cuTeBool: Fast and Scalable Boolean matrix factorization on GPUs using Tensor Cores

Andrea Beyer[1][0009−0001−0683−8803], Valentin Henkys[1][0009−0002−9379−7129], Robin Kobus[0000−0003−2726−1908], Stefan Kramer[1][0000−0003−0136−2540], and Bertil Schmidt[1][0000−0003−2597−8331]

Institute of Computer Science, Johannes Gutenberg University Mainz, Germany
{abeyer@students.uni-mainz.de, henkys@uni-mainz.de}

Abstract. Boolean matrix factorization aims to represent binary data as a product of two factor matrices, in order to uncover the underlying structure of the data and find a compressed representation. However, finding the factors of a given ground truth is computationally hard and calls for fast implementations that accomplish a good approximation in reasonable time. We present cuTeBool, a novel parallel algorithm that exploits Tensor Cores on CUDA-enabled GPUs for fast matrix operations based on a randomized approach. Our comprehensive performance evaluation shows that it produces approximate factorization competitive to other state-of-the-art tools within vastly reduced runtime for a variety of input matrices. Moreover, our algorithm is the only available method that scales well with the size of the ground truth and is able to factorize matrices that are at least one order-of-magnitude larger than all competitors. We further analyze algorithmic parameters allowing us to find a trade-off between performance and reconstruction quality.

Keywords: GPUs · Matrix Factorization · Parallel Computing

1 Introduction

Matrix factorization is an important technique in the field of unsupervised data mining and compression. It aims to find a decomposition of a ground truth matrix $\mathbf{C}$ into a product of n matrices $\mathbf{A_i}$

$$\mathbf{C} = \prod_{i=1}^{n} \mathbf{A_i}.$$

In most practical applications, $n \in \{2, 3\}$, like in *Singular Value Decomposition* with $n = 3$ and *Non-Negative Matrix Factorization* with $n = 2$.

Associative data can be naturally expressed in terms of Boolean matrices over $\mathcal{B}^{m\times n}$ with entries in $\mathcal{B} = \{0, 1\}$ instead of real-valued matrices. Rows and columns of such matrices are then interpreted as *objects* and *attributes*: an entry M_{ij} is set to 1, if object i has attribute j. Typical examples include whether a user listened to a song, read a book, or knows another person in a social network.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_18

Boolean matrices spawned by real-world applications are often sparse, i.e., the number of objects and attributes is large compared to the average number of attributes that are assigned to a single object.

In this context, **Boolean Matrix Factorization (BMF)** is defined as: Find $\mathbf{A} \in \mathcal{B}^{m \times k}$ and $\mathbf{B} \in \mathcal{B}^{k \times n}$ for a given ground truth $\mathbf{C} \in \mathcal{B}^{m \times n}$ such that

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B}.$$

Here, $\circ$ denotes matrix multiplication over the semi-ring $(\{0,1\}, \vee, \wedge)$. Since finding an exact representation is NP-hard [10,12], we often aim to find an approximation with $k \ll m$ and $k \ll n$. This results in a compressed representation of the ground truth and can be used for noise reduction, among others.

A lot of work has gone into optimizing BMF for non-parallel processors [2,9,19]. To improve associated runtimes some libraries started to use parallelism on multi-core CPUs [6] and many-core Graphics Processing Units (GPUs) [5,4]. With the rise of Machine Learning (ML), another type of compute unit has now emerged: the neural processing unit (NPU). It is optimized for Matrix Multiply-Accumulate (MMA) operations, which are frequently used in ML tasks. Our work focuses on NVIDIA's implementation, called Tensor Cores (TCs), available on most modern CUDA-enabled GPUs.

While typically designed for floating-point arithmetic, TCs now also feature support for Boolean matrix operations, making them a suitable candidate platform for BMF. In this work, we present cuTeBool – the first TC-enabled parallel BMF algorithm achieving high efficiency on modern GPUs.

Our detailed contributions consist of:

1. A novel BMF implementation utilizing TCs on GPUs, providing average speedups of $2\times$, $20\times$, $38\times$, $20\times$ compared to other limited-rank competitors cuBool, Panpal, Primp, MEBF, respectively, while maintaining better F_1-scores across all tested datasets.
2. Support for ground truth matrices that are at least one order-of-magnitude bigger than all competitors.
3. Automatic hyperparameter tuning, providing a good trade-off between reconstruction quality and performance.
4. Open-source implementation at https://gitlab.rlp.net/pararch/cutebool.

The remainder of the paper is structured as follows. We provide necessary background in Section 2, followed by Section 3 discussing related work. Algorithmic methods are introduced in Section 4. Parallelization details are described in Section 5. Performance is evaluated in Section 6. Finally, Section 7 concludes.

2 Background

2.1 Boolean Matrix Factorization

Formally, BMF operates within the Boolean semi-ring $(\{0,1\}, \vee, \wedge)$. Boolean Matrix Multiplication (BMM, denoted as $\circ$) is then defined as

$$C_{i,j} = \bigvee_{l=1}^{k} A_{i,l} \wedge B_{l,j}. \tag{1}$$

We denote the set of Boolean $x \times y$ matrices as $\mathcal{B}^{x \times y}$. BMF is defined as the task of finding two factor matrices $\mathbf{A} \in \mathcal{B}^{m \times k}$ and $\mathbf{B} \in \mathcal{B}^{k \times n}$ for a given ground truth $\mathbf{C} \in \mathcal{B}^{m \times n}$, such that

$$\mathbf{C} = (\mathbf{A} \circ \mathbf{B})\,. \tag{2}$$

We distinguish between the *decomposition rank*, which is the dimension k of a given factorization, and the *Boolean rank* of the ground truth: the smallest k, for which Eq. (2) can be satisfied.

