Eurographics Symposium on Parallel Graphics and Visualization (2009)
J. Comba, K. Debattista, and D. Weiskopf (Editors)

Parallelized Matrix Factorization for fast BTF Compression

Roland Ruiters, Martin Rump and Reinhard Klein

Institute for Computer Science II, University of Bonn †

Abstract

Dimensionality reduction methods like Principal Component Analysis (PCA) have become commonplace for the compression of large datasets in computer graphics. One important application is the compression of Bidirectional Texture Functions (BTF). However, the use of such techniques has still many limitations that arise from the large size of the input data which results in impractically high compression times. In this paper, we address these shortcomings and present a method which allows for efficient parallelized computation of the PCA of a large BTF matrix. The matrix is first split into several blocks for which the PCA can be performed independently and thus in parallel. We scale the single subproblems in such a way, that they can be solved in-core using the EM-PCA algorithm. This allows us to perform the calculation on current GPUs exploiting their massive parallel computing power. The eigenspaces determined for the individual blocks are then merged to obtain the PCA of the whole dataset. This way nearly arbitrarily sized matrices can be processed considerably faster than by serial algorithms. Thus, BTFs with much higher spatial and angular resolution can be compressed in reasonable time.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture G.4 [Mathematics of Computing]: Mathematical Software—Parallel and vector implementations

1. Introduction

One important challenge in photorealistic rendering is the representation of the reflectance properties of materials. Nowadays reflectance measurements provide a method to faithfully reproduce the appearance of materials. One kind of material representation is the Bidirectional Texture Function (BTF) introduced by Dana et al. [DvGNK97]. It is able to cover a wide range of materials with complex mesostructures and spatially varying reflectance properties. To deal with the large amounts of input data, several compression methods have been proposed by now. Many of them rely on dimensionality reduction methods like Principal Component Analysis (PCA). However, a serious drawback of these methods is that they are computationally very expensive because they require the factorization of large matrices. Another general issue is the huge data size which makes out-of-core algorithms necessary. These are severly encumbered by IO bottlenecks and thus get nearly unusable for large BTF datasets even if their in-core equivalents would suffice the needs.

Because of the comparatively long measurement times and the low resolution of older BTF acquisition setups, the compression times of these techniques were acceptable so far. However, the availability of high-resolution and low-cost digital cameras has made the development of highly parallel BTF acquisition devices possible. Modern devices are able to capture a material sample with high dynamic range, an angular resolution of more than 100 view and light directions and a spatial resolution of several megapixels in a few hours. This corresponds to data sizes of several hundreds of gigabytes. Current techniques do not scale well to these large datasets. For example, with non-parallelized methods the compression of a 512x512x95x95 BTF using an out-of-core PCA on the full BTF matrix takes about 13 hours which is in stark contrast to a few hours of measurement, severely hampering the practical operation of a BTF acquisition setup. Thus, the compression of high-resolution BTFs is still a challenging problem of high practical relevance.

† e-mail: {ruiters, rump, rk}@cs.uni-bonn.de

DOI: 10.2312/EGPGV/EGPGV09/025-032

In this paper, we propose a method for efficient and parallelizable factorization of large matrices and show its application to BTF compression. The core operations of the algorithm are performed on graphics hardware exploiting the massive parallel computing power of modern GPUs. The algorithm is designed to use only existing libraries for matrix operations and is thus very easy to implement. We achieve speed gains of up to a factor of 35 compared to implementations on a single CPU core.

Our basic idea is to subdivide the large BTF data matrix into several smaller blocks that can be processed in-core and then to use eigenspace merging to obtain the factorization of the complete matrix. We use the EM-PCA algorithm of S. Roweis [Row98] for the factorization of the small blocks. The runtime of this iterative algorithm is primarily dominated by matrix operations in its inner loop. By performing most of these operations on the GPU, we are able to gain a massive speed increase for the factorization of the individual matrix blocks.

