accesses are much more transparent while coherence between host and device arrays is given. However, a work by Landaverde *et al.* [23] observes a significant level of performance overhead introduced when accessing unified memory. For additional information about memory types, their declaration, scope and lifetime, also refer to [8].

2.3 CUDA Code

CUDA is a programming language extension by the NVIDIA Corporation released in 2007 for C and C++. Many other popular languages like Fortran, Python and MATHLAB are also able to execute CUDA commands by now [10]. The structure of a program using CUDA does not differ much from regular C code. Initialization and reading of data is done with CPU code while only the compute-intensive functions are outsourced to the GPU. NVIDIA itself talks about **5%** of code which is done by the GPU [49]. These intense computations are done by special function calls, so-called **kernels**. In contrast to regular functions in C code, a kernel has two additional parameters indicating how often it is called in parallel:

```
kernelName <<< numberBlocks, threadsPerBlock >>> (parameter1, parameter2);
```

Now the kernel does not run a single time but *threadsPerBlock* many threads in *numberBlocks* blocks are running the same function in parallel. Pointer parameters given to kernels must be point to allocated memory on the GPU initialized by *cudaMalloc*.

To calculate a global thread id in 1D, following equation is executed:

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

where the three factors are fixed expressions known to every thread in the grid. *threadIdx.x* specifies the local thread ID in its respective block in range $[0,$ threadsPerBlock $-1]$, *blockIdx.x* the position in the grid in range $[0,$ numberBlocks $-1]$ and *blockDim.x* is equal to *threadsPerBlock*, additionally illustrated by figure 2.2.

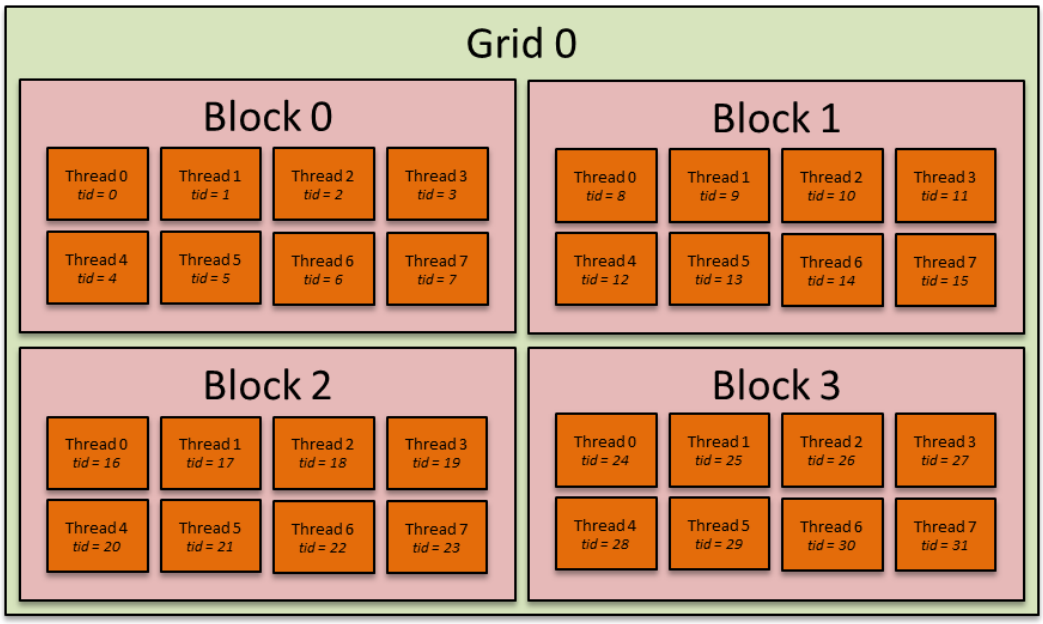

Fig. 2.2: A grid with four blocks and eight threads per block, all 1D.

Following the SIMT/SIMD structure, each thread works on its specified part of the same data using the global thread ID for accesses. The most common scenario is giving the kernel the pointers and calculating ID-dependant offsets to work on the right data. A basic kernel to take the elements in an array and multiplying by two in parallel could look like this:

```
__global__ void simpleMultiply(int* elements, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        elements[tid] *= 2;
    }
}
```

The integer tid indicates the position in the array *elements* each thread spawned is assigned to multiply one entry from. The check *if (tid < size)* prevents errors when more threads than number of entries in *elements* are created, otherwise resulting in memory access errors. On the other hand, declaring too little threads would yield an incorrect result with not processed data. Thus, it is necessary to create enough

blocks with a sufficient number of threads. Using regular C code, the initialization could look like this:

```
int* elements;
int* d_elements;
int size = 8192;

elements = (int*) malloc(sizeof(int) * size);         // Malloc on CPU memory
cudaMalloc((void **) d_elements, sizeof(int) * size) // Malloc on GPU memory
for (size_t i = 0; i < size; ++i) {                   // Fill array with values
    elements[i] = i;
}

// Copy array to device
cudaMemcpy(d_elements, elements, sizeof(int) * size, cudaMemcpyHostToDevice);

// Kernel call
// Multiply every element by two in parallel
simpleMultiply <<< size/512, 512 >>> (d_elements, size);

// Copy back to CPU
cudaMemcpy(elements, d_elements, sizeof(int) * size, cudaMemcpyDeviceToHost);
```

Kernels differ from regular C functions by the additional function parameters and by the prefix **__global__**. It tells the compiler that this function is only to be called by the CPU (also termed **host**) and executed on the GPU (**device**). With the addition of dynamic parallelism with CUDA devices capable of compute capability 3.2 or higher, these functions can also be called by devices. Other prefixes are **__device__** (called by GPU, done on GPU) and **__host__** (called by CPU, done on CPU).

2.4 Performance maximization

From NVIDIA's own presentation about performance optimization, three requirements are to be kept in mind for maximum performance [43]:

- sufficient parallelism
- coherent (vector) execution within warps
- coalesced memory access

Considering the given ground truth matrix is very large and elements are relatively independent from each other (one entry in A changes only one row in the result matrix, as described in section 1.6), a sufficient amount of parallelism is definitely possible.

The second point is the avoidance of thread divergence (to avoid control flow changes) which messes up the warp structure.

Maximal performance is reached when the L1 and L2 cache is used optimally. Similar to CPU cache, the available size is very limited and a lot of cache misses will subsequently lead to slow global memory accesses. The size of available L1 and L2 cache heavily depends on the used GPU and compute capability used, with an option to completely turn off the L1 cache, but for the GTX 1080 with compute ability 6.1, every SM holds 48 KB L1 cache while all SMs share a 2,048 KB L2 cache [38]. When a warp loads data from an address in the global memory, a line will be pulled and cached. This L1 cache line size is fixed at 128 bytes with the L2 line size being 32 bytes. An optimal use of cache consists of an access of 32 floats aligned successively in global memory, taking only one 128 byte load. The cache line would hold all 32 needed floats leading to minimal cache misses and maximum performance.

3 Implementation

3.1 First steps

While the pseudo code given on page 8 seems simplistic and easy, with the performance optimizations in mind, a lot of factors and complexity come into play when writing the matching CUDA C/C++ code. In this section, I will explain my first steps when given the problem and how parallelism is achieved best.

The biggest question regarding performance and parallelism is the distribution of matrix entries and computations on spawned threads and blocks. The algorithm itself alternates between optimizing one row of A and one column of B as explained earlier, so a lot of of small kernel calls will be executed. To keep the code readable, two kernels exist, one changing a row of the first factor matrix, one transforming a column in B. Both are surrounded by a loop optimizing until a break condition is fulfilled.

As an example, the row-changing kernel (now referred to as kernel A_K) randomly picks a line $a_{i,*}$ which ought to be optimized. Using $a^{old}_{i,*}$ and $a^{new}_{i,*}$ and given formula 1.2, the two lines $c^{old}_{i,*}$ and $c^{new}_{i,*}$ are reconstructed.
Instead of handling all m entries sequentially, m threads are spawned, each one executing only one calculation for entry $c_{i,tid}$ for $tid \in [0, m-1]$, resulting in each of the m threads running:

$$c^{old}_{i,tid} = \bigvee_{l=0}^{k-1} a^{old}_{i,l} \wedge b_{l,tid} \tag{3.1}$$

$$c^{new}_{i,tid} = \bigvee_{l=0}^{k-1} a^{new}_{i,l} \wedge b_{l,tid} \tag{3.2}$$

Figure 3.1 illustrates this approach, with $A(m \times k)$, $B(k \times n)$ and $C(m \times n)$. It does not matter how the m threads are exactly distributed over the blocks, the global thread ID is taken to identify the fitting entry. This is done analogously with kernel B_K: Every thread takes the changed column j in B and its row $a_{tid,*}$ for the reconstruction of $c_{*,j}$.

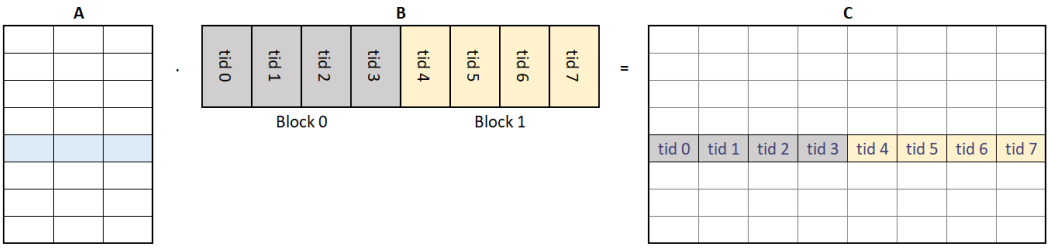

Fig. 3.1: Two blocks with six threads per block. Each thread takes its assigned column *tid* of B while the changed row in A is accessed by all threads.

Afterwards, the error measurement is done with $|c_{i,tid} - c^{old}_{i,tid}|$ and $|c_{i,tid} - c^{new}_{i,tid}|$ to find out if a refinement happened. The two errors are then subtracted from each other to get a delta error per thread which is then added up to get the global $(error\_new - error\_old)$. If it is smaller than 0, the changes will be applied and discarded otherwise. The delta error, $(error\_new - error\_old)$, will from now on only be referred to as *error* considering only the difference is being used for error measurement.

Disadvantages

Using this method, kernel A_K uses *m* threads while kernel B_K spawns *n* threads. If m or n is not large enough, a lot of potential performance is wasted since the graphics card would not use all its cores to serve the threads.

The biggest issue is raised by the lack of synchronization. The randomly changed row $a^{new}_{i,*}$ needs to be known by all threads in the grid **before** starting with the reconstruction and error measurement. As a result, a global synchronization is needed. Until CUDA Version 9.0 there was no command to realize this in a kernel itself, the kernel needed to be shut down first and called again to realize a global synchronization.

To counteract this issue, it would be necessary to apply the random changes beforehand on the CPU to then copy it on the GPU, leading to lacking performance and high latency due to the enormous amount of small copies. The usage of larger, asynchronous copies from host to device could prevent large delays but is still far from performant. Likewise, to simply do the check $if(error < 0)$ to know if an update should be applied, requires all threads to be done with their task of adding their error, otherwise wrong values would be read. A workaround involves letting the kernel finish and check it afterwards either on the CPU or with another kernel, adding unnecessary complexity and overhead to the code.

3.2 Block approach

Even though global synchronization cannot be achieved in a pleasing way, block synchronization can be realized with the help of the command *__syncthreads()*. To effectively use this, the program structure is changed to only assign one block to a single row/column. When the width or height of C is larger than 1,024, a single thread in a block has to manage more than one entry. This can be realized by either letting one thread take $\lceil m/threadsPerBlock \rceil$ successive entries or cycle through it as shown in Figure 3.2.

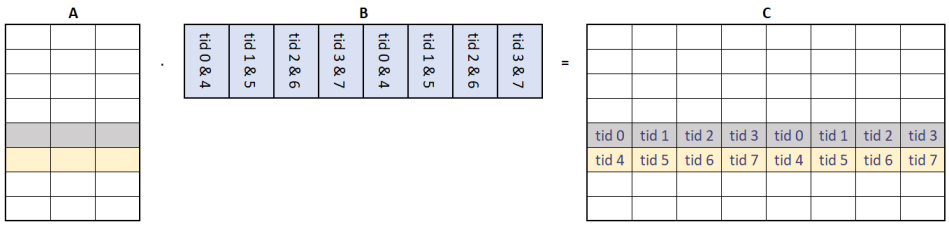

Fig. 3.2: Two blocks with four threads per block. Each block takes one row while the row is distributed over all available threads.

With the help of block synchronization, one thread per block can be delegated to randomly transform the given row or column in the beginning. Pulling a random number $startRow$ or $startCol$, the final line to change is calculated by $(blockIdx.x + startRow \text{ mod } height)$ or $(blockIdx.x + startCol \text{ mod } width)$ to change $numberOfBlocks$ successive rows or columns in wrapping mode.

Afterwards, the reconstruction of the line starts in parallel and after another synchronization the global error is measured using reduction before updating the matrices if necessary. The flow for kernel A_K is illustrated in Figure 3.3.

3.3 PRNG

A *Pseudo Random Number Generator* is an algorithm that generates random numbers on the PC. Considering a machine is not able to ensure *true* randomness, a number of generators exist. For CUDA, cuRAND is part of its toolkit. Unfortunately, it does not support host-sided calls defined by the API. In this work, the *keep it simple stupid (KISS)* PRNG, which is able be called from GPU and CPU, is used. It passes the *dieharder suite*, a test for random number generators [19].