Since finding exact solutions is NP-hard [10,12], low-rank approximations are typically preferred as a means to cancel out noise and compress data. The underlying notion is that the relations present in the data are influenced by a relatively small set of unknown factors. Hence, a factorization is commonly understood as a pair of basis vectors and coefficients that produce a close approximation and reveal structure of the underlying data.

In this work, we approach *approximate BMF* with a rank $k \leq 128$. The goal here is to find $\mathbf{A}$, $\mathbf{B}$ with a given k that minimize the *reconstruction error*:

$$E_{rec} = |\mathbf{C} - (\mathbf{A} \circ \mathbf{B})| = fp + fn. \tag{3}$$

fp and fn denote the number of false positives and false negatives, respectively. In some cases, false negatives and false positives may vary in significance for the reconstruction. This can be captured by the *weighted reconstruction error*:

$$E_{rec}^{w} = w \cdot fn + fp. \tag{4}$$

For $w > 1$, this implies that false negatives are penalized harder than false positives. We use *recall* (R), *precision* (P), and the *F-measure* (F_1) as metrics to assess the quality of approximations. They are expressed in terms of *true/false positives* (tp, fp) and *negatives* (tn, fn) as follows.

$$R = \frac{tp}{tp + fn}, P = \frac{tp}{tp + fp}, F_1 = 2\frac{P \cdot R}{P + R} = \frac{2tp}{2tp + fp + fn} \tag{5}$$

2.2 CUDA and Tensor Cores

CUDA-enabled GPUs feature multiple Streaming Multiprocessors (SMs) that can be programmed using a large number of threads. Threads are partitioned into *thread blocks* and *warps*. Each warp consists of 32 threads, each thread block in turn hosts up to 32 warps. Threads within the same thread block can access a common, but limited shared memory with reduced latency compared to global memory, which is accessible to all threads. Within a warp, threads benefit from data exchange comparable to register-speed through warp shuffles. The CUDA language extension for C++ distinguishes between host and device code. The host (CPU) can launch kernels on the device (GPU).

Working at warp-level, TCs are optimized to perform fast matrix multiplication. This is achieved through the MMA operation $\mathbf{W} = \mathbf{X} \cdot \mathbf{Y} + \mathbf{Z}$. $\mathbf{X}$ and $\mathbf{Y}$ are referred to as *factors*, $\mathbf{Z}$ and $\mathbf{W}$ as *accumulators*. Large matrices are partitioned into so-called *fragments* of fixed size, that can be understood as submatrices. Each warp can compute one fragment of the result by iterating over fragments in the factors, using the accumulator for subresults. This allows for efficient parallelization of arbitrary sized products.

Each thread only holds a subset of entries for each fragment, so that the warp collectively holds all entries of the considered submatrix. The choice of fragment dimensions and data type for the multiplication is restrained by the special layout required by TCs [14].

Newer TCs also provide support for 1-bit data types with a `m8n8k128` fragment type. That is, factor fragments of dimensions 128×8 and 8×128, and accumulator fragments of size 8×8 are supported. Since we aim for a low factorization rank, we do not extend the factors beyond 128 bit in their common dimension, so that the factorization rank either coincides with or is lower than the side length of a single fragment. Note that TCs do not directly perform BMM as defined in Eq. (1), but compute a sum instead of the logical OR:

$$w_{ij} = \texttt{POPC}\left(\mathbf{X}[i] \,\texttt{\&}\, \mathbf{Y}[j]\right) + z_{ij} = \sum_{l=1}^{k} (x_{il} \wedge y_{lj}) + z_{ij} \tag{6}$$

The obtained value can then be transformed using $w_{ij} \leftarrow w_{ij} \neq 0$ to acquire the desired result for OR-based BMM.

3 Related Work

In an early attempt, Miettinen et al. adapted the *Asso*-algorithm [9] to BMF by first generating a number of candidate vectors and greedily selecting k such vectors to form a factor that minimizes the reconstruction error. Dynamic BMF [8] provides an online algorithm that incorporates recent changes of the ground truth into an existing factorization. Moreover, Miettinen and Vreeken worked on the problem of the Minimum Description Length (MDL) for BMF [10]. Here, the error of BMF is encoded in a separate matrix $\mathbf{E}$, so that $(\mathbf{A} \circ \mathbf{B}) \oplus \mathbf{E}$ is an exact representation of the ground truth.

More recently, Wan et al. [19] presented median expansion for BMF (*MEBF*). This approach identifies patterns by permuting the rows and columns of the original matrix, such that positive entries accumulate in the top right. The best rank-1 approximation of the resulting matrix is then added to the factors of the ground truth.