The rest of the paper is organized as follows: In Section 2 we give an overview about previous work on matrix factorization and BTF compression. In Section 3 we introduce necessary theoretical background on our method. The implementation issues can be found in Section 4. The paper is concluded with result in Section 5 and a discussion in Section 6.

2. Previous Work

In this section we will given an overview on BTF compression and matrix factorization methods.

2.1. BTF compression

The Bidirectional Texture Function was introduced by Dana et al. [DvGNK97]. It represents the appearance of complex materials as a six-dimensional function $\rho(x, \omega_i, \omega_o)$ of surface position x, light direction ω_i and view direction ω_o.

Most of previous BTF compression techniques rely on matrix or tensor factorization, clustering, wavelets or the fitting of analytical models to the texels of the BTF. The most general approach of these are the factorization techniques, which work for a broader range of material classes since they are not based on fixed basis functions.

Several BTF compression techniques based on tensor factorization have been proposed by now, e.g. Vasilescu et al. [VT04], Wang et al. [WWS*05] and Wu et al. [WXC*08]. Since the BTF is a six-dimensional function, a tensor is its canonical representation. However, compression techniques which are based on tensor factorization have several drawbacks when compared to PCA-based representations of equal quality, as was reported by Müller in [Mül08]. On the one hand, the compression times are quite long with only small or no increase in compression ratio. On the other hand, the reconstruction speed is very slow if only one element of the tensor is to be reconstructed, which is the standard case in BTF rendering.

In contrast to this high dimensional representation, Principal Component Analysis (PCA) works only on two-dimensional matrices. Thus, compression methods based on PCA have to represent the six-dimensional dataset as matrices. There exist two main approaches for this. The first is to split the BTF data into several parts, for which PCA is performed totally independently and the other one is to unroll the higher dimensional data. Several combinations of the two methods exist, some of which achieve good compression ratios and at the same time offer fast decompression speeds.

Sattler et al. [SSK03] grouped all images for one view direction and then compressed them independently of each other. Müller et al. [MMK03] used local PCA for BTF compression by employing spatial clustering and applying PCA on each cluster. Suykens et al. [SBLD03] applied a technique called chained matrix factorization to BTF matrices. They factorize the data matrix repeatedly using a different parametrization each time.

Factorization of the whole BTF data as one matrix was used by Koudelka et al. [KM03] and Liu et al. [LHZ*04]. The main problem here is the sheer size of the matrix and the resulting processing times needed to factorize this matrix. Therefore, only BTFs with quite low resolution could be processed so far. Nevertheless, the main advantage of a full-matrix factorization is the possible exploitation of correlations throughout the full data set, which is not possible in between totally independently processed parts. Thus, the compression ratio of a full-matrix-factorization approach is superior to all other matrix-factorization methods, that were proposed to increase compression speed and make the matrices that small that they can be factorized in-core. With our method the good compression ratio becomes available even for high-resolution BTFs.

2.2. Matrix factorization

Compression can be performed by first applying Principal Component Analysis and then truncating after the first k principal components. For most BTF materials, about 100 components are sufficient for very faithful reproduction. The naive approach is to subtract the mean from the $m \times n$ data matrix $\mathbf{M}$ to form the mean-centered matrix $\overline{\mathbf{M}}$ and then to perform a singular value decomposition (SVD) $\overline{\mathbf{M}} = \mathbf{USV}^T$ by calculating the eigenvectors and eigenvalues of the covariance matrix $\overline{\mathbf{MM}}^T$. However, this is not the best approach with regard to numerical precision and there exist algorithms that directly compute the SVD from $\mathbf{M}$ for all singular vectors and values at the same time (see e.g. [GL96]). Unfortunately, they require $O\left(mn^2 + m^2n + n^3\right)$ time and

are furthermore not well suited for out-of-core implementations.