To utilize it, a kiss state has to be formed with a seed first. To receive a new random number, the command *fast_kiss32(&state)* is called to get a "random" $uint32\_t$. Random numbers are utilized in the following parts of $CuBin$:

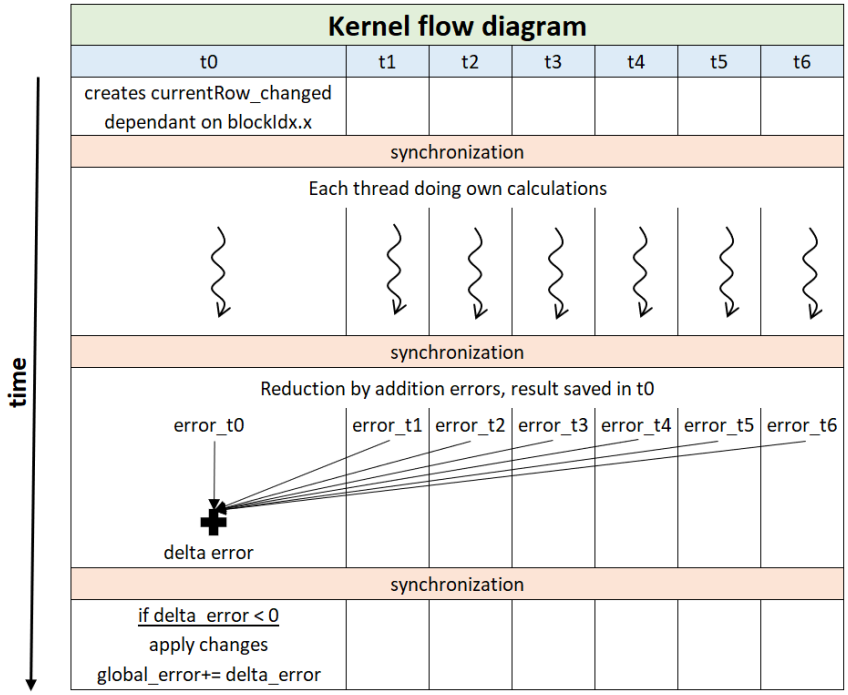

Fig. 3.3: Flow diagram of a kernel A_k with six threads per block.

- Initialization of A and B
- Choosing the row or column to be changed
- Changing the row or column

Concrete implementations for the first two are already explained in chapter 1.6 and 3.2. In contrast to the first point, which only needs to be called once for all entries of both factor matrices and the second one only being called once before a kernel call, random transformations are executed much more often. For $numberBlocks \cdot k$ many entries in each kernel call, there must be a "random" decision to either flip it ($0 \rightarrow 1$ or $1 \rightarrow 0$) or keep its value.

A naive approach to generate a number $\in [0, 1]$ and use some threshold like 0.1 to 0.4 (10-40%) to decide if a flip happens works fine but can slow down the system when executed thousand of times. Better performance is achieved by the usage of bitwise operations. Writing $randomNumber\&\&11$ compares the last two bits of the random number to 11, having a 25% chance to be true while yielding better performance than $if((randomNumber/UINT32\_MAX) < 0.25)$. The flipping is done with the *xor* operator, resulting in the following code bit:

```
uint32_t randomNumber = fast_kiss32(&state);
for (size_t i = 0; i < k; ++i) // k is factor
    currentRow_changed[i] = currentRow[i] ^ ((randomNumber >> i) & 11 ? 0 : 1);
```

In this example, only one number needs to be generated and bit-shifted ensuring the use of all available bits. Furthermore, the factor must be below or equal to 32 to ensure correctness.

3.4 Reduction

The global error change is calculated by accumulating values from every thread in a thread block with the "+" operator. An old presentation of NVIDIA themselves brought up many ways how to efficiently implement and parallelize global reduction and is worth a read regardless of its age [15].

However, the Kepler architecture, released in 2012, introduced a new way of executing reductions even faster with the help of a *shuffle* instruction [12]. The programmer is now able to shuffle values across a single warp. This way up to 32 threads can communicate and copy data (e.g. for reduction) without utilizing shared or global memory. Because every thread holds its own variable *error_thread* in register memory, *__shfl_down* can be used to implement *sequential addressing reduction* as seen in [15] on a warp level, using this function:

```
__inline__ __device__
int warpReduceSum(int val) {
    for (int offset = warpSize / 2; offset > 0; offset /= 2)
        val += __shfl_down(val, offset);
    return val;
}
```

After doing this for every of the maximum 32 warps (1,024 maximum threads per block / 32 (warp size)), all threads with $warpID = 0$ now contain the accumulated values from their warp. Next, these threads write their value into a shared memory array of size 32. After another synchronization, the 32 remaining values are added up by the first 32 threads in the block using the *warpReduceSum* function with the result being contained by thread 0 as the final value:

```
int lane = threadIdx.x % warpSize;  // ID of thread in warp
int wid = threadIdx.x / warpSize;   // ID of warp
distance_thread = warpReduceSum(error_thread);
if (lane == 0) reductionArray[wid] = error_thread;
__syncthreads();
error_thread = (threadIdx.x < blockDim.x / warpSize) ?
                    reductionArray[lane] : 0;
if (wid == 0) error_thread = warpReduceSum(distance_thread);
```

3.5 Memory access and management

Up until now, only the case where matrix A was changed has been explained since it is mostly analogous to updating B by just replacing *row* with *column* and by updating the indexing. In regard to the algorithm itself it did not matter but for memory accesses and performance these kernels turn out very differently. When taking a close look on the execution with the help of NVIDIA's own profiler (nvvp [40]), the kernel changing columns took up to 12x the time of kernel A_k: 2.4 ms vs. $200\mu s$ on average. The matrices A and B, as well as C were stored densely in *row-major order*.

When executing kernel A_k, adjacent threads work on consecutive elements in memory as seen in figure 3.2, leading to an optimum usage of cache and only few cache misses. Even though the width and height of a matrix might not be perfectly divisible by 32, this is an nearly optimal and desired access pattern, also described in [43].

For kernel B_K on the other hand, accesses to the data of A happen to be the worst possible pattern. One thread is responsible for one complete row, stored successively in memory. Therefore, thread warps are accessing elements with a stride of k in between entries. The cache line has a very limited size, resulting in a lot of slow global memory transactions for k many entries in a row.

By storing A and B in a sparse fashion instead of a dense dense this phenomenon can be mitigated but it cannot be eliminated completely considering even sparse matrices have to be stored in either row-major or column-major order. Double storing matrix A as transposed and normal yields way better results and more similar kernel execution times because both kernels then use the same access pattern. As seen by the disparity of kernel execution times, coalesced memory access is a high priority for realizing good CUDA-accelerated parallelism.

Matrix C, residing on global memory also yields different access times for both kernels. While maintaining good access times on kernel A_K, which pulls a row (consecutive entries in memory), kernel B_K suffers from accessing entries with a large offset in global memory.

3.5.1 Bitwise representation of entries

By far the biggest speedup was seen by not using one variable (float, char, etc.) per entry of A, B and the ground truth C, but using only **one** $uint32\_t$ for **multiple** entries. Using a discrete representation, every bit of an (unsigned) integer now

expresses one entry in the matrix, dividing the memory used by up to 32. Matrix $A(m \times k)$ now can be stored as an array with size m instead of size $m \cdot k$ while matrix $B(k \times n)$ is now modeled with an array of size n where a complete column is only one integer. As an example, the matrix row $a_{i,*} = 10010001111000010010$ would be represented as the integer $A[i] = 2^{31} + 2^{28} + 2^{24} + 2^{23} + 2^{22} + 2^{21} + 2^{16} + 2^{13} = 2,447,450,112_{10}$. Since only the highest k many bits are set and used, all other bits are set to zero.

Now, executing a parallel matrix-vector multiplication is simple: For kernel A_K where each thread takes the changed row $a_{i,*}$ and its own column of matrix B $b_{*,tid}$, the result is computed with the basic dot product $c_{i,j} = a_{i,*} \bullet b_{*,tid}$. While working in a Boolean fashion, the following code produces correct results:

```
int cEntry = (A[i] & B[tid]) > 0 ? 1 : 0;
```

Both integers are joined using the bit-wise *AND*. If this integer is bigger than zero, at least one bit is set, so the result entry $c_{i,j}$ should equal one and otherwise zero, analog to calculation 1.4.

In $CuBin$, the factor k must be smaller or equal to 32, otherwise wrong results will occur. Most implementations however only use factors smaller than 32 [51, 46, 20]. If k should be larger than that, loops must be introduced to iterate over all $uint32\_t$'s of a row or column.

Since the dense coding of matrix C also did not yield the best results, the same method used for A and B is applied to store the matrix more efficiently. The implementation was more complex, since a row or a column of the ground truth is always larger than 32. In regard to the indexing of C, a column-major order scheme, illustrated in figure 3.4 and 3.5, was implemented. As a result, the 36×11 matrix of figure 3.4 does not need $36 \cdot 11 = 396$ entries in global memory, it shrunk down to $\lceil 396/32 \rceil = 13$ used unsigned integers, each representing up to 32 entries.

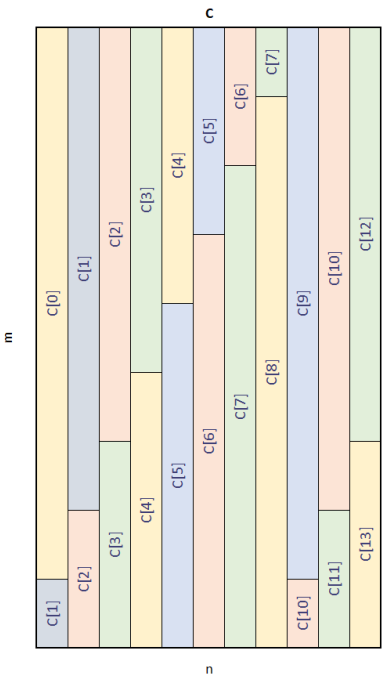

Fig. 3.4: Allocated integers for C.

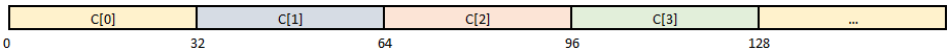

Fig. 3.5: Memory addresses

3.5.2 Bit manipulation

Threads of both kernels now do not need to access A and B k many times to pull a line anymore, once is now sufficient. Matrices A and B are both stored as a linear array of *uint32_t* values, leading to optimal cache usage and coalesced memory access. While this does not completely solve the alignment issue of matrix C, accesses are still much more performant now. Even though there is still a column-major order implementation, the strides between entries in a single row are now up to 32x lower.

To manipulate or access specific entries $a_{i,j}, b_{i,j}$ or $c_{i,j}$, bit-shifts together with bitwise operations are done, leading to a more complex code but a big performance boost.

Generally speaking, using a 1D array for a $(m \times n)$ matrix, an entry p_{ij} is indexed as follows:

```
// column-major order
int cm_entry = P[j * height + i];
// row-major order
int rm_entry = P[i * width + j];
```

This radically changes when setting values in A, B and C coded in the purposed manner. Even though bit-shifts of A and B are not needed when executing the updating process, the initialization has to be bit-wise:

```
// Initialize Ab (A in bit coding) and Bb
bool threshold;
uint32_t randomNumber;
for (int i = 0; i < height; i++) {
    Ab[i] = 0;
    for (int j = 0; j < DIM_PARAM; j++) {
        switch(INITIALIZATIONMODE) {
            case 1: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < (sqrt(1 - pow(1 - density, 1 / (double) DIM_PARAM)));
                                break;
            case 2: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < (density / (double) 100);
                                break;
            case 3: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < density;
                                break;
        }
    }
}
```

```
// Initialize Bb
for (int i = 0; i < width; i++) {
    Bb[i] = 0;
    for (int j = 0; j < DIM_PARAM; j++) {
        switch(INITIALIZATIONMODE) {
            case 1: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < (sqrt(1 - pow(1 - density, 1 / (double) DIM_PARAM)));
                                break;
            case 2: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < (density / (double) 100);
                                break;
            case 3: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                < density;
                                break;
        }
    }
}
```

As seen, both use a nested for-loop along the "longer" side and along the variable DIM_PARAM, representing the factor k of the decomposition (20 as a default value). For every entry, a random number is generated and a threshold is given as explained in chapter 1.6. The bit is set if the random number is smaller than the threshold.

When bit-shifting, we need to pay close attention to the indexing in the matrix and bit position in the fitting integer. While an integer takes 32 bit of memory, assigning the first bit on address 0 is done by setting the most significant bit to 1 using an *or* (| in C++ and C) operation with 1.

For matrix C, this also changes considering there is a variable amount of integers for one row or column, depending on the distribution of integers over the matrix. In $CuBin$, I took the regular column-major scheme and introduce two additional variables *intId* and *intLane*. *intId* takes the regular index and divides it by 32. Values are consecutively split in blocks of 32 bit integers, thus dividing by 32 yields the index of the corresponding integer. By taking the remainder of dividing the index by 32, the position in a given integer is returned. As a result, the following code is used to access the value of entry c_{ij}:

```
int intId = (j * height + i) / 32;
int intLane = (j * height + i) % 32;
int cEntry = (Cb[intId] >> 32 - intLane - 1) & 1;
```

While this may seem complex, a simple example clears things up easily: Take the matrix C with the dimensions (1000×500). $c_{100,30}$ would be stored in $C[30 \cdot 1000 + 100] = C[30100]$ when saved element-by-element and in column-major order. When representing 32 instead of one entry with a single integer, the value would be stored in integer number $940 = 30100/32$ on position $20 \equiv 30100\ (modulo\ 32)$.

3.6 Final implementation

Putting it all together, I will start with the usage of the compiler and command line for additional parameter manipulation. Afterwards, I go through the algorithm putting the previously explained methods into use. The complete source code can be found at `https://github.com/dethknell/CuBin`.