Belohlavek and Vychodil [2] connected BMF to formal concept analysis by proposing the *GreCon* and *GreConD* algorithms. Both algorithms greedily add candidate rows to the factors and pursue a from-below approach, i.e., no false positives are allowed. GreConD differs from GreCon by only considering a subset of formal concepts which increases performance but yields comparable results. These algorithms have since been extensively studied and optimized. Trnecka and Vyjidacek revisited GreCon and presented *GreCon2* [17] – speeding up the algorithm without a loss in quality. In [1], *GreConD+* is introduced, which deviates from the from-below approach taken by GreConD. Here, accepted factors are expanded, allowing false positives if the overall reconstruction error benefits from the expansion. Trnecka and Krajča developed *ParaGreConD* [6], a performance optimized version of the GreConD algorithm, by using multiple CPU-threads to evaluate candidates. Note, that their implementation approaches BMF as a coverage problem in the sense that it extends the factors until a desired coverage factor is reached. This crucially differs from our approach, which tries to minimize the reconstruction error for a fixed factorization rank. As ParaGreConD is the only GreCon modification that explores parallelization, it is most relevant to the work presented in this paper.

Similarly, Outrata and Trnecka [15] note that many previous BMF algorithms work with greedy heuristics, sequentially improving the result. They introduce a general parallelization scheme for such algorithms, that computes multiple locally optimal subresults in parallel, and returns the best found approximation. Here, the main benefit is the improved quality of the factorization, compared to single-core execution.

Hess et al. implemented the *PALTiling* framework [4], featuring the algorithms *Panpal* and *Primp*. Both algorithms are iterative in nature, but use NVIDIA's cuBLAS library to benefit from GPU-acceleration for individual operations. PALTiling attempts to decompose the ground truth into densely populated tiles, allowing factors with values in $[0, 1]$ for computation, and enforcing a binary result by applying a threshold to the individual factor values. The main difference between the algorithms is the use of different cost measures. Panpal uses a L_1 regularization, following the example of Panda [7], whilst Primp adapts the cost function of Krimp [18], originally used in the context of MDL.

Also recognizing the inherent capability for massive parallelization of matrix multiplication, *cuBool* [5] harnesses GPUs to compute a factorization. Here, an initial factorization is guessed and iteratively improved by evaluating random bit-flips, until an error threshold or time limit is reached. Multiple warps of threads are used to explore different updates in parallel. Our work presented in this paper builds upon the basic idea of cuBool [5], but extends it to gain even higher speeds by making it amenable to TCs.

4 Algorithmic Method

Our overall algorithmic approach is based on local search: initially, random factor matrices are generated and iteratively improved. Within each iteration, we update factors **A** and **B** successively, aiming to minimize the reconstruction error. The algorithm terminates, when it was unable to improve the reconstruction for a certain number of iterations, indicating a local minimum.

Updates are generated by introducing minor changes to factor values by flipping randomly selected bits. Subsequently, the resulting weighted reconstruction error is calculated. Updates that improve the error are written to memory.

When generating updates, single bit-flips are preferred over multiple flips within the same row. This has two main causes. First, flipping many bits at once makes it impossible to determine which of the individual flips are beneficial, as only the update as a whole can be kept or discarded. This potentially keeps non-beneficial bit-flips or discards beneficial ones. Second, performing many bit-flips tends to overpopulate the factors, leading to dense results and low precision.

Since updates in one factor may influence the error calculation for updates in the other factor, each factor has to be updated successively. As the error calculation of different values in the same factor can be done independently, cuTeBool can harness GPUs to compute multiple updates on the same factor in parallel. Our novel parallelization scheme using TCs is described in detail in Sections 5.2 and 5.3.

A common issue with the randomized approach is that the algorithm may produce an empty reconstruction. Especially on sparse datasets the initial reconstruction will contain a relatively high number of false positives. This incentivizes discarding true positives along with false positives to reduce the reconstruction error. As a result, the algorithm may produce a zero reconstruction with high probability, losing all information about the data.

To avoid this problem, we apply a weight w to the reconstruction error (see Eq. (4)) that penalizes false negatives more than false positives in an attempt to achieve a higher recall. I.e., a true positive is only discarded if at least w false positives are discarded along with it. This weight is gradually decreased over many iterations. At a weight of 1, false negatives and false positives are penalized equally. Initial weight and reduction factor can be customized as command line arguments. A higher weight typically favors higher recall at the expense of precision. During execution, we monitor the (weighted) improvement achieved by each update to detect local minima. We terminate, when it got stuck for a certain number of iterations without finding an update, that would improve the error. Alternatively, it can also be set to run for a fixed amount of iterations. Due to the randomized nature of this approach, the algorithm may get stuck in a sub-optimal minimum. To increase the chance of finding a good solution, we suggest to spawn multiple instances with different seeds, as discussed in Section 5.

5 Parallelization Scheme

The goal of our implementation scheme is to minimize CPU-GPU communication and let the GPU handle most of the computation. Therefore, after reading the input data, the ground truth is moved to the GPU and the factors are initialized directly on the GPU. Subsequently, our main optimization loop starts. In this loop we iteratively update the factors by a few random bits, check the new errors and decide whether to keep the new bits or not. Since each update of a factor has to be performed independently from the other factor, we use two distinct kernels: One for column and one for row updates, where each has to wait for the previous update to be finished.