Since for compression purposes only the first k eigenvalues and eigenvectors are needed, performing a full factorization is not very efficient. To overcome this problem, several techniques have been proposed by now, which allow to calculate only the k largest eigenvalues and corresponding eigenvectors in considerably smaller time and which are also better suited to out-of-core implementation. In the incremental SVD algorithm from Brand [Bra02] the data is processed column-wise by updating the eigenspace as new columns are added. Its time complexity is $O(mnk)$. Roweis [Row98] proposed an iterative expectation-maximization (EM) algorithm for PCA which also has a time complexity of $O(mnk)$. A different approach has been taken by Liu et al. [LWWT07]. They subdivided the data matrix into several blocks, performed a traditional PCA on each block and then merged the eigenspaces of the single blocks with the method of Skarbek [Ska03] to obtain the eigenspace of the whole matrix.

Block-wise processing is also the basic idea behind our approach, as it allows to parallelize the computation of the single blocks. Additionally, the blocks can be choosen in such a way that they fit into the memory of a GPU and can therefore be processed in-core. We decided to use the EM-PCA algorithm from Roweis for these subproblems because it can process one block of data at once with only a few matrix operations. In contrast to the incremental SVD method where each column must be added successively, this allows for easy GPU acceleration.

3. Theory

Given a BTF $\rho(x, \omega_i, \omega_o)$ as a six-dimensional table with a RGB-triple in every entry, we can define a BTF data matrix $\mathbf{M}_{BTF}$ by unrolling the color channels c and the directions in one dimension as well as the spatial position x in the other one by defining indexing operators i and j. We end up with the $m \times n$ matrix $\mathbf{M}_{BTF}(i(\omega_i, \omega_o, c), j(x)) = \rho(x, \omega_i, \omega_o)[c]$ with m as the number of light and view direction combinations times the number of color channels and n as the number of texels.

Given such a $m \times n$ BTF matrix $\mathbf{M}_{BTF}$, its PCA can be calculated by first determining the mean $\mathbf{m}$ of $\mathbf{M}_{BTF}$ and then performing a singular value decomposition (SVD) of the matrix $\mathbf{M}$ obtained by subtracting this mean from $\mathbf{M}_{BTF}$. The full SVD of $\mathbf{M}$ is a decomposition $\mathbf{M} = \mathbf{U}_f \mathbf{S}_f \mathbf{V}_f^T$, with orthogonal matrices $\mathbf{U}_f$ and $\mathbf{V}_f$ and a diagonal matrix $\mathbf{S}_f$ containing the singular values sorted in descending order. Here, $\mathbf{U}_f$ is a $m \times m$ and $\mathbf{V}_f$ is a $n \times n$ matrix, but for BTF compression this representation is truncated, by only keeping the first k columns of $\mathbf{U}_f$ and $\mathbf{V}_f$ corresponding to the first k largest singular values of $\mathbf{S}_f$. In the following, we will thus only consider the $m \times k$ matrix $\mathbf{U}$ the $k \times k$ matrix $\mathbf{S}$ and the $n \times k$ matrix $\mathbf{V}$ obtained after this truncation.

For our algorithm, the matrix $\mathbf{M}$ is first subdivided into N blocks $\mathbf{M} = [\mathbf{M}_1 \ \cdots \ \mathbf{M}_N]$ of the respective sizes $m \times s_i$. On each of these blocks, a SVD is performed independently resulting in the matrices $\mathbf{U}_i, \mathbf{S}_i, \mathbf{V}_i$. This step thus can be easily be performed in parallel. The SVDs for these blocks are then merged to finally obtain the decomposition of the complete matrix $\mathbf{M}$. See Figure 1 for an illustration of this process. In our implementation, we merge the matrices successively. Instead, the merging could also be performed in a binary tree, as suggested by Liu et al. in [LWWT07]. As the results in their paper show, tree structured merging does reduce the error, but only by a small amount (below 1% in all examples given there). On the other hand, when using tree structured merging, it is necessary to store more intermediate results, increasing the memory requirements.