Compiling

As shown in the previous sections, there are a lot of factors which can lead to a different outcome. Some of them can be changed while compiling the program, some are parameters from the command line. Using *nvcc* and CUDA Version 8.0.61, the following line is entered to compile $CuBin$:

```
nvcc source/CuBin_final.cu -std=c++11 -arch=sm_61 -Xcompiler="-
fopenmp"
```

The openMP tag is used to compare the GPU parallelization to a CPU one using openMP pragmas. The Compute ability is set to 6.1 since it resembles the most recent one suited for the GTX 1080. Additional -D parameters are:

- DIM_PARAM (default 20) - changes factor k, must be between 1 and 32
- THREADSPERBLOCK (default 1024) - for smaller matrices below 1,024 in width or length to prevent thread idling
- CPU - turns on CPU optimization after GPU optimization
- CPUITERATIONS (default 100) - set iterations of CPU (very slow)
- STARTINITIALIZATION {1,2,3} (default 2) - sets the method to calculate the density of the initialized factor matrices
  - 1: $d_{A,B}$ calculated with equation discussed in section 1.6.1
  - 2: $d_{A,B} = d_C/100$
  - 3: $d_{A,B} = d_C$
- PERF - turns on performance measurement mode: to plot error curves, intermediate results are written into csv file

Executing

To execute, the following specifications can be passed:

- outputfile - compiled file, error when no input
- updateStep (default 1,000) - after how many steps the current error is shown

- linesAtOnce (default 4,000) - how many rows or columns are changed in parallel
- threshold (default 0) - abort optimization on specific error
- gpuIterations (default 10,000) - total iterations done
- maxIterationsNoImp (default 1,000) - sets abort condition: how many times a line should be accessed without improvement
- startTemperature (default 0) - starting temperature for the Metropolis algorithm
- tempDecreaseStep (default INT_MAX) - after how many iterations the temperature is decreased
- tempDecreaseFactor (default 0.99) - factor the temperature is decreased by

As an example, to run this program on the file *MNIST.data* with progress being shown every 2,000 steps while changing 1,000 rows at once, the following line is entered:

./a.out data/MNIST.data 2000 1000

3.6.1 Read input and initialization

A sparse matrix can be coded in many different ways. The format supported by $CuBin$ consists of a line with the structure *[height],[width]*, followed by one line per non-zero entry, coded in *[col],[row]* coordinate format using the comma as delimiter. Regular COO format contains an additional $value$ field but since it is a binary matrix, this is not needed.

The file is read using the *ifstream* methods from C++. After taking the first line and reading *height* and *width*, the memory is allocated. The setting of the bits in matrix C is done the same way bit shifting as done with matrix A and B.

```
while (getline(infile, linestring)) {
    stringstream sep1(linestring); // line to stringstream
    string fieldtemp;
    getline(sep1, fieldtemp, ',');
    y = stoi(fieldtemp, nullptr);
    getline(sep1, fieldtemp, '\n');
    x = stoi(fieldtemp, nullptr);
    intID = (x * height + y) / 32;
    intLane = (x * height + y) % 32;
    C0[intID] |= 1 << 32 - intLane - 1; // set right bit to 1
    nonzeroelements++;
}
cudaMemcpy(d_C0, C0, sizeof(uint32_t) * sizeC, cudaMemcpyHostToDevice);
```

The ground truth is copied to the GPU with the help of *cudaMemcpy*, so both host and device can access the data. As a reminder: Every array or pointer for the GPU has to be specifically allocated. Even though it is not needed to do the same in C++11, it is done similarly for consistency.

```
sizeC = (int) ceil(width * height / (double) 32.0);
C0 = (uint32_t *) malloc(sizeof(uint32_t) * sizeC); // CPU array
cudaMalloc((void **) d_C0, sizeof(uint32_t) * sizeC); // GPU array
```

The initialization of A and B is done with the help of a specific density and bit manipulation as shown in sections 3.5.2. This is only followed by transferring both matrices to the global memory of the GPU.

3.6.2 Starting error

The one-off calculation of the full global error is done in parallel on the GPU. As previously disclosed, parallelism is created by each thread working on a distinct entry in a row resulting in *n* threads, each reconstructing *m* total entries, reducing the complexity to $\mathcal{O}(m)$.

```
computeFullError <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                        (d_Ab, d_Bb, d_Cb, width, height, (*d_distance_C0_C_start));
```

By calling a maximum of 1,024 threads per block, $width/threadsperblock + 1$ blocks are needed. Due to the cut of decimals when using normal division of integers, one additional block is allocated. Otherwise, using a matrix with 1,025 entries each row would result in one block of 1,024 threads, yielding wrong results. Since the kernel itself is very similar to the main optimizing kernels due to the fact it is a vector-vector multiplication, looped over the height m and paired with an error reduction in the end, refer to the following sections or the complete code in the end for more information.

3.6.3 Optimization

The optimization kernel calls are nested in a while-do loop with three abort conditions: a threshold for the current error, a maximum number of total iterations or a number of iterations where no optimization to each line was done (likely to be a local minimum). The current error is not shown after each optimization step. Instead, the variable *updateStep* can be set for the amount of calculations done without yielding an intermediate result to the terminal.

The kernel loop is then defined as follows:

```
while (threshold < error_C0_C && iterations < gpuiterations
        && iterationsNoImp < maxIterationsNoImp) {

    if (iterations % iterationsTillReduced == 0) // Metropolis
        temperature *= tempFactor;

    // Pull the error from GPU to show it
    if (iterations % updateStep == 0)
        printf("Current error: %f\n", error_C0_C / (double) (width * height));

    /*
    *  Kernels get the matrices A,B and C, the dimensions, the line to be changed for thread 0
    *  the current error variable, a random number as seed for the PRNG
    *  and temperature of the system
    */
    toBeChanged = ((unsigned int) fast_kiss32(&state)) % width;
    matrixCompareCol <<< min(linesAtOnce, width), THREADSPERBLOCK >>>
                            (d_Ab, d_Bb, d_C0, width, height, toBeChanged, d_error_C0_C,
                            ((fast_kiss32(&state) + iterations) % UINT32_MAX), temperature);
    cudaDeviceSynchronize();

    toBeChanged = ((unsigned int) fast_kiss32(&state)) % height;
    matrixCompareRow <<< min(linesAtOnce, height), THREADSPERBLOCK >>>
                            (d_Ab, d_Bb, d_C0, width, height, toBeChanged, d_error_C0_C,
                            ((fast_kiss32(&state) + iterations) % UINT32_MAX), temperature);
    cudaDeviceSynchronize();
    cudaMemcpy(&error_C0_C, d_error_C0_C, sizeof(int), cudaMemcpyDeviceToHost);

    if (error_C0_C_before - error_C0_C == 0) iterationsNoImp += updateStep;
    error_C0_C_before = error_C0_C;
    iterations++;
}
```

As illustrated, each of the *linesAtOnce* many blocks spawned works on its own row or column. The seed for the random number generator for each block, created by thread 0, is a random seed where the iteration and the block ID is added to ensure all blocks do not use the same number chain in every iteration.

Kernel code

Unfortunately, only three variables can be used for storage on shared memory. Saving a complete row or column of C in shared memory will exhaust it for very large dimensions. However, utilizing it for block-wide communication of the altered row or column with size of 32 bit is feasible. Additionally, the PRNG, only used by thread 0 and the reduction array, used for block-wide reduction with the size of 32 integers, are stored on shared memory. All other additional variables can be stored in registers for minimum access latency.

The kernel itself can be split up into four small parts as already illustrated in Figure 3.3. For now, only kernel A_K will be shown. Besides the difference in indexing of the matrices, they are almost similar. Refer to the appendix for the full source code. The thread with *threadIdx.x == 0* initializes and creates a changed row:

```
uint32_t currentRow = A[rowToBeChanged];
if (threadIdx.x == 0) {
    shared_currentRow_changed = currentRow; // load row to be changed in shared memory

    state = get_initial_fast_kiss_state32((seed + blockIdx.x) % UINT32_MAX);
    randomNumber = fast_kiss32(&state);
    #pragma unroll
    for (int i = 0; i < DIM_PARAM; i++) // flip random bits in the row
        shared_currentRow_changed ^= (randomNumber >> i) & 11 ? (0 << i) : (1 << i);
}
```

Afterwards, the parallel work is beginning: Successive threads pull successive columns from matrix B, all looped since the width could easily be over 1,024:

```
for (int i = 0; i <= ((width - 1) / blockDim.x); i++) {
    if ((i * blockDim.x + threadIdx.x) < width) {
        currentColThread = B[i * blockDim.x + threadIdx.x];
        intId = (((i * blockDim.x + threadIdx.x) * height) + rowToBeChanged) / 32;
        intLane = (((i * blockDim.x + threadIdx.x) * height) + rowToBeChanged) % 32;
        cTruthEntry = (C[intId] >> 32 - intLane - 1) & 1;

        cEntryOld = (currentRow         & currentColThread) > 0 ? 1 : 0;
        cEntryNew = (currentRow_changed & currentColThread) > 0 ? 1 : 0;
        error_thread += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
                        ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
    }
}
```

For additional information about the loop used when $width > threadsperblock$, please refer to Figure 3.2. After all $|threadsPerBlock|$ many threads have calculated their own delta error from (potentially) multiple entries of C, the block reduction starts. Refer to Chapter 3.4 for full source code of the reduction using the *shuffle* function.

In the end, the error is always updated if a better solution has been found. If this is not the case, the Metropolis algorithm comes into play. The probability p_A is calculated and the changed row/column is then applied in case the random number is below the probability:

```
if (threadIdx.x == 0) {
    if (error_thread < 0) {
        A[rowToBeChanged] = shared_currentRow_changed;
        atomicAdd(global_error, error_thread);
    } else { // Metropolis algorithm
        randomNumber = fast_kiss32(&state) / (double) UINT32_MAX;
        metro = temperature > 0.0f ? fminf(1, expf(-error_thread / temperature)) : 0 ;
        if (randomNumber < metro) {
            A[rowToBeChanged] = shared_currentRow_changed;
            atomicAdd(global_error, error_thread);
        }
    }
}
```

3.6.4 Post-optimization testing

For debugging and testing purposes, an after-test function was implemented. It takes the final bit-wise matrices and the ground truth and reconverts them to a dense element-by-element representation applying bit-shifting methods:

Matrix A:

```
// A = element by element
// Ab = uint32_t for 32 elements
for(int i = 0; i < height; i++){
    for(int j = 0; j < DIM_PARAM; j++){
        A[i * DIM_PARAM + j] = (Ab[i] >> DIM_PARAM - j - 1) & 1;
        if (A[i*DIM_PARAM + j]) counterDensity++;
    }
}
densityA = counterDensity / (double) (height*DIM_PARAM);
```

Matrix B:

```
counterDensity = 0;
for(int i = 0; i < width; i++) {
    for(int j = 0; j < DIM_PARAM; j++) {
        B[j * width + i] = (Bb[i] >> DIM_PARAM - j - 1) & 1;
        if (B[j * width + i]) counterDensity++;
    }
}
densityB = counterDensity / (double) (DIM_PARAM * width);
```

Matrix C:

```
for (int i = 0; i < width; i++) {
    for (int j = 0; j < height; j++) {
         intId = (i * height + j) / 32;
         intLane = (i * height + j) % 32;
         C0[j * width + i] = (C0b[intId] >> 32 - intLane - 1) & 1;
    }
}
```

Afterwards, two full matrix-matrix multiplications are executed, first with the matrices in bit-wise implementation and afterwards using the dense and therefore larger matrices. The first kernel for parallel matrix multiplication is very simple. In contrast to the optimization kernel, where exactly one block is responsible for a single row of $width$ length, covering a complete row with the whole grid simplifies the kernel to a few simple lines:

```
__global__ void matrixMultiply( uint32_t *A, uint32_t *B, uint32_t *C,
                                int width, int height) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < width)
        for (int i = 0; i < height; i++)
            C[i * width + tid] = (A[i] & B[tid]) > 0 ? 1 : 0; // C in row-major order
}
```

This kernel is simply called by the following line to spawn at least $width$ many threads:

```
matrixMultiply <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                        (d_Ab, d_Bb, d_C_test_GPU, width, height);
```

The code for regular matrix multiplication with the additional use of shared memory can be found all over the internet and is basic CUDA code [6, 7, 17, 9]. I took the implementation from [6] and adjusted it for Boolean matrices by adding $min(1, sum)$ on the final write to the result matrix. Afterwards, both dense matrices are compared to the ground truth to see if all three errors are the same.

For further use of the factor matrices, they are written to a text file in a dense, csv format.

3.6.5 CPU implementation

To see if the GPU implementation is even worth it, I also tried to implement a CPU version of the same code and parallelize it with OpenMP pragmas. Even though the code is much simpler, the performance results are devastating. The CPU implementation took almost 250 times longer to complete a single update step.

While the GPU can parallelize over rows/columns and over multiple lines at once, OpenMP pragmas only parallelize over a single row/column. While it is technically possible to parallelize a double for-loop, the implementation with all helper variables would be very complex.

Additionally, the ultra fast accesses to register memory by the threads and best possible cache usage implemented in the GPU code, outclass the CPU version by

far. As said in the beginning, the algorithm is not really feasible with serial or non-high-parallel implementations due to its high number of small calculations on many elements.