We support the initialization of multiple instances to be computed at once, so that available GPU resources can be fully utilized. Each instance differs in the random seed used to initialize the factors and to determine the bits flipped in each iteration. This variety leads to possibly faster convergence and better results than exploring a single seed. Only when the break condition is reached, we choose the instance with the lowest error and move the computed factors back to the CPU for possible storage. This overall workflow is shown in Algorithm 1.

5.1 GPU Data Layout

While our factor and ground truth matrices are read frequently, they generally do not fit into the fast CUDA shared memory. Consequently, we use an efficient data layout for global memory to allow for efficient coalesced memory accesses as proposed by cuBool [5], seen in Fig. 1.

Our focus is on a decomposition rank of 128 or less, due to the perfect fit into the Tensor Cores' $8 \times 8 \times 128$ fragment dimensions for 1-bit data types. To allow the threads to perform coalesced data accesses, we store our factors in a row-major fashion for $\mathbf{A}$ and column-wise for $\mathbf{B}$, respectively, using the 4×32-bit wide integer type `uint4`. For a decomposition rank lower than 128, we pad the matrices with zeros to reach the fragment size.

For the ground truth matrix another storage layout is needed, since it has to be read row- and column-wise, depending on which error we want to compute. We compute rows and columns in batches of 32 to better accommodate the required layout for TCs. Thus, we divide the ground truth into multiple 32×32 submatrices. If necessary, the ground truth's dimensions are padded with zeros to a multiple of 32. Each submatrix can then be read in a coalesced way.

5.2 Update Kernels

While there are two distinct kernels for row and column updates, both work analogously. Therefore, we explain the column kernel as example. A general overview over its workflow is shown in Algorithm 2.

In the column kernel, the selected random bits of the factor $\mathbf{B}$ are changed and the resulting error is calculated. If the changes reduce the error, they are kept, else discarded. Since these steps are the most time consuming part of the program, a lot of care has been taken to make them as efficient as possible.

Each thread block is assigned a batch of 32 consecutive columns of factor $\mathbf{B}$. These 32 columns are further sub-divided into 4 groups of 8 columns corresponding to the width of the chosen TC fragments. Each batch is processed by

Algorithm 1 cuTeBool workflow

```
 1: for i=0,.. instances-1 do in parallel
 2:     A[i], B[i] = random_init(seed)
 3:     while not stuck do
 4:         for j=0,.. break do
 5:             update(A, weight, new_seed)
 6:             update(B, weight, new_seed)
 7:         end for
 8:         update_parameters(factor, weight)
 9:     end while
10:     errors[i] = final_error(A[i], B[i])
11: end parallel for
12: best = arg $\min_i$(errors[i])
13: return A[best], B[best]
```

Alg. 1: Most work is done on the GPU.

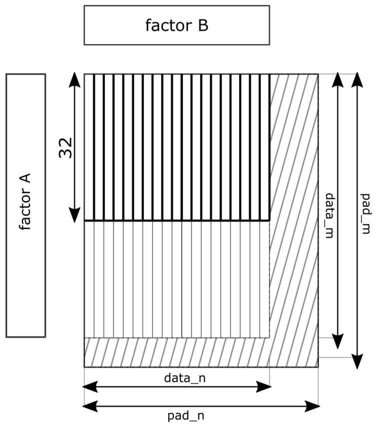

Fig. 1: Data layout of the ground truth.

one GPU thread block, which we spawn with 32 warps. Thus 8 warps cooperate on the same columns in a warp group. The values of **A** are needed by multiple threads. Hence, we can benefit from coalesced reads and the lower latency of shared memory, by splitting **A** into chunks of 1024 rows, that are read in a coalesced way and temporarily stored in shared memory. This warp cooperation scheme is illustrated in Fig. 2. Each of the warp groups update random bits of their respective columns and then calculate the old and updated errors, splitting the rows of **A** equally across the collaborating warps.

5.3 Error Computation on Tensor Cores

In our implementation, we address TCs using Parallel Thread Execution (PTX) instructions, which are used as an intermediate code representation for the NVIDIA CUDA compiler (NVCC). PTX allows for direct control over the registers participating in operations on Tensor Cores giving a transparent data layout. This has the benefit, that data does not need to be written to and read from memory between individual TC computations.

To decide if our updated columns lead to an improvement, we need to compute the original error and the updated (weighted) error. Here, each warp group is responsible for 8 columns of the ground truth matrix. Using this distribution, each thread loads its respective column of **B** and its updated version into registers and iteratively works through the rows of the current chunk of **A**. For each thread, the **B** values remain constant throughout kernel execution, removing the need to read **B** multiple times. Once all rows of the current chunk of **A** are exhausted, the block collectively loads the next chunk into shared memory and proceeds iterating over **A**.

Within each 8×8 tile of the result matrix, each thread of one warp is assigned two horizontally adjacent elements. This corresponds to a transposed submatrix, compared to how the ground truth is stored. To avoid extra memory accesses or computational overhead, we pass the input for each BMMA operation in a

Algorithm 2 Column Error Kernel

```
1:  error = 0
2:  b = to_shared()
3:  for chunk = 0,.. num_chunks-1 do
4:      sync()
5:      frags = to_shared(A, Truth, chunk)
6:      sync()
7:      for frag in frags do
8:          a, t = get_matrix_frags(frag)
9:          result = tensor_core_bmma(b, a)
10:         error += weighted_error(result, t)
11:     end for
12: end for
13: total_error = accumulate_warp_errors()
```

Alg. 2: Note that the algorithm for computing the row-wise errors only differs in memory access and data exchange patterns.