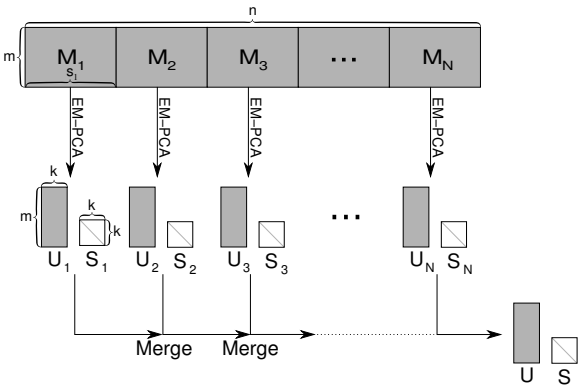

Figure 1: *Illustration of the algorithm. The matrix* $\mathbf{M}$ *is first divided into blocks* $\mathbf{M}_1, \cdots, \mathbf{M}_N$. *For each of the blocks, an independent SVD is calculated via the EM-PCA algorithm to obtain* $\mathbf{U}_i$ *and* $\mathbf{S}_i$. *Then, the individual decompositions are merged to obtain the final result* $\mathbf{U}$ *and* $\mathbf{S}$.

Since the matrix $\mathbf{V}$ contains the projections of $\mathbf{M}$ into the $\mathbf{U}$-space, parts of the data vectors orthogonal to this space are not represented. In each merge step, however, the subspace spanned by the matrix $\mathbf{U}$ changes. Thus, if the vectors in $\mathbf{V}$ are reprojected into this new $\mathbf{U}$-space, only the part in the intersection of the old and the new space can be represented and the orthogonal part is lost. This would lead to an accumulation of errors during the merge steps. To avoid this accumulation, we first compute only $\mathbf{U}$. Instead of calculating and merging the individual matrices $\mathbf{V}_i$, we calculate $\mathbf{V}$ in an additional step by projecting the columns of $\mathbf{M}$ on the subspace spanned by $\mathbf{U}$. For the same reason, we also recalculate the singular values in the final projection step, even though we have to update $\mathbf{S}$ during the calculation of $\mathbf{U}$ since it is needed to perform the merge steps. In addition to the improved accuracy, this reduces the complexity of the implementation as well as the memory requirements. The drawback of this approach is that we must spend additional IO time for this final step since we have to load the full matrix again. Thus, for applications where speed is more important than precision, it might be advantegeous to instead update $\mathbf{V}$ together with $\mathbf{U}$ during the merge steps.

In the following sections, we will give a short overview of the EM-PCA algorithm we used for the factorization of the sub-problems and the technique we used to merge the individual factorizations to obtain the full SVD.

3.1. EM-PCA

Instead of calculating a SVD of $\mathbf{M}_i$ directly, we approximate it by first using the EM-PCA algorithm introduced in [Row98] to find the subspace spanned by the first k principal components of $\mathbf{M}_i$ and then performing the SVD on the projection of $\mathbf{M}_i$ into this subspace. This way, we only have to compute the SVD for a $(k+1) \times s_i$ matrix, containing the data vectors projected into the subspace spanned by the first k principal components and the mean direction of $\mathbf{M}$. This factorization can be done very fast for small k.

The EM-PCA algorithm is an expectation-maximization algorithm which allows to find the subspace spanned by the first k principal components, without explicitly calculating all principal components. After initializing the $m \times k$ output matrix $\mathbf{C}$ with random values, it iterates between the following two steps:

E-step:

$$\mathbf{X} = (\mathbf{C}^T\mathbf{C})^{-1}\mathbf{C}^T\overline{\mathbf{M}}_i$$

M-step:

$$\mathbf{C}^{new} = \overline{\mathbf{M}}_i\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}$$

Here, $\overline{\mathbf{M}}_i$ is a $m \times s_i$ input matrix with zero mean and $\mathbf{X}$ is a $k \times s_i$ matrix of unknown states. After the iteration has completed, the columns of $\mathbf{C}$ span the principal subspace. As analyzed in [Row98], this EM algorithm always converges and the number n_i of necessary iterations is rather small and independent from the size of $\overline{\mathbf{M}}_i$. In our experience, 15 iterations were sufficient for good compression results.