The differences between the kernels and the CPU functions are very minor. There is an additional for-loop which iterates over the lines changed "at once". In reality, they are now processed sequentially on the CPU:

```
for (int l = 0; l < rowsAtOnce; l++) {
    rowToBeChanged = (startrow + l) % height;
    ...
}
```

Instead of spreading the entries of a line or column across GPU blocks, pragmas are used to parallelize the corresponding for-loop, comparable to the second part of the GPU kernel:

```
#pragma omp parallel for private(cEntryOld, cEntryNew, cTruthEntry) reduction(+: error)
for (int tid = 0; tid < width; tid++) {
    uint32_t currentCol = Bb[tid];
    int intId = (tid * height + rowToBeChanged) / 32;
    int intLane = (tid * height + rowToBeChanged) % 32;
    cTruthEntry = (C0[intId] >> 32 - intLane - 1) & 1;

    cEntryOld = (currentRow         & currentCol) > 0 ? 1 : 0;
    cEntryNew = (currentRow_changed & currentCol) > 0 ? 1 : 0;
    error += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
             ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
}
```

4 Performance evaluation

For the performance evaluation a GTX 1080 will be used. It is considerably hard to properly measure the quality of the factorization since the applicability of a given solution can vary widely depending on its intended use. As a consequence, recreating the inherent structure of the ground truth and measuring the results proves to be a challenging task. The optimization landscape shapes up very bumpy and therefore the starting point of the optimization has a high impact on the outcome. Even though the optimization will find a local minimum, it does not express how good this minimum exactly compares in relation to all others.

Different data sets from different sources are taken, illustrated in table 4.1. *MNIST* is constructed with binarized images of handwritten digits [20]. All *Amazon* data sets express user ratings on various product groups with at least five ratings per user. For Boolean matrices, ratings from 1 to 5 for each product are not usable, so the files have been prepared first. Inside the final cleaned file, there exists a line with $\{userId, productId\}$ iff $rating(userId, productId) > 2.5$. The first line of each data set contains the height and width of the matrix. The *C Movie Rec* set is a Boolean matrix prepared for the *C-Salt* algorithm by Hess and Morik [16]. The data is built from the Movie Review Data [11] containing 1,000 positive and 1,000 negative reviews and was transformed from a dense format into a sparse one.

The database of MovieLens [37] offers a lot of data to work with and is widely used in science-related work. Unfortunately, the data contains all kinds of different delimiters for different set sizes. Additionally, while there is a certain amount of

Data Set	Rows	Columns	Entries	Density (%)	Source
Amazon Patio	1,686	962	12,080	0.744	[20]
C Movie Rec	4,442	1,995	326,048	3.679	[16]
Amazon Office	4,905	2,420	50,402	0.425	[20]
Amazon Music	5,541	3,568	58,905	0.298	[20]
MNIST	10,001	785	1,052,359	13.40	[20]
MovieLens 1m	6,040	3,706	836,478	3.737	[36]
Amazon Tools	16,638	10,217	124,371	0.073	[20]
Amazon Toys	19,412	11,924	156,592	0.068	[20]
MovieLens 10m	69,878	10,677	8,242,124	1.104	[35]

Tab. 4.1: Used data sets with their sources and properties.

different movies and users present, not all of the movies are rated at all, which holds also true for the users. To counteract these issues, the MovieLens data set is first brought to a uniform format. The movie IDs and user IDs are cleaned so they can be accessed consecutively as final x and y coordinate in the ground truth matrix. To receive better results, ratings below 2.5 on a $[0, 5]$ point scale are not taken into account. The first line is organized in a *height,width* scheme to match the format of all other data sets for equal initialization.

All tests in this work by **all** algorithms are done with a dimension parameter **k=20** for easier and correct comparison. Files with the ending $.data$ are accepted where the first line contains the dimensions of the matrix and all other lines contain the y, x parameters of a non-zero entry.

4.1 Testing CuBin parameters

4.1.1 Randomness

Due to the high level of dependency on random numbers, their impact will be tested in isolation first. Using the **MovieLens10m** data set, the effect of the following factors on the optimization are tested and plotted:

- Start initialization density
- Seed for random number generator
- Lines per row changed per kernel call

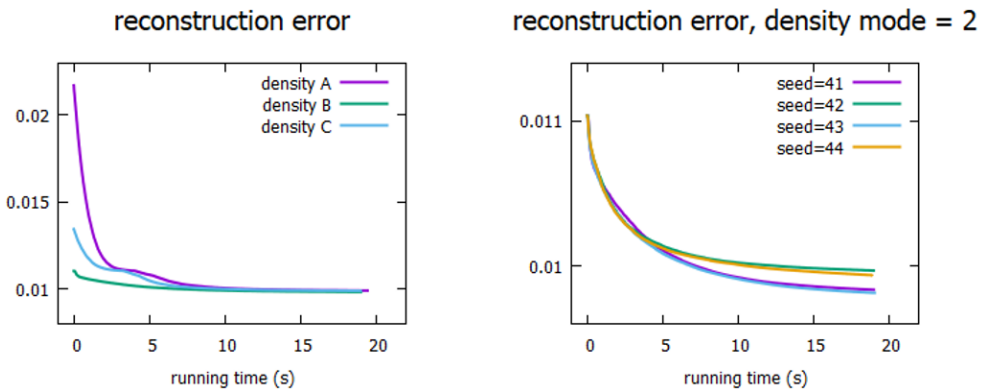

Fig. 4.1: Effects on the reconstruction error for different seeds and different starting densities with 4,000 lines per kernel.

All three starting densities for the factor matrices were introduced in section 1.6.1 and formally written in section 3.6. As seen in figure 4.1, the one purposed by Dost *et al.* brings the highest starting error thus the steepest curve. Using a much more

lower density for the factor matrices, the curve starts relatively low and is therefore not changing as much. For all experiments from now on, the following starting density is taken since it is shown to yield the best starting error:

$$d_{A,B} = d_{C0}/100$$

While this does not lead to the greatest disparities between starting and ending error, this visual drawback is gladly taken considering the end result of the factorization is equal in both cases. The seed for the random number generator is also set to **41** for the isolation tests due to its small impact on the factorization in the long run.

4.1.2 Lines changed per kernel call

Tests in this section are done on **MovieLens 10m**. All tests access **16 million** rows and columns and for optimization. Since this number can be calculated by $iterations \cdot linesChangedPerIteration$, it is possible to change the factors to test the impact of themselves in regard to run time.

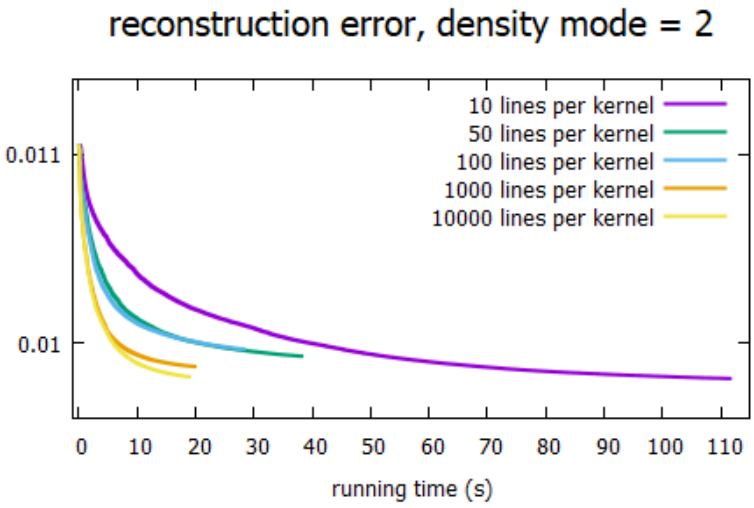

Fig. 4.2: Different splitting of lines changed in parallel and total iterations.

Lines per Iteration	Iterations	Error $\cdot 10^{-3}$	Duration (s)
10	1,600,000	9.81091	115.528
50	320,000	9.99213	38.179
100	160,000	9.96559	28.324
1,000	16,000	9.84721	19.938
10,000	1,600	9.81966	18.937

Tab. 4.2: Final error and run time for different splitting of 16 mil. line changes, corresponding to figure 4.2.

While all tests pulled 16 million lines and applied the optimization algorithm, large differences can be observed. Especially altering less then 50 lines at once, corresponding to very little blocks spawned, results in a heavy performance loss considering the hardware is not even fully utilized. When disregarding said case, there are still seconds of difference for the same effective work. This can be explained by the characteristics of latency hiding and less overhead produced by kernel invocations.

4.1.3 Memory alignment

As previously stated, the implementation of the algorithm started by storing all matrices with a dense, element-by-element scheme. Now, three different methods are tested:

- Storing all three as dense
- Storing C densely in texture memory, A and B with 32 elements for each integer
- Using the bit representation for A and B as well as C

All tests in this section are done with **4,000 lines per iteration** for **4,000 total iterations** on **MovieLens1m**. Every test results in the exact same factorization but the focus lies in the speedup by better aligning the memory. Storing C in texture memory on the second test case allows for faster access on two dimensional objects like the ground truth matrix. As a disadvantage, the maximum dimension of a single texture reference is currently $65,536 \times 65,536$ [5], so the MovieLens10m data set cannot be stored as one dense reference on texture memory.

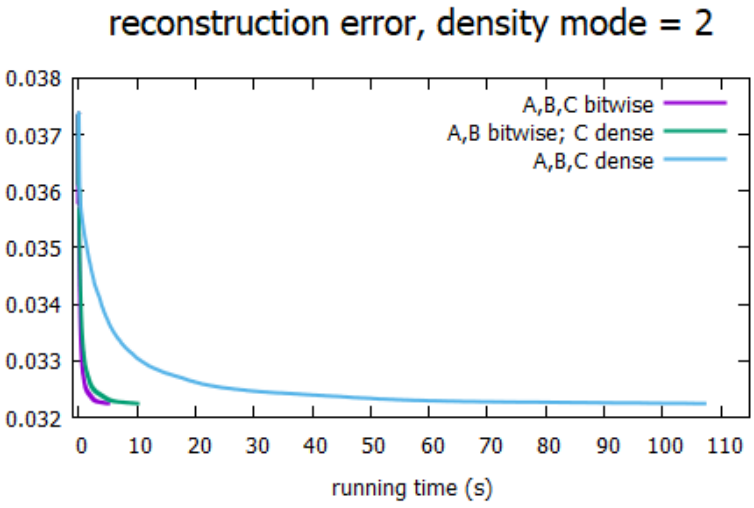

Fig. 4.3: Effects of different memory layouts on the run time.

The exact numbers for the run times are:

- 5.05 seconds for the bit wise implementation
- 10.10 seconds for bit wise implementations of only A and B
- 107.21 seconds for the all-dense scheme

As expected, the time differences illustrated in figure 4.3 are enormous. It was possible to reach speedups of around 20x due to better memory access and cache usage. While storing and accessing with bit manipulation leads to higher complexity in the code, this trade is gladly taken for achieving massive speedups.

4.1.4 Metropolis algorithm

Finding optimal parameters for introducing the *Markov Chain Monte Carlo* has proven to be extremely difficult. The heavy dependency on the three parameters and their fine tuning make it hard to get a desirable result. Unfortunately, in all my testing I have not been able to tweak the system in a way that the factorization yields two better factors in regard to the reconstruction error. While I still believe there might be ways to utilize the Metropolis algorithm to get a more steady curve eventually leading to a better solution, the time needed to find perfect parameters was sadly just not there.

Tests in this section are done on **MovieLens 10m** with **1000** lines changed per kernel for **64,000** iterations.

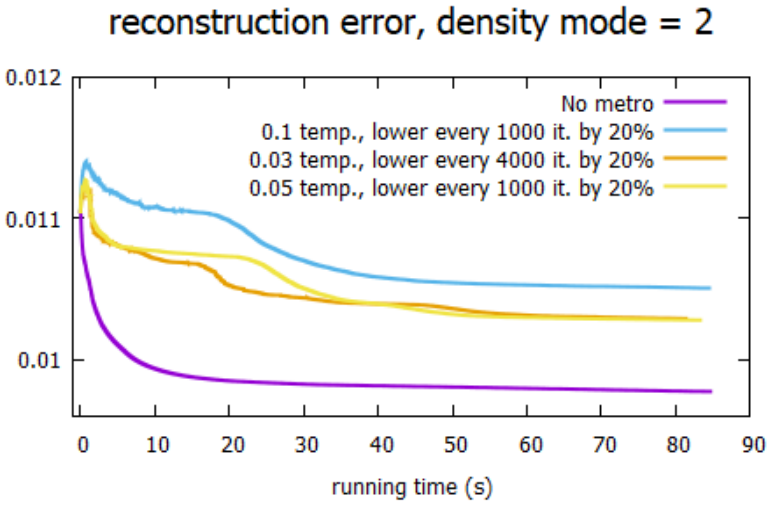

Fig. 4.4: Error curves with different Metropolis parameters in comparison.

As seen in figure 4.4, the different curves applying the Metropolis algorithm all follow an expected pattern: At first a lot of changes are accepted so the error rises. After an initial spike, it gets reduced over time. In contrast to the regular error curve, which looks like a hyperbola, the other error curves shape up more flatly. It might be

possible to get better results with more iterations and a wider range of parameters but testing everything out would be a laborious work. While it might seem that only a temperature reduction of 20% was tested, using parameters of 10-90% did not improve the outcome, neither regarding the illustration nor by providing a better end result.

4.2 Real-world data comparison

Different approaches may take a wide range of computation time to finish but the final error might vary in the same way. If an algorithm yields a higher error but finishes way faster, a slower one with a better structure might even contain a better interim result by the time the first one finishes. Additionally, checking if the inherent structure of the ground truth matrix is kept as good as possible, can be interpreted in many ways, depending on the application.

For comparison the *BMaD* framework [47] and the *parabool* [20] algorithm are tested. Even though I needed to make a few changes to use them for my data sets, the default parameters of the optimization are not touched. The paper about *parabool* was given to me internally through university connections. It is not published but the source code is uploaded on dropbox for reviewing [41].

The *BMaD* framework offers different algorithms to choose from. For the evaluation *LocIter* [30] and *DBP* (also called *asso*) [34] are measured. They are evaluated on a PC with a **AMD Ryzen 1600** CPU @3.2 GHz with 6 cores plus simultaneous multithreading.