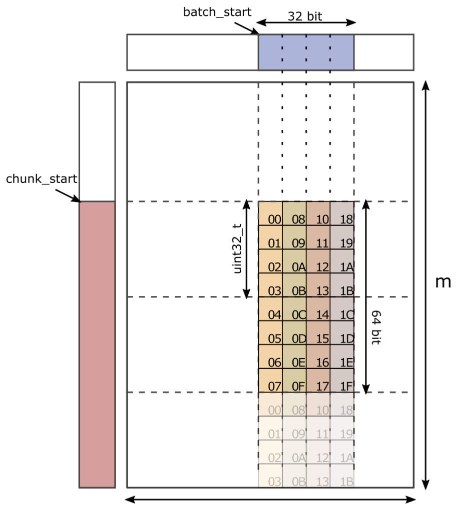

Fig. 2: Warp distribution: each group of 8 consecutive warps work together on an 8-bit wide column. Warps are numbered in hexadecimal format.

reversed order, which will produce the transposed result $\mathbf{B}^T\mathbf{A}^T = (\mathbf{AB})^T$. After each MMA, the threads compare their results to the ground truth and update the reconstruction error accordingly.

Afterwards all partial errors for each column are accumulated. This process is performed in three steps:

1. Gather the partial errors from collaborating threads within each warp using warp-shuffles.
2. Employ shared memory to exchange these column-specific errors with cooperating warps.
3. Execute another warp shuffle to accumulate the errors that have been received in the previous step.

These steps have to be performed both for the updated errors, as well as the original one. If the total updated error for a column is lower than the previous version, the change is kept and written to memory.

5.4 Automatic Hyperparameter Selection

Our program depends on a number of hyperparameters influencing cuTeBool's behavior and the quality of results. We automatically select the best parameters for a good trade-off between reconstruction quality and efficiency, based on a grid search on multiple datasets. The outcome on one representative smaller dataset, namely AOffice (see Table 1), is illustrated in Fig. 3. While the starting weight, shown in the left most plot, is dataset dependent, there is a general trade-off

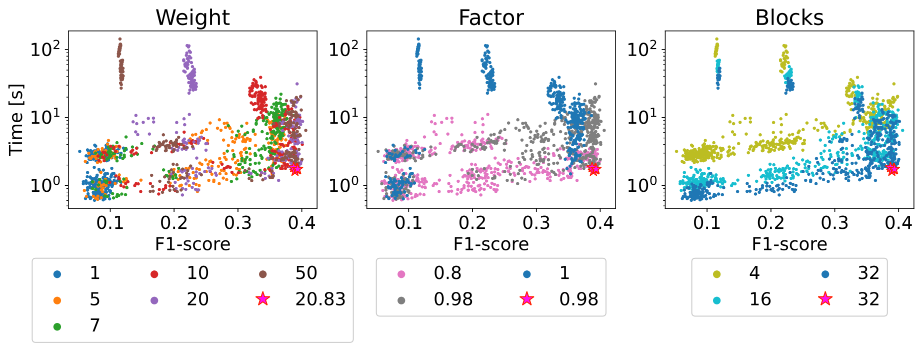

Fig. 3: Effect of hyperparameters on runtime (y-axis) and F_1-score (x-axis) of the AOffice dataset. Bottom right is better. Each point represents one run. Automatically set parameters are indicated as a star.

noticeable between reconstruction quality and runtime. Additionally, we noticed that, if the weight is high, the algorithm tends to find more true positives on sparser datasets early on. Thus, we implemented the following measure, clamping the results to avoid extremes:

$$\text{weight} = \max\left\{3,\ \min\left\{\frac{\#\text{entries in ground truth}}{\min\{m,n\}},\ 50\right\}\right\}$$

To alleviate the advantages of both high and low weights, we introduce a factor reducing the weight each iteration. Thus, we can profit from a higher starting weight, finding a lot of true positives early on, while reducing the runtime in the long run. As seen in the center plot in Fig. 3, reducing the weight to fast results in worse results. Thus, we always choose a factor of 0.98.

On large, very sparse datasets we find that the effect of a high, non-constant starting weight is diluted by the high number of iterations needed. Thus we introduce a minimum weight for datasets with a ground truth density of $< 1\%$ and size of > 3 GB. As seen in Fig. 4 by an example of the Goodreads Comics dataset (see Table 1), setting this parameter directly correlates to a trade-off in precision vs. recall. Since the density has the greatest impact on the achievable recall, we use it to determine the minimal weight:

$$\text{min\_weight} = \max\left\{3,\ \min\left\{\text{density} \cdot 1000,\ 10\right\}\right\}$$

As final parameter we take a look at the number of GPU thread blocks used, which impacts the number of factors updated each iteration. This directly correlates to how much the GPU is utilized by a single run of our algorithm. We notice that, while more blocks yield to lower runtimes, the difference between 16 and 32 blocks is less noticeable than from 4 to 16. We exploit this fact by launching four instances of our algorithm at once, each using a quarter of the GPU's SMs. Each instance is provided with a different seed, resulting in different convergence times, but often with similar F_1-scores. When the first instance finds a suitable result, the program stops and the best result is returned.