Thus, only matrix multiplications and inversions are needed for the calculation of the PCA. In practice, the runtime is dominated by the two multiplications with the $m \times s_i$ matrix $\overline{\mathbf{M}}_i$, since these require $O(kms_i)$ operations. The algorithm is therefore practically linear in the size of the input data. This approach is thus very well suited for GPU implementations, as the matrix multiplications are easily parallelizable, especially for very large matrices like $\overline{\mathbf{M}}_i$. Matrix inversions have a runtime which is cubic in the matrix dimension and are furthermore not easily parallelizable. However, the contribution of the two inversions in this algorithm to the total runtime is negligible for small k, because the matrices $\mathbf{C}^T\mathbf{C}$ and $\mathbf{X}\mathbf{X}^T$ are both only of size $k \times k$.

When using EM-PCA to calculate the principle subspace for the matrices $\mathbf{M}_i$, it is important to keep in mind that, even though the mean has been subtracted from the full matrix $\mathbf{M}$, the individual block matrices can have non-zero mean and are therefore not directly suited as input data for the EM-PCA. We thus calculate the mean $\mathbf{m}_i$ for each block matrix independently and subtract it from $\mathbf{M}_i$ to obtain the matrix $\overline{\mathbf{M}}_i$ which is then used for the calculation of the subspace. Afterwards, we add the mean vector as an additional column to the matrix $\mathbf{C}$, obtaining the matrix $\mathbf{C}_m$. This is necessary, since the component of the mean vector, which is orthogonal to the determined subspace would otherwise be lost when projecting the data points into the space spanned by $\mathbf{C}$ and thus neglected during the following SVD.

The actual SVD calculation is then performed on a projection of $\mathbf{M}_i$ into the subspace spanned by $\mathbf{C}_m$. For this, $\mathbf{C}_m$ is first orthogonalized, obtaining the matrix $\mathbf{C}_o$. The projection can then be calculated as $\mathbf{P} = \mathbf{C}_o^T\mathbf{M_i}$. Since the columns of $\mathbf{P}$ contain only $k+1$ entries, the SVD $\mathbf{U}_P\mathbf{S}_P\mathbf{V}_P^T = \mathbf{P}$ can be calculated efficiently. To obtain the final result, we project the matrix $\mathbf{U}_P$ back into the original space by setting $U_i = \mathbf{C}_o\mathbf{U}_P$ and $\mathbf{S}_i = \mathbf{S}_P$.

3.2. SVD Merging

Let $\mathbf{U}_1\mathbf{S}_1\mathbf{V}_1^T$ and $\mathbf{U}_2\mathbf{S}_2\mathbf{V}_2^T$ be two singular value decompositions which have been truncated after c_1 and c_2 singular values respectively and let $\widetilde{\mathbf{M}}_1 = \mathbf{U}_1\mathbf{S}_1\mathbf{V}_1^T$ and $\widetilde{\mathbf{M}}_2 = \mathbf{U}_2\mathbf{S}_2\mathbf{V}_2^T$ be the matrices reconstructed from these. We have to find the singular value decomposition $\mathbf{U}\mathbf{S}\mathbf{V}^T$ of the composed matrix $\widetilde{\mathbf{M}} = [\widetilde{\mathbf{M}}_1\ \widetilde{\mathbf{M}}_2]$. For this, we generalized and adapted the update step of the incremental SVD [Bra02] to the merging of the two SVDs. Similar eigenspace merging techniques like the one in [HMM00] could however be used instead.