While different parameters of $CuBin$ potentially change the outcome by a lot, the same default parameters are taken for each data set. As illustrated earlier, the algorithm has three abort conditions with the run time and end error being heavily dependant on these conditions. For this evaluation, the optimization is stopped when every line has been accessed at least **1000** times on average:

```
int maxIterationsNoImp = (std::max(width,height) / linesAtOnce + 1) * 1000;
```

For the PRNG used in $CuBin$, a different seed is taken each run. Instead of setting it to a fixed number like 41, the current UNIX time is taken. This ensures different results each run to them take an average from.

The results of $CuBin$ seen in table 4.3 are very satisfying. While keeping a competitive reconstruction error in comparison to *parabool, LocIter* and *DBP*, the running

	Alg.	APatio	CPolitics	AOffice	AMusic	MNIST	Movie1m	ATools	AToys	Movie10m
E_{rec} $(\cdot 10^{-3})$	CuBin	7.33	24.72	4.21	2.95	78.26	32,13	0.73	0.67	9.77
	parabool	7.45	24.90	4.23	2.97	79.92	32.79	0.73	0.67	9.99
	LocIter	7.34	26.03	4.23	2.97	104.4	35.60	0.73	0.68	dnf
	DBP	6.50	23.45	3.94	2.82	93.1	33.24	0.71	0.61	dnf
Run time (s)	CuBin	0.5	5.6	2.2	3.4	40	34	16	20	5,380
	parabool	2.1	10	8.3	16	53	119	113	167	6,180
	LocIter	6.2	46	123	228	1,398	750	5,518	8,692	dnf
	DBP	7.5	59	174	461	167	1,624	35,749	27,648	dnf

Tab. 4.3: Performance on data sets (dnf - did not finish in < 10h).

time was decreased by a lot with the help of CUDA and the high level of parallelism the GPU brings to the table. Evaluating the MovieLens10M data set took considerably longer considering for large data sets, small improvements can be made easier, leading to a higher run time since no abort condition is met early.

However, it is not possible to say with certainty that $CuBin$ would always perform better on real-world applications. Randomness brings uncertainty and additionally, it is impossible to say if the inherent structure of the reconstructed matrix is better or worse.

Nevertheless, the parallelization of the algorithm works well and the GPU is utilized to its full extent. These calculations were executed five times each (except for the very long ones) and the average was taken. All measured times for every algorithm can be seen on GitHub. Similar to *parabool*, the algorithm produces the best reconstruction errors when working on relatively dense matrices like *MNIST* and *MovieLens1m*. The error differences to *LocIter* and *DBP* are the highest on these data sets.

The CPU implementation of $CuBin$ has proven to be not usable for real-world applications. The run time exceeds the one by the GPU variant by a factor of 50 to almost 250 even with a parallelization by openMP. Even though the CPU implementation was executed by an Intel i7-3930K, not representing the most current CPUs, the run time is expected to be considerably slower even when using a CPU comparable to the GTX 1080.

4.3 Future work

Due to different factors, mostly time constraints, I was not able to address all issues and possible extensions in the way I wanted to. When considering possible additions for $CuBin$ to handle data sets with sizes in gigabyte or terabyte range, the storage of the ground truth matrix C needs to be addressed first, for example by storing it in a truly sparse fashion. While saving 32 entries of the matrix as one integer works decently and even better for sparsities over $1/32 = 3.125\%$, the space needed for larger matrices with lower densities exceeds the limits of $CuBin's$ possibilities. This is also the reason why the most recent MovieLens20M data set could not be evaluated - the global memory on the GTX 1080 was simply not large enough.

While the use of the Metropolis algorithm did not yield desirable results, it is still important to note that this does not mean it is useless. With more time and a few tweaks to the parameters, it could have led to a better factorization for the price of a longer computation time.

The last addition for $CuBin$, which would potentially result in a better structure of the factor matrices, consists of using different weights for the error measurement of applied changes. As also noted by Dost *et al.* in the *parabool* work, the flip of a 1 to a 0 should be penalized harder using a weight factor. Since the common purpose of Boolean Matrix Factorization is the implementation of some kind of recommender system by finding specific products that are likely to be desired by unique users, the primary goal is to set a (hopefully correct) *user, product* combination from 0 to 1 instead of setting another user's already set 1 to 0. Additionally, there exists a tendency for the algorithm to potentially optimize the factors to all-zero matrices where the final error would equal the density of the given matrix, strengthening the need for a weighted error measurement.

5 Conclusion

Algorithms for Boolean Matrix Factorization exist en masse but most are limited to a single core implementation and thus only suitable for small matrices. The purposed algorithm, **Cu**da-accelerated **Bin**ary Matrix Factorization or short, $CuBin$, solves this issue using the high amount of cores and parallelism of modern NVIDIA GPUs to efficiently implement an optimization based on a random approach. For a given dimension parameter k, two factor matrices are built and optimized by taking multiple rows and columns in parallel to optimize by randomizing and trying out changes. $CuBin$ was evaluated using real-world data sets from different sources with diverse structures and compared with other existing algorithms. The evaluation has shown that the reconstruction error is similar while the completion time was way lower, making it easier and more practical to factorize large and sparse matrices with millions of entries.

For even larger matrices, the algorithm can be extended to support multiple GPUs or even clusters. While this would only be worthwhile for matrix dimensions from hundred thousands upwards, data sets of this margin are no longer uncommon today. The most prominent example would be a streaming platform like Netflix or Amazon Prime Video. Netflix even hosted a competition to improve their recommendation algorithms. The demand for fast and effective Boolean matrix factorization therefore exists.

Further advances to algorithms in regard to the initialization, error management and other tweaks to receive a higher quality factorization leave a lot of room for additional refinement.