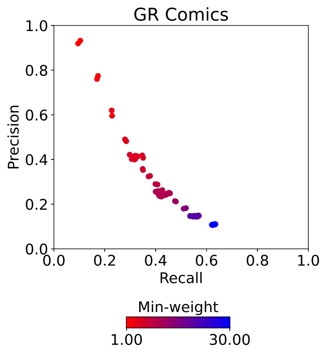

Fig. 4: The minimum weight steers the trade-off between recall and precision: a higher value boosts recall at the expense of precision.

6 Experimental Results

We evaluate our implementation by comparing the performance of cuTeBool to different state-of-the-art BMF tools. To gain a comprehensive understanding of the performance of the algorithms under different settings, the tools are tested on real-world datasets varying in densities and sizes, as displayed in Table 1.

The performance of our proposed algorithm is investigated by running cuTeBool with a rank of 128 and automatically chosen hyperparameters as described in Section 5.4. We compare against a set of different publicly available implementations, differentiating between two types of tools:

`t1`: Tools, that solve approximate BMF by means of a limited rank: cuBool [5], Primp [4], Panpal [4], and MEBF [19]. cuBool was run with a rank of 32, which is the highest supported rank. The others were run with a maximum rank of 128, matching the maximum rank imposed by the GPUs TCs.

`t2`: One tool providing no option to limit the rank, possibly generating factorizations of much higher reconstruction rank: ParaGreConD [6].

In cuBool [5] different parameter settings are tested; we ran cuBool for each of these, but only report the best of their results for each dataset. To account for the random nature of both cuBool and cuTeBool, we performed five full runs with different seeds and 4 instances each. We report the average of these 5 runs. We note, that the quality metrics vary in less than 1% between these runs, i.e., the randomization mostly affects the runtime of the algorithm.

Primp and Panpal are both part of the PALTiling framework [4] and were both run with at most $50,000$ iterations.

Moreover, we consider different CPU-based tools. MEBF [19] was run with a threshold parameter of 0.1, and a default coverage factor of 0.9. We further examined the performance of the CPU-parallel implementation of the GreConD algorithm (ParaGreConD [6]), which does not limit the reconstruction rank.

Table 1: Datasets used for benchmarking BMF tools

	#Rows	#Columns	#Ones	Density in %	Notes
20News [11]	11,269	61,188	10,817	0.0016	(binarized) bag-of-words for 20Newsgroups articles
AMusic [13]	5,541	3,568	58,905	0.2979	
AOffice [13]	4,905	2,420	50,402	0.4246	Amazon user reviews
ATools [13]	16,638	10,217	124,371	0.0732	
GR Comics [20]	342,415	89,411	7,347,630	0.0240	Goodreads user reviews by book genre
GR Fantasy [20]	726,932	258,585	55,397,550	0.0295	
Movie25M [3]	162,541	56,887	25,000,095	0.2704	MovieLens movie ratings
StackOverflow (2 GB) [16]	131,072	131,072	3,954,912	0.0230	User interactions on stackoverflow
StackOverflow (8 GB) [16]	262,144	262,144	6,671,042	0.0097	

The CPU-based tools (MEBF, ParaGreConD) were tested on an AMD Ryzen Threadripper 3990X CPU using 64 threads, while the GPU-based ones (cuTeBool, cuBool, Primp, Panpal) were run on an NVIDIA H100 SXM GPU. Each tool was given up to 5 hours to factorize each dataset.

The results of our benchmark are shown in Fig. 5. CuTeBool is the only tool that is able to report results on all dataset sizes, with most other tools not being able to process datasets of 1 GB or larger in time without throwing an error.

On smaller datasets, cuTeBool stands out with high speedups and F_1-scores compared to all competitors in `t1`. Generally, discovering new true positives is harder for cuTeBool than removing false ones, resulting in high precision accompanied by moderate recall values. This is partially counter-acted by the minimal weight: on datasets where the weight is not reduced to one, recall and precision tend to balance out more evenly.

Compared to cuBool, we report an increase in both precision and recall, in turn leading to a higher F_1-score; all while reaching a lower runtime, with an average speedup of $2\times$ for datasets that both tools processed. We attribute the improvement in reconstruction quality to the greater reconstruction rank enforced by the usage of TCs and to our tuned hyperparameter choices. While being able to process bigger datasets than the other competitors, cuBool started failing at datasets of size 8 GB. In contrast, cuTeBool is only limited by the amount of GPU memory and runtime. For the largest dataset, we stopped cuTeBool at the time limit of 5 hours, but since cuTeBool is able to save results of stopped runs, we can still report its results here.