The merging of the two SVDs is performed by first constructing an orthogonal space for the subspace spanned by both $\mathbf{U}_1$ and $\mathbf{U}_2$ and then performing the SVD within this subspace.

For this, the singular value decomposition of $\widetilde{\mathbf{M}}_2$ is split into the part which lies within the subspace spanned by $\mathbf{U}_1$ and the part orthogonal to this subspace. First, $\mathbf{U}_2$ is projected into this space, resulting in $\mathbf{L} = \mathbf{U}_1^T\mathbf{U}_2$. Then, the orthogonal part is computed as $\mathbf{H} = \mathbf{U}_2 - \mathbf{U}_1\mathbf{L}$. In the next step, an orthogonal basis $\mathbf{Q}$ for the space spanned by $\mathbf{H}$ is determined. Now, $\mathbf{H}$ is projected into this space by setting $\mathbf{R} = \mathbf{Q}^T\mathbf{H}$. $\mathbf{U}' = [\mathbf{U}_1\ \mathbf{Q}]$ is thus an orthogonal basis for the subspace spanned by both $\mathbf{U}_1$ and $\mathbf{U}_2$.

We can now consider the following identity:

$$\widetilde{\mathbf{M}} = \mathbf{U}'\mathbf{U}'^T\widetilde{\mathbf{M}} \tag{1}$$

$$= \begin{bmatrix}\mathbf{U_1} & \mathbf{Q}\end{bmatrix} \begin{bmatrix}\mathbf{U_1}^T \\ \mathbf{Q}^T\end{bmatrix} \begin{bmatrix}\mathbf{U}_1\mathbf{S}_1\mathbf{V}_1^T & \mathbf{U}_2\mathbf{S}_2\mathbf{V}_2^T\end{bmatrix} \tag{2}$$

$$= \begin{bmatrix}\mathbf{U_1} & \mathbf{Q}\end{bmatrix} \begin{bmatrix}\mathbf{U}_1^T\mathbf{U}_1\mathbf{S}_1 & \mathbf{U}_1^T\mathbf{U}_2\mathbf{S}_2 \\ \mathbf{Q}^T\mathbf{U}_1\mathbf{S}_1 & \mathbf{Q}^T\mathbf{U}_2\mathbf{S}_2\end{bmatrix} \begin{bmatrix}\mathbf{V}_1 & 0 \\ 0 & \mathbf{V}_2\end{bmatrix}^T \tag{3}$$

$$= \underbrace{\begin{bmatrix}\mathbf{U_1} & \mathbf{Q}\end{bmatrix}}_{\mathbf{U}'} \underbrace{\begin{bmatrix}\mathbf{S}_1 & \mathbf{L}\mathbf{S}_2 \\ 0 & \mathbf{R}\mathbf{S}_2\end{bmatrix}}_{\mathbf{C}} \underbrace{\begin{bmatrix}\mathbf{V}_1 & 0 \\ 0 & \mathbf{V}_2\end{bmatrix}^T}_{\mathbf{V}'^T} \tag{4}$$

In this identity, (4) is already of similar structure as a

SVD of $\widetilde{\mathbf{M}}$ because $\mathbf{U}'$ and $\mathbf{V}'$ are orthogonal matrices. However, $\mathbf{C}$ is not a diagonal matrix. Therefore, we have to perform a singular value decomposition $\mathbf{U}''\mathbf{S}''\mathbf{V}''^T = \mathbf{C}$ which is computationally not very expensive in our case since $\mathbf{C}$ is a $(c_1+c_2)\times(c_1+c_2)$ matrix and $c_1, c_2 \ll m, n$.