Appendix

6

6.1 Source code

```
int main(int argc, char **argv) {
    cudaProfilerStart();
    /* ./a.out  [data]
                [updateStep]
                [lines at once]
                [threshold]
                [gpu iterations]
                [number of changed each line gets without change before aborting]
                [startingTemperature]
                [iterations till temperature is reduced]
                [factor the temperature is lowered]
    */
    if(argc == 1) return 0;
    std::string filename = argv[1];
    int updateStep = argc > 2 ? atoi(argv[2]) : 1000;
    int linesAtOnce = argc > 3 ? atoi(argv[3]) : 4000;
    float threshold = argc > 4 ? atof(argv[4]) : 0;
    int gpuiterations = argc > 5 ? atoi(argv[5]) : 10000000;
    int everyLineChanged = argc > 6 ? atoi(argv[6]) : 1000;
    float temperature = argc > 7 ?  atof(argv[7]) : 0;
    int iterationsTillReduced = argc > 8 ? (atoi(argv[8]) > 0? atoi(argv[8]) : INT_MAX) : INT_MAX;
    float tempFactor = argc > 9 ? atof(argv[9]) : 0.99;
    // int seed = argc > 10 ? atoi(argv[10]) : 41;
    int seed = (unsigned long)time(NULL) % UINT32_MAX;
    fast_kiss_state32_t state = get_initial_fast_kiss_state32(seed);

    // Discard first 100000 entries of PRNG
    for (int i = 0; i < 100000; i++)
        fast_kiss32(&state);

    // Read file and save matrix in C0 and d_C0
    uint32_t *C0, *d_C0;
    int width, height;
    double density;

    // COO coordinates
    string ending = "data";
    if (endsWith(filename, ending)) {
        readInputFileData(&C0, &d_C0, &width, &height, &density, filename);
    } else {
        cout << "Wrong data file" << endl;
        return 0;
    }

    // Initialize Texture Memory with C0
    // initializeTextureMemory(&C0, width, height);

    // Initialize Ab, Bb, d_Ab, d_Bb, all bitwise used matrices
    uint32_t *Ab, *Bb;
    uint32_t *d_Ab, *d_Bb;
    initializeFactors(&Ab, &Bb, &d_Ab, &d_Bb, width, height, density, &state);
    // A and B now initialized on device and host

    // Calculate original error, save it two times each one for GPU, one for CPU
    int error_C0_C_start = 0;
    int error_C0_C       = 0;
    int *d_error_C0_C_start, *d_error_C0_C;
    cudaMalloc((void **) &d_error_C0_C, sizeof(int));                                   CUERR
    cudaMalloc((void **) &d_error_C0_C_start, sizeof(int));                             CUERR
    cudaMemcpy(d_error_C0_C_start, &error_C0_C_start, sizeof(int), cudaMemcpyHostToDevice);  CUERR

    computeStartError(d_Ab, d_Bb, d_C0, width, height, &d_error_C0_C_start, &error_C0_C_start);
```

```
cudaMemcpy(d_error_C0_C, d_error_C0_C_start, sizeof(int), cudaMemcpyDeviceToDevice);
error_C0_C = error_C0_C_start;
// Now the starting errors are in stored in 4 values
// error_C0_C_start, error_C0_C on CPU and GPU


// MAIN PART
// on GPU
int iterations = 0;
int lineToBeChanged;
int iterationsNoImp = 0;
int error_C0_C_before = error_C0_C;
threshold *= (width*height);
#ifdef PERF
vector<double> errorVector; // vector for error measurement
vector<int> impVector;
#endif

// Every line and row changed x times before aborting because of no improvement
int maxIterationsNoImp = (std::max(width,height) / linesAtOnce + 1) * everyLineChanged;
printf("\n- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
printf("- - - - Starting %i GPU iterations, showing error every %i steps - - - -\n", gpuiterations, updateStep);
printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
TIMERSTART(GPUKERNELLOOP)
while (threshold < error_C0_C && iterations < gpuiterations && iterationsNoImp < maxIterationsNoImp) {
    iterations++;
    if (iterations % iterationsTillReduced == 0)
        temperature *= tempFactor;

    // Pull error from GPU to show it
    if (iterations % updateStep == 0) {
        #ifndef PERF
        printf("Current error: %f\n", error_C0_C / (double) (width * height));
        #endif
        // For debugging
        // checkDistance(d_Ab, d_Bb, d_C0, height, width);
    }

    // Change col
    lineToBeChanged = ((unsigned int) fast_kiss32(&state)) % width;
    vectorMatrixMultCompareCol <<< min(linesAtOnce, width), THREADSPERBLOCK >>>
                                  (d_Ab, d_Bb, d_C0, width, height, lineToBeChanged, d_error_C0_C,
                                  ((fast_kiss32(&state) + iterations) % UINT32_MAX), temperature);        CUERR
    cudaDeviceSynchronize();                                                                              CUERR

    // Change row
    lineToBeChanged = ((unsigned int) fast_kiss32(&state)) % height;
    vectorMatrixMultCompareRow <<< min(linesAtOnce, height), THREADSPERBLOCK >>>
                                  (d_Ab, d_Bb, d_C0, width, height, lineToBeChanged, d_error_C0_C,
                                  ((fast_kiss32(&state) + iterations) % UINT32_MAX), temperature);        CUERR
    cudaDeviceSynchronize();                                                                              CUERR

    cudaMemcpy(&error_C0_C, d_error_C0_C, sizeof(int), cudaMemcpyDeviceToHost);                           CUERR

    // To check how many iterations are needed for improvement
    if (error_C0_C_before - error_C0_C == 0) {
        iterationsNoImp++;
    } else {
        #ifdef PERF
        impVector.push_back(iterationsNoImp);
        #endif
        iterationsNoImp = 0;
    }
    error_C0_C_before = error_C0_C;
    #ifdef PERF
    errorVector.push_back(error_C0_C / (double) (width * height));
    #endif
}
// Pull final error from GPU
cudaMemcpy(&error_C0_C, d_error_C0_C, sizeof(int), cudaMemcpyDeviceToHost);                               CUERR
printf("- - - - - - - - -\n");
printf("Final Error on GPU: %f, %i wrong entries\n", error_C0_C / (double) (height * width), error_C0_C);
TIMERSTOP(GPUKERNELLOOP)

// Aftertest GPU
#ifdef TEST
aftertestGPU(d_Ab, d_Bb, d_C0, width, height);
#endif
```

```
    // Write result matrices to files
    #ifndef PERF
    writeToFiles(d_Ab, d_Bb, width, height);
    #endif

    // Output CSV files for plotting
    #ifdef PERF
    string writeFile = string("perf.csv");
    ofstream myfile(writeFile);
    if (myfile.is_open()) {
        myfile << "x,y\n";
        for (int i = 0; i < errorVector.size(); i++) {
            myfile << (i * timeGPUKERNELLOOP / (double) iterations) / (double) 1000 << "," << errorVector[i] << "\n";
        }
    }
    writeFile = string("update.csv");
    ofstream myfile1(writeFile);
    if (myfile1.is_open()) {
        myfile1 << "x,y\n";
        for (int i = 0; i < impVector.size(); i++) {
            myfile1 << i << "," << impVector[i] << "\n";
        }
    }
    #endif

    #ifdef CPU
    // CPU COMPUTATION
    CPUcomputation(Ab, Bb, C0, width, height, error_C0_C_start, 42, updateStep, threshold, linesAtOnce);

    // Aftertest CPU
    aftertestCPU(Ab, Bb, d_Ab, d_Bb, C0, width, height);
    #endif

    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");

    // Cleaning up
    cudaProfilerStop();
    cudaDeviceReset();
    free(Ab);
    free(Bb);
    free(C0);
    cudaFree(d_Ab);
    cudaFree(d_Bb);
    cudaFree(d_C0);
    cudaFree(d_error_C0_C);
    cudaFree(d_error_C0_C_start);
    return 0;
}

__inline__ __device__
int warpReduceSum(int val) {
    for (int offset = warpSize / 2; offset > 0; offset /= 2)
        val += __shfl_down(val, offset);
    return val;
}

// [A] row Change kernel
__global__ void
vectorMatrixMultCompareRow( uint32_t *A, uint32_t *B, uint32_t *C,
                            int width, int height,
                            int startrow, int *global_error,
                            uint32_t seed, float temperature) {

    int rowToBeChanged = (blockIdx.x + startrow) % height;
    int cTruthEntry;
    int cEntryOld;
    int cEntryNew;
    int error_thread;
    int intId;
    int intLane;
    uint32_t randomNumber;
    uint32_t currentColThread;
    uint32_t currentRow;
    uint32_t currentRow_changed;
    float metro;
    __shared__ fast_kiss_state32_t state;
    __shared__ int reductionArray[32];
    __shared__ uint32_t shared_currentRow_changed;

    currentRow = A[rowToBeChanged];
```

```
    if (threadIdx.x == 0) {
        shared_currentRow_changed = currentRow; // load row to be changed in shared memory

        state = get_initial_fast_kiss_state32((seed + blockIdx.x) % UINT32_MAX);
        randomNumber = fast_kiss32(&state);
        #pragma unroll
        for (int i = 0; i < DIM_PARAM; i++)
            shared_currentRow_changed ^= (randomNumber >> i) & 11 ? (0 << 32 - 1 - i) : (1 << 32 - 1 - i);
    }

    __syncthreads();

    currentRow_changed = shared_currentRow_changed;
    error_thread = 0;
    for (int i = 0; i <= ((width - 1) / blockDim.x); i++) {
        if ((i * blockDim.x + threadIdx.x) < width) {
            currentColThread = B[i * blockDim.x + threadIdx.x];
            intId = (((i * blockDim.x + threadIdx.x) * height) + rowToBeChanged) / 32;
            intLane = (((i * blockDim.x + threadIdx.x) * height) + rowToBeChanged) % 32;
            cTruthEntry = (C[intId] >> 32 - intLane - 1) & 1;

            cEntryOld = (currentRow         & currentColThread) > 0 ? 1 : 0;
            cEntryNew = (currentRow_changed & currentColThread) > 0 ? 1 : 0;
            error_thread += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
                            ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
        }
    }
    __syncthreads();

    // Reduction across block
    int lane = threadIdx.x % warpSize;
    int wid = threadIdx.x / warpSize;
    error_thread = warpReduceSum(error_thread);
    if (lane == 0) reductionArray[wid] = error_thread;
    __syncthreads();
    error_thread = (threadIdx.x < blockDim.x / warpSize) ? reductionArray[lane] : 0;
    if (wid == 0) error_thread = warpReduceSum(error_thread);
    // Thread with threadIdx.x==0 now has error in error_thread

    // Thread 0 checks if new low has been found and applies if necessary
    if (threadIdx.x == 0) {
        if (error_thread < 0) {
            A[rowToBeChanged] = shared_currentRow_changed;
            atomicAdd(global_error, error_thread);
        } else { // Metropolis-Hastings algorithm
            randomNumber = fast_kiss32(&state) / (double) UINT32_MAX;
            metro = temperature > 0.0f ? fminf(1, expf((double) -error_thread / temperature)) : 0 ;
            if (randomNumber < metro) {
                A[rowToBeChanged] = shared_currentRow_changed;
                atomicAdd(global_error, error_thread);
            }
        }
    }
}

// [B] col change kernel
//
__global__ void
vectorMatrixMultCompareCol(uint32_t *A, uint32_t *B, uint32_t *C,
                           int width, int height,
                           int startcol, int *global_error,
                           uint32_t seed, float temperature) {

    int colToBeChanged = (blockIdx.x + startcol) % width;
    int cTruthEntry;
    int cEntryOld;
    int cEntryNew;
    int error_thread;
    int intId;
    int intLane;
    uint32_t randomNumber;
    uint32_t currentRowThread;
    uint32_t currentCol;
    uint32_t currentCol_changed;
    float metro;
    __shared__ fast_kiss_state32_t state;
    __shared__ int shared[32];
    __shared__ uint32_t shared_currentCol_changed;

    currentCol = B[colToBeChanged];
```

```
    if (threadIdx.x == 0) {
        shared_currentCol_changed = currentCol; // load row to be changed in shared memory

        state = get_initial_fast_kiss_state32((seed + blockIdx.x) % UINT32_MAX);
        randomNumber = fast_kiss32(&state);
        #pragma unroll
        for (int i = 0; i < DIM_PARAM; i++)
            shared_currentCol_changed ^= (randomNumber >> i) & 11 ? (0 << 32 - 1 - i) : (1 << 32 - 1 - i);
    }
    __syncthreads();

    currentCol_changed = shared_currentCol_changed;
    error_thread = 0;
    for (int i = 0; i <= ((height - 1) / blockDim.x); i++) {
        if ((i * blockDim.x + threadIdx.x) < height) {
            currentRowThread = A[i * blockDim.x + threadIdx.x];
            intId = ((colToBeChanged * height) + (i * blockDim.x + threadIdx.x)) / 32;
            intLane = ((colToBeChanged * height) + (i * blockDim.x + threadIdx.x)) % 32;
            cTruthEntry = (C[intId] >> 32 - intLane - 1) & 1;

            cEntryOld = (currentCol         & currentRowThread) > 0 ? 1 : 0;
            cEntryNew = (currentCol_changed & currentRowThread) > 0 ? 1 : 0;
            error_thread += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
                            ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
        }
    }
    __syncthreads();

    // Reduction across block
    int lane = threadIdx.x % warpSize;
    int wid = threadIdx.x / warpSize;
    error_thread = warpReduceSum(error_thread);
    if (lane == 0) shared[wid] = error_thread;
    __syncthreads();
    error_thread = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;
    if (wid == 0) error_thread = warpReduceSum(error_thread);
    // Thread with threadIdx.x==0 now has error in error_thread

    // Thread 0 checks if new low has been found and applies if necessary
    if (threadIdx.x == 0) {
        if (error_thread < 0) {
            B[colToBeChanged] = shared_currentCol_changed;
            atomicAdd(global_error, error_thread);
        }  else { // Metropolis-Hastings algorithm
            randomNumber = fast_kiss32(&state) / (double) UINT32_MAX;
            metro = temperature > 0.0f ? fminf(1, expf((double)-error_thread / temperature)) : 0 ;
            if (randomNumber < metro) {
                B[colToBeChanged] = shared_currentCol_changed;
                atomicAdd(global_error, error_thread);
            }
        }
    }
    __syncthreads();
}

// Start error kernel
__global__ void computeFullError(   uint32_t *A, uint32_t *B, uint32_t *C,
                                    int width, int height, int *distance_test) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int lineSum;
    int truthEntry;
    int intId;
    int intLane;
    int error_thread;
    //__shared__ volatile int shared_distance[THREADSPERBLOCK];
    //shared_distance[threadIdx.x] = 0;
    __shared__ int reductionArray[32];


    if (tid < width) {
        error_thread = 0;
        for (int j = 0; j < height; j++) {
            lineSum = (A[j] & B[tid]) > 0 ? 1 : 0;
            intId = (tid * height + j) / 32;
            intLane = (tid * height + j) % 32;
            truthEntry = (C[intId] >> 32 - intLane - 1) & 1;
            error_thread += ((lineSum - truthEntry) * (lineSum - truthEntry));
        }
        __syncthreads();
```

```
        // Reduction across block
        int lane = threadIdx.x % warpSize;
        int wid = threadIdx.x / warpSize;
        error_thread = warpReduceSum(error_thread);
        if (lane == 0) reductionArray[wid] = error_thread;
        __syncthreads();
        error_thread = (threadIdx.x < blockDim.x / warpSize) ? reductionArray[lane] : 0;
        if (wid == 0) error_thread = warpReduceSum(error_thread);
        // Thread with threadIdx.x==0 now has error in error_thread

        if (threadIdx.x == 0)
            atomicAdd(distance_test, error_thread);
        __syncthreads();

    }
}

// Each thread one entry of a row
__global__ void matrixMultiply( uint32_t *A, uint32_t *B, uint32_t *C,
                                int width, int height) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < width)
        for (int i = 0; i < height; i++)
            C[i * width + tid] = (A[i] & B[tid]) > 0 ? 1 : 0;
}

// https://gist.github.com/wh5a/4313739
__global__ void matrixMultiplyInt(  int * A0, int * B0, uint32_t * C0,
                                    int m, int k, int n) {
    __shared__ int ds_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ int ds_B[TILE_WIDTH][TILE_WIDTH];
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int sum = 0;

    for(int t = 0; t < (n - 1) / TILE_WIDTH + 1; t++) {
        if(row < m && t * TILE_WIDTH + tx < n)
            ds_A[ty][tx] = roundf(A0[row * n + t * TILE_WIDTH + tx]);
        else
            ds_A[ty][tx] = 0.0;
        if(t * TILE_WIDTH + ty < n && col < k)
            ds_B[ty][tx] = roundf(B0[(t * TILE_WIDTH + ty) * k + col]);
        else
            ds_B[ty][tx] = 0.0;
        __syncthreads();
        for(int i = 0; i < TILE_WIDTH; i++){
            sum += ds_A[ty][i] * ds_B[i][tx];
        }
        __syncthreads();
    }
    if(row < m && col < k)
        C0[col + row * k] = min(1, sum);

}

void readInputFileData( uint32_t **C0, uint32_t **d_C0,
                    int *width, int *height,
                    double *density, string filename) {
    int x, y;
    int nonzeroelements = 0;
    int sizeC;
    int intID;
    int intLane;
    ifstream infile;
    string linestring;
    string field;

    // First line: #height,#width,#non-zero-elements
    infile.open(filename);
    getline(infile, linestring);
    stringstream sep(linestring);
    getline(sep, field, ',');
    (*height) = stoi(field, nullptr);
    getline(sep, field, ',');
    (*width) = stoi(field, nullptr);

    // Malloc for C0 and d_C0
    sizeC = (int) ceil((*width) * (*height) / (double) 32.0);
```

```
    (*C0) = (uint32_t *) malloc(sizeof(uint32_t) * sizeC);
    cudaMalloc((void **) d_C0, sizeof(uint32_t) * sizeC);                                   CUERR

    // Set all entries 0
    for (int i = 0; i < sizeC; i++)
        (*C0)[i] = 0;

    // Read rest of file
    while (getline(infile, linestring)) {
        stringstream sep1(linestring);
        string fieldtemp;
        getline(sep1, fieldtemp, ',');
        y = stoi(fieldtemp, nullptr);
        getline(sep1, fieldtemp, ',');
        x = stoi(fieldtemp, nullptr);
        intID = (x * (*height) + y) / 32;
        intLane = (x * (*height) + y) % 32;
        (*C0)[intID] |= 1 << 32 - intLane - 1;
        nonzeroelements++;
    }

    (*density) = (double) nonzeroelements / ((*width) * (*height));

    cudaMemcpy((*d_C0), (*C0), sizeof(uint32_t) * sizeC, cudaMemcpyHostToDevice);           CUERR

    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
    printf("READING OF .DATA FILE COMPLETE\n");
    printf("Read height: %i\nRead width: %i\nNon-zero elements: %i\nDensity: %f\n",
           (*height), (*width), nonzeroelements, (*density));
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
}

// https://stackoverflow.com/questions/874134/find-if-string-ends-with-another-string-in-c
bool endsWith(const string& s, const string& suffix) {
    return s.rfind(suffix) == (s.size()-suffix.size());
}