For large-scale datasets we notice a decline in reconstruction quality for cuTeBool, especially in precision. We attribute this to two main factors. Firstly, large dimensions of the ground truth mean that each individual entry in the factors is updated less often. This increases the runtime, but also makes good updates harder to find. Moreover, large datasets tend to have a much higher Boolean rank and are inherently harder to factorize and compress to a rank of 128. Both showing the limits of the Tensor Core based maximum rank of 128. The CPU-based tool MEBF is the only tool in `t1` with comparable F_1-scores to cuBool, but is only able to fully process the 4 smallest datasets. While cuTeBool is faster on almost all datasets, with speedups ranging from $8.5\times$ up to $56.7\times$, MEBF is able to beat cuTeBool in the 20News dataset by a factor of 1.16, resulting in an average speedup of $20\times$. But in terms of F_1-scores and precision cuTeBool wins across the board, with MEBF achieving higher recall in 3 of the 4 datasets.

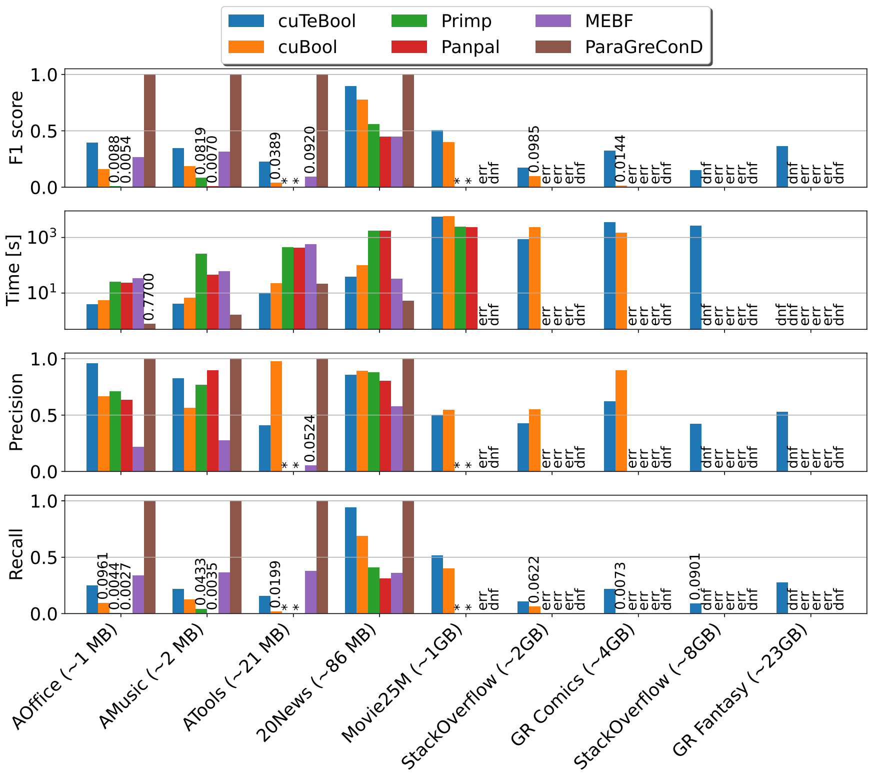

Fig. 5: cuTeBool against competitors. When a library threw an error (*err*), did not finish (*dnf*), or produced a 0 matrix (*) this is shown instead of a bar.

Primp and Panpal rarely produced good factorizations and – despite also exploiting GPU parallelization – lag behind cuTeBool's runtime. On 2 out of 5 datasets they processed, they only produced 0 matrices. Across the 3 datasets in which they provided results, cuTeBool achieves an average speedup of 38× and 20× compared to Primp and Panpal, respectively.

The only library surpassing cuTeBool in terms of speed and quality for the 4 smallest datasets is ParaGreConD in `t2`. It is important to note here that ParaGreConD always tries to find the perfect factorization, even at the cost of a much higher reconstruction rank – often orders of magnitude higher. The reconstruction ranks ParaGreConD produced in our tests are shown in Table 2. Since the rank often blows up the size of the factor matrices to dimensions higher than the ground truth, this library is not usable for compression. In our benchmarks, ParaGreConD only achieved compression for the 20News dataset. But, in contrast to cuTeBool, it fails to calculate a factorization for Movie25M in time, and errors out for all larger datasets. Also, while it is faster in 3 cases, cuTeBool is significantly faster on ATools, resulting in a similar average runtime. Therefore, ParaGreConD is only usable for relatively small datasets of less than 1 GB in size and only if compression is not of interest.

Table 2: Compression ratios compared to the ground truth size. Higher value corresponds to higher compression. In our experiments cuTeBool always used a rank $k = 128$.

		AOffice	AMusic	ATools	20News	Movie25M	GR Fantasy
Rank	ParaGreConD	2472	3958	10664	389	dnf	dnf
Compr. Ratio	ParaGreConD	0.66	0.55	0.59	24.46	dnf	dnf
	cuTeBool	12.66	16.96	49.45	74.35	329.22	1490.13

7 Conclusion

We have presented cuTeBool, a novel freely available parallel open-source BMF framework for modern GPUs using Tensor Cores. It outperforms limited rank competitors in speed, reaching average speedups of 2×, 20×, 38×, 20× compared to cuBool, Panpal, Primp, MEBF, respectively, while achieving a better reconstruction quality. In addition, cuTeBool is the only available method that can scale to large matrix sizes > 1 GB. ParaGreConD [6] is only able to achieve better reconstruction quality for small matrices at the expense of using unlimited ranks leading to poor compression factors and very limited scalability (i.e. it is unable to process larger matrices exceeding 1 GB in size within 5 hours).