Since $\mathbf{U}', \mathbf{U}'', \mathbf{V}', \mathbf{V}''$ are orthogonal matrices, $\mathbf{U} = \mathbf{U}'\mathbf{U}''$, $\mathbf{S} = \mathbf{S}'$ and $\mathbf{V} = \mathbf{V}'\mathbf{V}''$ is the singular value decomposition of $\widetilde{\mathbf{M}}$:

$$\widetilde{\mathbf{M}} = \mathbf{U}'\mathbf{C}\mathbf{V}'^T = \mathbf{U}'\mathbf{U}''\mathbf{S}\mathbf{V}''^T\mathbf{V}'^T \quad (5)$$

$$= \mathbf{U}\mathbf{S}\mathbf{V}^T \quad (6)$$

The calculation of the matrix $\mathbf{U}$ is thus possible from only $\mathbf{U}_1, \mathbf{S}_1$ and $\mathbf{U}_2, \mathbf{S}_2$. Therefore, we do not need to calculate and update $\mathbf{V}$ during the calculation of $\mathbf{U}$, but can neglect it first and then obtain $\mathbf{V}$ afterwards by projecting the data on the basis $\mathbf{U}$.

After the merge step, the new SVD has $c_1 + c_2$ singular values and vectors. Since we always merge a matrix with $k+1$ columns to the already computed result this would grow by $k+1$ in each merge step. Therefore, it is necessary to truncate after each merge step. To reduce the error introduced by this truncation, we simply keep $2k$ singular values and vectors instead of only k during the calculation. In our experiments, this was sufficient for good BTF compression results. However, an approach to further reduce the error would be to instead use a threshold on the singular values to decide where to truncate the decomposition, as done in [Bra02]. We avoid this, because the decomposition would continue to grow during the merge operations, though to a lesser degree.

4. Implementation

As can be seen in Pseudocode 1, our algorithm is based on just a few basic matrix operations, most of which can be easily parallelized on the GPU. For this, we use the NVIDIA CUBLAS™library (for more information see [NVI08]), which allows to perform many basic linear algebra operations efficiently on the GPU. For our algorithm, the most important of these are the matrix multiplications, for which we use the `cublasSgemm` function. Similarly, we also calculate the column means, using a matrix-vector product, and the mean subtraction, using the rank-1 matrix update function `cublasSger`, with the CUBLAS™library, though none of these operations has a very high contribution to the total runtime. We also accelerated the matrix orthogonalization on the GPU.

Thus, the only parts of the algorithm not accelerated on the GPU are operations on the tiny matrices of size $k \times k$ and $(3k+1)\times(3k+1)$ respectively. For BTF compression, k is choosen quite small and therefore these operations are

Function: BlockSVD($\mathbf{M}, \mathbf{m}, n_i, k$)

```
// Subtract mean
m_l := mean (M)
M := add-to-columns (M, −m_l)
// EM-PCA
C := [k random unit column vectors]
foreach i ∈ {1,...,n_i} do
    X := (C^T C)^{-1} C^T M
    C := M X^T (X X^T)^{-1}
end
// Perform SVD in subspace
m_d := m_l − m
C_o := orthogonalize ([C m_d])
M := add-to-columns (M, m_d)
U,S,V = svd (C_o^T M)
return C_o U, S
```

Function: MergeBlocks($\mathbf{U}_1, \mathbf{S}_1, \mathbf{U}_2, \mathbf{S}_2, k$)

```
// Find orthogonal subspace for U_2
L := U_1^T U_2
H := U_2 − U_1 L
Q := orthogonalize (H)
R := Q^T H
// Merge the SVDs
C := [ S_1   L S_2 ]
     [ 0     R S_2 ]
U'',S'',V'' := svd ( C )
U := [U_1 Q] U''
return U_{1:m,1:2k}, S''_{1:2k,1:2k}
```

Function: BlockwisePCA($\mathbf{M}, k, n_i$)

```
m := mean (M)
// Calculate U
foreach i ∈ {1,...,N} do
    M_i = LoadBlock (i)
    U',S' = BlockSVD (M_i, m, n_i, k)
    if i = 1 then
        U := U'
        S := S'
    else
        U,S = MergeBlocks (U,S,U',S',k)
    end
end
// Project M into subspace
// to get S and V
V := (U^T M)^T
S := Diag(ColumnNorms(V))
V := V S^{-1}
return U,S,V,m
```

Pseudocode 1: *Our factorization method*

mostly irrelevant for the total runtime. Thus, an GPU implementation of these parts is not necessary, reducing the implementation complexity considerable, since CPU implementations of these algorithms are readily available, for example in the LAPACK library [ABD*90].