void computeStartError(uint32_t *d_Ab, uint32_t *d_Bb, uint32_t *d_Cb,
                       int width, int height,
                       int **d_distance_C0_C_start, int *distance_C0_C_start) {
    TIMERSTART(ERRORFIRST)

    computeFullError <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                       (d_Ab, d_Bb, d_Cb, width, height, (*d_distance_C0_C_start));         CUERR

    cudaMemcpy(distance_C0_C_start, (*d_distance_C0_C_start), sizeof(int), cudaMemcpyDeviceToHost);   CUERR

    printf("Starting error between AxB=C and C0: %f \n",
           (*distance_C0_C_start) / ((double) width * height));
    TIMERSTOP(ERRORFIRST)
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
}

// Only for debugging
void checkDistance(uint32_t *d_Ab, uint32_t *d_Bb, uint32_t *d_C0, int height, int width) {
    int distance_test;
    int *d_distance_test;
    cudaMalloc((void **) &d_distance_test, sizeof(int));                                    CUERR
    distance_test = 0;
    cudaMemcpy(d_distance_test, &distance_test, sizeof(int), cudaMemcpyHostToDevice);       CUERR

    computeFullError <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                         (d_Ab, d_Bb, d_C0, height, width, d_distance_test);                CUERR

    cudaMemcpy(&distance_test, d_distance_test, sizeof(int), cudaMemcpyDeviceToHost);       CUERR
    printf("Real Error: %f\n", (distance_test/(double)(height*width)));
}


// Initialization of A and B
void initializeFactors( uint32_t **Ab, uint32_t **Bb,
                        uint32_t **d_Ab, uint32_t **d_Bb,
                        int width, int height,
                        float density, fast_kiss_state32_t *state) {

    (*Ab) = (uint32_t *) malloc(sizeof(uint32_t) * height);
    (*Bb) = (uint32_t *) malloc(sizeof(uint32_t) * width);
    cudaMalloc((void **) d_Ab, sizeof(uint32_t) * height);                                  CUERR
    cudaMalloc((void **) d_Bb, sizeof(uint32_t) * width);                                   CUERR
```

```
    // Initialize A and B and copy to device
    bool threshold;
    for (int i = 0; i < height; i++) {
        (*Ab)[i] = 0;
        #pragma unroll
        for (int j = 0; j < DIM_PARAM; j++) {
            switch(INITIALIZATIONMODE) {
                case 1: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < (sqrt(1 - pow(1 - density, 1 / (double) DIM_PARAM)));
                                    break;
                case 2: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < (density / (double) 100);
                                    break;
                case 3: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < density;
                                    break;
            }
            (*Ab)[i] |= threshold ? 1 << (32 - j - 1) : 0 ;
        }
    }
    for (int i = 0; i < width; i++) {
        (*Bb)[i] = 0;
        #pragma unroll
        for (int j = 0; j < DIM_PARAM; j++) {
            switch(INITIALIZATIONMODE) {
                case 1: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < (sqrt(1 - pow(1 - density, 1 / (double) DIM_PARAM)));
                                    break;
                case 2: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < (density / (double) 100);
                                    break;
                case 3: threshold = (fast_kiss32(state) / (double) UINT32_MAX)
                                    < density;
                                    break;
            }
            (*Bb)[i] |= threshold ? 1 << (32 - j - 1) : 0 ;
        }
    }

    // copy to device arrays
    cudaMemcpy((*d_Ab), (*Ab), sizeof(uint32_t) * height, cudaMemcpyHostToDevice);                    CUERR
    cudaMemcpy((*d_Bb), (*Bb), sizeof(uint32_t) * width, cudaMemcpyHostToDevice);                     CUERR

    printf("Initialization of A and B complete\n");
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
}

// Used for debugging and checking for correctness, not optimized
void aftertestGPU(  uint32_t *d_Ab, uint32_t *d_Bb, uint32_t *d_COb,
                    int width, int height) {
    TIMERSTART(aftertestGPU)
    uint32_t *d_C_test_GPU;
    uint32_t *C_test_GPU;

    int *A, *B, *CO;
    uint32_t *Ab, *Bb, *COb;
    int *d_A, *d_B;
    float densityA, densityB;
    uint32_t counterDensity;
    A = (int*) malloc(sizeof(int) * DIM_PARAM * height);
    Ab = (uint32_t*) malloc(sizeof(uint32_t) * height);
    B = (int*) malloc(sizeof(int) * width * DIM_PARAM);
    Bb = (uint32_t*) malloc(sizeof(uint32_t) * width);
    CO = (int*) malloc(sizeof(int) *width * height);
    COb = (uint32_t*) malloc(sizeof(uint32_t) * ((long long) (height * width) / 32.0 + 1));

    cudaMalloc((void**)&d_A, sizeof(int) * DIM_PARAM * height);                                       CUERR
    cudaMalloc((void**)&d_B, sizeof(int) * width * DIM_PARAM);                                        CUERR

    cudaMemcpy(Ab, d_Ab, sizeof(uint32_t) * height, cudaMemcpyDeviceToHost);                          CUERR
    cudaMemcpy(Bb, d_Bb, sizeof(uint32_t) * width, cudaMemcpyDeviceToHost);                           CUERR
    cudaMemcpy(COb, d_COb, sizeof(uint32_t) * ((long long)(height*width) / 32.0 + 1),
                    cudaMemcpyDeviceToHost);                                                          CUERR

    counterDensity = 0;
    for(int i = 0; i < height; i++){
        for(int j = 0; j < DIM_PARAM; j++){
            A[i * DIM_PARAM + j] = (Ab[i] >> 32 - j - 1) & 1;
            if (A[i*DIM_PARAM + j]) counterDensity++;
        }
```

```
    }
    densityA = counterDensity / (double) (height*DIM_PARAM);

    counterDensity = 0;
    for(int i = 0; i < width; i++) {
        for(int j = 0; j < DIM_PARAM; j++) {
            B[j * width + i] = (Bb[i] >> 32 - j - 1) & 1;
            if (B[j * width + i]) counterDensity++;
        }
    }
    densityB = counterDensity / (double) (DIM_PARAM * width);

    int intId;
    int intLane;
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
             intId = (i * height + j) / 32;
             intLane = (i * height + j) % 32;
             C0[j * width + i] = (C0b[intId] >> 32 - intLane - 1) & 1;
        }
    }

    cudaMemcpy(d_A, A, sizeof(int)*height*DIM_PARAM, cudaMemcpyHostToDevice);                          CUERR
    cudaMemcpy(d_B, B, sizeof(int)*width*DIM_PARAM, cudaMemcpyHostToDevice);                           CUERR

    // Doing a check two times: once with A,B and once with Ab,Bb just to make sure
    // First check
    C_test_GPU = (uint32_t *) malloc(sizeof(uint32_t) * width * height);
    cudaMalloc((void **) &d_C_test_GPU, sizeof(uint32_t) * height * width);                            CUERR

    matrixMultiply <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                        (d_Ab, d_Bb, d_C_test_GPU, width, height);                                     CUERR

    cudaMemcpy(C_test_GPU, d_C_test_GPU, sizeof(uint32_t) * height * width,
                    cudaMemcpyDeviceToHost);                                                           CUERR

    int error_test_GPU = 0;
    for (int i = 0; i < height * width; i++)
        error_test_GPU += (((C0[i] - C_test_GPU[i]) * (C0[i] - C_test_GPU[i])));

    // Second check with regular matrix multiplication
    dim3 dimGrid((width - 1) / TILE_WIDTH + 1, (height - 1) / TILE_WIDTH + 1, 1);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    matrixMultiplyInt <<< dimGrid, dimBlock >>>
                            (d_A, d_B, d_C_test_GPU, height, width, DIM_PARAM);                        CUERR

    cudaMemcpy(C_test_GPU, d_C_test_GPU, sizeof(int) * height * width, cudaMemcpyDeviceToHost);        CUERR
    int error_test_GPU_2 = 0;
    for (int i = 0; i < height * width; i++)
        error_test_GPU_2 += (((C0[i] - C_test_GPU[i]) * (C0[i] - C_test_GPU[i])));

    TIMERSTOP(aftertestGPU)
    printf("Aftertest error between C0 and C on GPU (bitwise): %i\n", error_test_GPU);
    printf("Aftertest error between C0 and C on GPU (float): %i\n", error_test_GPU_2);
    printf("Density A: %f, Density B: %f\n", densityA, densityB);

}

// Write result matrix in file
void writeToFiles(uint32_t* d_Ab, uint32_t* d_Bb, int width, int height){
    int *A, *B;
    uint32_t *Ab, *Bb;
    A = (int*) malloc(sizeof(int) * DIM_PARAM * height);
    Ab = (uint32_t*) malloc(sizeof(uint32_t) * height);
    B = (int*) malloc(sizeof(int) * width * DIM_PARAM);
    Bb = (uint32_t*) malloc(sizeof(uint32_t) * width);

    cudaMemcpy(Ab, d_Ab, sizeof(uint32_t) * height, cudaMemcpyDeviceToHost);                           CUERR
    cudaMemcpy(Bb, d_Bb, sizeof(uint32_t) * width, cudaMemcpyDeviceToHost);                            CUERR

    for(int i = 0; i < height; i++)
        for(int j = 0; j < DIM_PARAM; j++)
            A[i * DIM_PARAM + j] = (Ab[i] >> 32 - j - 1) & 1;

    for(int i = 0; i < width; i++)
        for(int j = 0; j < DIM_PARAM; j++)
            B[j * width + i] = (Bb[i] >> 32 - j - 1) & 1;

    time_t rawtime;
    struct tm * timeinfo;
```

```
    char buffer[80];

    time (&rawtime);
    timeinfo = localtime(&rawtime);

    strftime(buffer, sizeof(buffer), "%X", timeinfo);
    string str = buffer;

    string a = string("A_") + buffer + string(".data");
    string b = string("B_") + buffer + string(".data");

    ofstream myfile(a);
    if (myfile.is_open()){
        myfile << height << "," << DIM_PARAM << "\n";
        for (int i = 0; i < height; i++){
            for (int j = 0; j < DIM_PARAM; j++){
                myfile << A[i * DIM_PARAM + j] << ((j != DIM_PARAM - 1) ? "," : "");
            }
            myfile << "\n";
        }
        myfile.close();
    }

    ofstream myfile2(b);
    if(myfile2.is_open()){
        myfile2 << DIM_PARAM << "," << width << "\n";
        for (int i = 0; i<DIM_PARAM; i++){
            for (int j = 0; j < width; j++){
                myfile2 << B[i * width + j] << ((j != width - 1) ? "," : "");
            }
            myfile2 << "\n";
        }
        myfile2.close();
    }
    cout << "Writing to files \"" << a << "\" and \"" << b << "\" complete" << endl;
}

// CPU computation
void CPUcomputation(uint32_t *Ab, uint32_t *Bb, uint32_t *C0,
                    int width, int height,
                    int startDistance, uint32_t seed, int updateStep,
                    float threshold, int linesAtOnce) {

    int *hDistance = &startDistance;
    fast_kiss_state32_t state;
    state = get_initial_fast_kiss_state32(seed);
    int toBeChanged;
    int iterations = 0;
    int iterationsNoImp = 0;
    int maxIterationsNoImp = (std::max(width,height) / linesAtOnce + 1) * 1000;
    int error_before = startDistance;

    TIMERSTART(CPUcomputation)
    printf("\n- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
    printf("- - - - Starting CPU opimization, showing error every %i steps - - - - -\n", updateStep);
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
    while (*hDistance > threshold && iterations < CPUITERATIONS && iterationsNoImp < maxIterationsNoImp) {

        iterations++;
        if (iterations % updateStep == 0)
            printf("Current Distance: %i\n", *hDistance);

        // Change row
        toBeChanged = ((unsigned int) fast_kiss32(&state)) % height;
        CPUvectorMatrixMultCompareRow(Ab, Bb, C0, width, height, toBeChanged, hDistance, &state, linesAtOnce);

        // Change col
        toBeChanged = ((unsigned int) fast_kiss32(&state)) % width;
        CPUvectorMatrixMultCompareCol(Ab, Bb, C0, width, height, toBeChanged, hDistance, &state, linesAtOnce);

        if (error_before - *hDistance == 0) {
            iterationsNoImp++;
        } else {
            iterationsNoImp = 0;
        }
        error_before = *hDistance;

    }
    printf("- - - - - - - - -\n");
    printf("End Distance on CPU: %i, Number of Iterations: %i,  Error remaining: %f\n",
```

```
                *hDistance, iterations, *hDistance / (double) (height * width));
    TIMERSTOP(CPUcomputation)
}

void CPUvectorMatrixMultCompareRow( uint32_t *Ab, uint32_t *Bb,
                                    uint32_t *C0, int width, int height, int startrow,
                                    int *hDistance, fast_kiss_state32_t *state, int rowsAtOnce) {
    int rowToBeChanged = startrow;
    int error;
    int cTruthEntry;
    int cEntryOld;
    int cEntryNew;
    uint32_t currentRow;
    uint32_t currentRow_changed;
    uint32_t randomNumber;

    // Change multiple lines from A
    for (int l = 0; l < rowsAtOnce; l++) {
        rowToBeChanged = (startrow + l) % height;
        currentRow = Ab[rowToBeChanged];
        currentRow_changed = currentRow;

        randomNumber = fast_kiss32(state);
        for (int i = 0; i < DIM_PARAM; i++)
            currentRow_changed ^= (randomNumber >> i) & 11 ?(0 << 21 - 1 - i) : (1 << 32 - 1 - i);

        error = 0;

#pragma omp parallel for private(cEntryOld, cEntryNew, cTruthEntry) reduction(+: error)
        for (int tid = 0; tid < width; tid++) {
            uint32_t currentCol = Bb[tid];
            int intId = (tid * height + rowToBeChanged) / 32;
            int intLane = (tid * height + rowToBeChanged) % 32;
            cTruthEntry = (C0[intId] >> 32 - intLane - 1) & 1;

            cEntryOld = (currentRow         & currentCol) > 0 ? 1 : 0;
            cEntryNew = (currentRow_changed & currentCol) > 0 ? 1 : 0;
            error += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
                     ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
        }

        if (error < 0) {
            Ab[rowToBeChanged] = currentRow_changed;
            *hDistance = *hDistance + error;
        }

    }
}

void CPUvectorMatrixMultCompareCol( uint32_t *Ab, uint32_t *Bb, uint32_t *C0,
                                    int width, int height, int startcol,
                                    int *hDistance, fast_kiss_state32_t *state, int rowsAtOnce) {
    int colToBeChanged = startcol;
    int error;
    int cTruthEntry;
    int cEntryOld;
    int cEntryNew;
    uint32_t currentCol;
    uint32_t currentCol_changed;
    uint32_t randomNumber;

    // Change multiple cols from B
    for (int l = 0; l < rowsAtOnce; l++) {
        colToBeChanged = (colToBeChanged + l) % width;
        currentCol = Bb[colToBeChanged];
        currentCol_changed = currentCol;

        randomNumber = fast_kiss32(state);
        for (int i = 0; i < DIM_PARAM; i++)
            currentCol_changed ^= (randomNumber >> i) & 11 ? (0 << 32 - 1 - i) : (1 << 32 - 1 - i);

        error = 0;
        #pragma omp parallel for private(cEntryOld, cEntryNew, cTruthEntry) reduction(+: error)
        for (int tid = 0; tid < height; tid++) {
             uint32_t currentRow = Ab[tid];
            int intId = (colToBeChanged * height + tid) / 32;
            int intLane = (colToBeChanged * height + tid) % 32;
            cTruthEntry = (C0[intId] >> 32 - intLane - 1) & 1;

            cEntryOld = (currentCol         & currentRow) > 0 ? 1 : 0;
```

```
            cEntryNew = (currentCol_changed & currentRow) > 0 ? 1 : 0;
            error += ((cEntryNew - cTruthEntry) * (cEntryNew - cTruthEntry)) -
                     ((cEntryOld - cTruthEntry) * (cEntryOld - cTruthEntry));
        }

        if (error < 0) {
            Bb[colToBeChanged] = currentCol_changed;
            *hDistance = *hDistance + error;
        }
    }
}

// Used for debugging and checking, not optimized
void aftertestCPU(  uint32_t *Ab, uint32_t *Bb, uint32_t *d_Ab, uint32_t *d_Bb, uint32_t *COb,
                    int width, int height) {
    TIMERSTART(aftertestCPU)
    int *A, *B, *C0;
    int *d_A, *d_B;
    uint32_t *C_test_CPU;
    uint32_t *d_C_test_CPU;
    A = (int*)malloc(sizeof(int) * DIM_PARAM * height);
    B = (int*)malloc(sizeof(int) * width * DIM_PARAM);
    C0 = (int*)malloc(sizeof(int) * width * height);
    C_test_CPU = (uint32_t *) malloc(sizeof(uint32_t) * width * height);                CUERR
    cudaMalloc((void**)&d_A, sizeof(int) * DIM_PARAM * height);                         CUERR
    cudaMalloc((void**)&d_B, sizeof(int)*width * DIM_PARAM);                            CUERR
    cudaMalloc((void**) &d_C_test_CPU, sizeof(uint32_t) * height * width);              CUERR


    for(int i=0; i<height;i++)
        for(int j=0;j<DIM_PARAM;j++)
            A[i*DIM_PARAM + j] = (Ab[i] >> 32 - j - 1) & 1;

    for(int i=0;i<width;i++)
        for(int j=0;j<DIM_PARAM;j++)
            B[j*width+i] = (Bb[i] >> 32 - j - 1) & 1;

    int intId;
    int intLane;
    for(int i=0; i<width; i++){
        for(int j=0;j<height;j++){
             intId = (i*height + j) / 32;
             intLane = (i*height + j) % 32;
             C0[j*width + i] = (C0b[intId] >> 32 - intLane - 1) & 1;
        }
    }


    cudaMemcpy(d_A, A, sizeof(uint32_t) * height * DIM_PARAM, cudaMemcpyHostToDevice);  CUERR
    cudaMemcpy(d_B, B, sizeof(uint32_t) * width * DIM_PARAM, cudaMemcpyHostToDevice);   CUERR
    cudaMemcpy(d_Ab, Ab, sizeof(uint32_t) * height, cudaMemcpyHostToDevice);            CUERR
    cudaMemcpy(d_Bb, Bb, sizeof(uint32_t) * width, cudaMemcpyHostToDevice);             CUERR

    // Doing a check two times: once with A,B and once with Ab,Bb just to make sure

    matrixMultiply <<< width / THREADSPERBLOCK + 1, THREADSPERBLOCK >>>
                        (d_Ab, d_Bb, d_C_test_CPU, width, height);                      CUERR

    cudaMemcpy(C_test_CPU, d_C_test_CPU, sizeof(uint32_t) * height * width,
                    cudaMemcpyDeviceToHost);                                            CUERR

    int distance_test_CPU = 0;
    for (int i = 0; i < height * width; i++)
        distance_test_CPU += ((C0[i] - C_test_CPU[i]) * (C0[i] - C_test_CPU[i]));

    dim3 dimGrid((width-1)/TILE_WIDTH+1, (height-1)/TILE_WIDTH+1, 1);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    matrixMultiplyInt <<< dimGrid, dimBlock >>>
                (d_A, d_B, d_C_test_CPU, height, width, DIM_PARAM);                     CUERR
    cudaMemcpy(C_test_CPU, d_C_test_CPU, sizeof(int) * height * width,
                    cudaMemcpyDeviceToHost);                                            CUERR
    int distance_test_CPU_2 = 0;
    for (int i = 0; i < height * width; i++)
        distance_test_CPU_2 += ((C0[i] - C_test_CPU[i]) * (C0[i] - C_test_CPU[i]));

    TIMERSTOP(aftertestCPU)
    printf("Aftertest error between C0 and C on CPU (bitwise): %i\n", distance_test_CPU);
    printf("Aftertest error between C0 and C on CPU (float): %i\n", distance_test_CPU_2);
}
```