Besides the performance benefits, we introduced automatic hyperparameter selection, allowing the tool to reach a good trade-off between reconstruction quality and performance. We showed that our tool is able to find good reconstructions in a reasonable amount of time on very big datasets, only limited by the GPU's total memory. For larger datasets, cuTeBool is the only tool able to perform the BMF, without throwing errors. For future work it would be interesting to investigate the usage of a flexible reconstruction rank beyond 128. While this may lead to extra computational costs and a lower compression ratio, it could result in a better reconstruction quality for large datasets of high Boolean rank.

Acknowledgments. The research in this paper is partly funded by Carl Zeiss Foundation, grant number P2021-02-014 (TOPML project).

References

1. Belohlavek, R., Trnecka, M.: A new algorithm for boolean matrix factorization which admits overcovering. Discrete Applied Mathematics **249**, 36–52 (2018). https://doi.org/10.1016/j.dam.2017.12.044
2. Belohlavek, R., Vychodil, V.: Discovery of optimal factors in binary data via a novel method of matrix decomposition. JCCS **76**(1), 3–20 (2010). https://doi.org/10.1016/j.jcss.2009.05.002
3. Harper, F.M., Konstan, J.A.: The movielens datasets: History and context. ACM TiiS **5**(4) (12 2015). https://doi.org/10.1145/2827872

4. Hess, S., Morik, K., Piatkowski, N.: The PRIMPING routine - tiling through proximal alternating linearized minimization. Data Min. Knowl. Discov. **31**(4), 1090–1131 (2017). https://doi.org/10.1007/S10618-017-0508-Z
5. Kobus, R., Lamoth, A., Muller, A., Hundt, C., Kramer, S., Schmidt, B.: cuBool: Bit-parallel boolean matrix factorization on cuda-enabled accelerators. In: ICPADS 2018. pp. 465–472 (2018). https://doi.org/10.1109/PADSW.2018.8644574
6. Krajča, P., Trnecka, M.: Parallelization of the grecond algorithm for boolean matrix factorization. In: Formal Concept Analysis. pp. 208–222. Springer International Publishing, Cham (05 2019). https://doi.org/10.1007/978-3-030-21462-3_14
7. Lucchese, C., Orlando, S., Perego, R.: Mining top-k patterns from binary datasets in presence of noise. pp. 165–176 (04 2010). https://doi.org/10.1137/1.9781611972801.15
8. Miettinen, P.: Dynamic boolean matrix factorizations. In: ICDM 2012. pp. 519–528 (2012). https://doi.org/10.1109/ICDM.2012.118
9. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The discrete basis problem. IEEE Transactions on Knowledge and Data Engineering **20**(10), 1348–1362 (2008). https://doi.org/10.1109/TKDE.2008.53
10. Miettinen, P., Vreeken, J.: Mdl4bmf: Minimum description length for boolean matrix factorization. ACM Trans. Knowl. Discov. Data **8**(4) (10 2014). https://doi.org/10.1145/2601437
11. Mitchell, T.: Twenty Newsgroups. UCI Machine Learning Repository (1999). https://doi.org/10.24432/C5C323
12. Nau, D.S., Markowsky, G., Woodbury, M.A., Bernard Amos, D.: A mathematical analysis of human leukocyte antigen serology. Mathematical Biosciences **40**(3), 243–270 (1978). https://doi.org/10.1016/0025-5564(78)90088-3
13. Ni, J., Li, J., McAuley, J.: Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In: 2019 EMNLP-IJCNLP. pp. 188–197. ACL, Hong Kong, China (Nov 2019). https://doi.org/10.18653/v1/D19-1018
14. NVIDIA Corporation: CUDA C++ Toolkit Documentation, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma, accessed 18 Oct 2024
15. Outrata, J., Trnecka, M.: Parallel exploration of partial solutions in boolean matrix factorization. Journal of Parallel and Distributed Computing **123**, 180–191 (2019). https://doi.org/https://doi.org/10.1016/j.jpdc.2018.09.014
16. Paranjape, A., Benson, A.R., Leskovec, J.: Motifs in temporal networks. In: WSDM '17. p. 601–610. ACM, NY, USA (2017). https://doi.org/10.1145/3018661.3018731
17. Trnecka, M., Vyjidacek, R.: Revisiting the grecon algorithm for boolean matrix factorization. Knowledge-Based Systems **249**, 108895 (2022). https://doi.org/https://doi.org/10.1016/j.knosys.2022.108895
18. Vreeken, J., Leeuwen, M., Siebes, A.: Krimp: Mining itemsets that compress. Data Min. Knowl. Discov. **23**, 169–214 (07 2011). https://doi.org/10.1007/s10618-010-0202-x
19. Wan, C., Chang, W., Zhao, T., Li, M., Cao, S., Zhang, C.: Fast and efficient boolean matrix factorization by geometric segmentation. Proc. of the AAAI Conf. on AI **34**(04), 6086–6093 (Apr 2020). https://doi.org/10.1609/aaai.v34i04.6072
20. Wan, M., McAuley, J.: Item recommendation on monotonic behavior chains. In: RecSys '18. p. 86–94. ACM, NY, USA (2018). https://doi.org/10.1145/3240323.3240369