Bibliography

[1] C. Bauckhage. „k-Means Clustering Is Matrix Factorization“. In: *ArXiv e-prints* (Dec. 2015). arXiv: 1512.07548 [stat.ML] (cit. on p. 5).

[2] Sven Bergmann, Jan Ihmels, and Naama Barkai. „Iterative signature algorithm for the analysis of large-scale gene expression data“. In: *Phys. Rev. E* 67 (3 Mar. 2003), p. 031902 (cit. on p. 7).

[3] C. Boutsidis and E. Gallopoulos. „SVD based initialization: A head start for nonnegative matrix factorization“. In: *Pattern Recognition* 41.4 (2008), pp. 1350 –1362 (cit. on p. 8).

[4] *CUDA*. https://en.wikipedia.org/wiki/CUDA. Accessed: 2017-01-04 (cit. on p. 12).

[5] *CUDA Differences b/w Architectures and Compute Capability*. http://blog.cuvilib.com/2014/11/12/cuda-differences-bw-architectures-and-compute-capability/. Accessed: 2017-01-30 (cit. on p. 38).

[6] *CUDA Matrix Multiplication with Shared Memory*. https://gist.github.com/wh5a/4313739. Accessed: 2017-01-13 (cit. on p. 33).

[7] *CUDA Matrix Multiplication with Shared Memory*. http://cseweb.ucsd.edu/classes/wi12/cse260-a/Lectures/Lec08.pdf. Accessed: 2017-01-13 (cit. on p. 33).

[8] *CUDA Memory Types*. https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf (cit. on p. 14).

[9] *CUDA toolkit documentation v9.1.85*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2017-01-02 (cit. on pp. 11, 12, 14, 33).

[10] *CUDA Zone*. https://developer.nvidia.com/cuda-zone/. Accessed: 2017-01-01 (cit. on p. 14).

[11] *CUDA Zone*. http://www.cs.cornell.edu/people/pabo/movie-review-data/. Accessed: 2017-01-20 (cit. on p. 35).

[12] *Faster Parallel Reductions on Kepler*. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/. Accessed: 2017-01-06 (cit. on p. 22).

[13] M. J. Flynn. „Some Computer Organizations and Their Effectiveness“. In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960 (cit. on p. 11).

[14] *Global Memory Usage and Strategy*. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf. Accessed: 2017-01-02 (cit. on p. 13).

[15]Mark Harris. *Optimizing Parallel Reduction in CUDA*. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. 2007 (cit. on p. 22).

[16]Sibylle Hess and Katharina Morik. „C-SALT: Mining Class-Specific ALTerations in Boolean Matrix Factorization“. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski. Cham: Springer International Publishing, 2017, pp. 547–563 (cit. on p. 35).

[17]Robert Hochberg. *Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model*. http://www.shodor.org/media/content//petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf. Accessed: 2017-01-13. 2002 (cit. on p. 33).

[18]Patrik O. Hoyer. „Non-negative matrix factorization with sparseness constraints“. In: *CoRR* cs.LG/0408058 (2004) (cit. on p. 2).

[19]Christian Hundt, Andreas Hildebrandt, and Bertil Schmidt. „rapidGSEA: Speeding up gene set enrichment analysis on multi-core CPUs and CUDA-enabled GPUs“. In: *BMC Bioinformatics* 17.1 (Sept. 2016), p. 394 (cit. on p. 20).

[20]Jörg Wicker Stefan Kramer Katharina Dost Andrey Tyukin. „Scaling Up Boolean Matrix Decomposition: An Effective Parallelization Scheme“. Unpublished work recieved by university connections (cit. on pp. 8, 24, 35, 40).

[21]Hyunsoo Kim and Haesun Park. „Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis“. In: *Bioinformatics* 23.12 (2007), pp. 1495–1502. eprint: /oup/backfile/content_public/journal/bioinformatics/23/12/10.1093/bioinformatics/btm134/2/btm134.pdf (cit. on p. 2).

[22]Hyunsoo Kim and Haesun Park. „Sparse Non-negative Matrix Factorizations via Alternating Non-negativity-constrained Least Squares for Microarray Data Analysis“. In: *Bioinformatics* 23.12 (June 2007), pp. 1495–1502 (cit. on p. 2).

[23]R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. „An investigation of Unified Memory Access performance in CUDA“. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2014, pp. 1–6 (cit. on p. 14).

[24]Amy Nicole Langville, Carl Dean Meyer, Russell Albright, James Cox, and David Duling. „Algorithms, Initializations, and Convergence for the Nonnegative Matrix Factorization“. In: *CoRR* abs/1407.7299 (2014). arXiv: 1407.7299 (cit. on p. 8).

[25]Daniel Lee and H Sebastian Seung. „Learning the Parts of Objects by Non-Negative Matrix Factorization“. In: 401 (Nov. 1999), pp. 788–91 (cit. on pp. 2, 6).

[26]Daniel D. Lee and H. Sebastian Seung. „Algorithms for Non-negative Matrix Factorization“. In: *Advances in Neural Information Processing Systems 13*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, 2001, pp. 556–562 (cit. on pp. 2, 5, 6).

[27]*Matrix Factorization: A Simple Tutorial and Implementation in Python*. http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/. Accessed: 2017-12-26 (cit. on p. 6).

[28]Pauli Miettinen. *Boolean matrix factorizations*. https://people.mpi-inf.mpg.de/~pmiettin/slides/BooleanMatrixFactorizationsForDataMining_NRC_slides.pdf (cit. on pp. 2, 4).

[29]Pauli Miettinen. *Boolean matrix factorizations*. https://people.mpi-inf.mpg.de/~pmiettin/slides/BooleanMatrixFactorization_UMannheim2014.pdf (cit. on pp. 5–7).

[30]Pauli Miettinen. „Matrix Decomposition Methods for Data Mining : Computational Complexity and Algorithms“. In: (Jan. 2018) (cit. on p. 40).

[31]Pauli Miettinen. „The Boolean column and column-row matrix decompositions“. In: *Data Mining and Knowledge Discovery* 17.1 (Aug. 2008), pp. 39–56 (cit. on p. 2).

[32]Pauli Miettinen and Jilles Vreeken. „MDL4BMF: Minimum Description Length for Boolean Matrix Factorization“. In: *ACM Trans. Knowl. Discov. Data* 8.4 (Oct. 2014), 18:1–18:31 (cit. on p. 2).

[33]Pauli Miettinen and Jilles Vreeken. „Model Order Selection for Boolean Matrix Factorization“. In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '11. San Diego, California, USA: ACM, 2011, pp. 51–59 (cit. on p. 2).

[34]Pauli Miettinen, Taneli Mielikäinen, Aristides Gionis, Gautam Das, and Heikki Mannila. „The Discrete Basis Problem“. In: *Knowledge Discovery in Databases: PKDD 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 335–346 (cit. on p. 40).

[35]*MovieLens 10m data set*. https://grouplens.org/datasets/movielens/10m/. Accessed: 2017-01-21 (cit. on p. 35).

[36]*MovieLens 1m data set*. https://grouplens.org/datasets/movielens/1m/. Accessed: 2017-01-21 (cit. on p. 35).

[37]*MovieLens latest data set*. https://grouplens.org/datasets/movielens/latest/. Accessed: 2017-01-21 (cit. on p. 35).

[38]*NVIDIA GeForce GTX 1080*. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf. Accessed: 2017-01-02 (cit. on p. 17).

[39]*Nvidia GeForce GTX 1080 review - Pascal GPU Architecture*. http://www.guru3d.com/articles-pages/nvidia-geforce-gtx-1080-review,3.html. Accessed: 2017-01-02 (cit. on p. 12).

[40]*NVIDIA Visual Profiler*. https://developer.nvidia.com/nvidia-visual-profiler. Accessed: 2017-01-07 (cit. on p. 23).

[41]*Parabool with BMaD and Scavenger on Dropbox*. https://dropbox.com/sh/xm8nzeqckl7piff/AABJ-Id_8QzTHs98HnBsgMqha?dl=0. Accessed: 2017-01-30 (cit. on p. 40).

[42]NVIDIA Paulius Micikevicius. *Performance Optimization*. http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Fundamental_Optimizations.pdf. 2010 (cit. on p. 12).

[43]NVIDIA Paulius Micikevicius. *Performance Optimization*. http://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf. 2011 (cit. on pp. 13, 16, 23).

[44]Shaina Race. *Initializations for Nonnegative Matrix Factorization*. http://www4.ncsu.edu/~slrace/nmfinitializations.pdf. 2012 (cit. on p. 8).

[45]Y. Shitov. „A short proof that NMF is NP-hard“. In: *ArXiv e-prints* (May 2016). arXiv: 1605.04000 [math.CO] (cit. on p. 5).

[46]Václav Snásel, Jan Platos, and Pavel Krömer and Dusan Húsek and Roman Neruda and Alexander A. Frolov. „Investigating Boolean Matrix Factorization“. In: 2008 (cit. on p. 24).

[47]Andrey Tyukin, Stefan Kramer, and Jörg Wicker. „BMaD – A Boolean Matrix Decomposition Framework“. English. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo. Vol. 8726. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 481–484 (cit. on p. 40).

[48]S. A. Vavasis. „On the complexity of nonnegative matrix factorization“. In: *ArXiv e-prints* (Aug. 2007). arXiv: 0708.4149 [cs.NA] (cit. on p. 5).

[49]*What is GPU-accelerated computing?* http://www.nvidia.com/object/what-is-gpu-computing.html. Accessed: 2017-01-01 (cit. on p. 14).

[50]Z. Zhang, T. Li, C. Ding, and X. Zhang. „Binary Matrix Factorization with Applications“. In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. Oct. 2007, pp. 391–400 (cit. on p. 7).

[51]Z. Zhang, T. Li, C. Ding, and X. Zhang. „Binary Matrix Factorization with Applications“. In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. Oct. 2007, pp. 391–400 (cit. on p. 24).

List of Figures

List of Tables

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, 05.02.2018

Adrian Lamoth