The size of the invidual matrices $\mathbf{M}_i$ should be chosen as large as the available GPU memory allows, because each merge step introduces a certain error and we should strife to minimize the number of merge steps. Since the matrix is processed blockwise, the runtime can easily be further improved by performing the IO asynchronously to the actual calculation. For this, we use an additional thread which preloads the next block of the matrix while the current one is processed.

Using this technique, we can on the one hand directly perform a factorization of the full BTF data matrices. However, we also applied our algorithm to the LocalPCA BTF-compression algorithm of Müller et al. [MMK03]. This algorithm first performs a clustering step in the spatial dimension and then applies the PCA to each cluster independently. The advantage of this method is, that a very low number of components is sufficient to faithfully reproduce the data in the single clusters. Therefore, the decompression speed is considerably higher than for techniques based on a full matrix factorization. We use our method to accelerate both the clustering and the final projection steps of this algorithm. Furthermore, we perform the clustering within the projection of $\mathbf{M}$ into the $\mathbf{U}$-space by first performing a factorization of the full BTF matrix, as this further increases the performance by reducing the time needed for the error calculations.

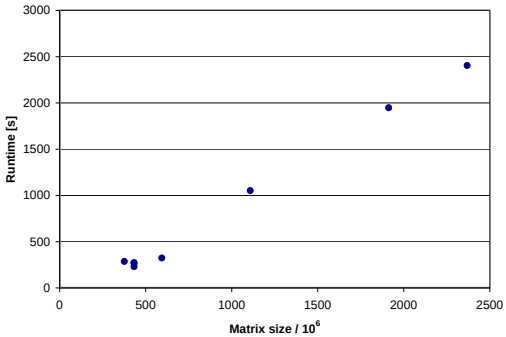

Figure 2: *Runtime of our algorithm for different matrix sizes and $k = 120$ components. Runtime for the smaller matrices is heavily influenced by the caching behaviour of the operating system.*

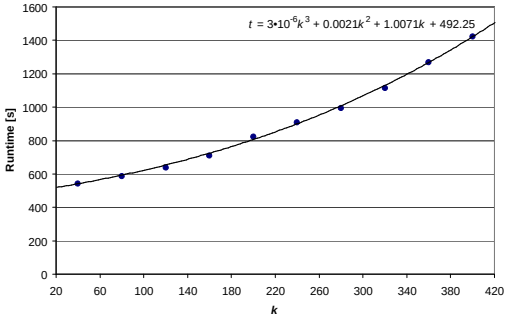

Figure 3: *Runtime of our algorithm for increasing number of components k.*

5. Results

To show the advantages of our parallelized factorization method in the context of BTF compression, we applied it to full BTF matrices of several materials. For the reconstruction, we used the first 120 principal components. We compare our runtimes and reconstruction errors to an out-of-core implementation of the EM-PCA algorithm performed on a single CPU core. For this, we simply used the average ABRDF RMSE:

$$E = \frac{1}{n}\sum_{i=1}^{n}\sqrt{\frac{\left\|\mathbf{M}_i - \tilde{\mathbf{M}}_i\right\|^2}{m}} \qquad (7)$$

Here, $\mathbf{M}_i$ is the i-th column of the BTF matrix and $\tilde{\mathbf{M}}_i$ is the i-th column of the reconstructed matrix. Table 1 shows timings and the achieved reconstruction errors. All timings were measured on a computer with Q6600 CPU, 8GB of main memory and a GeForce 8800 GTX GPU with 768MB GPU memory. Additionally we compared our extension of the LocalPCA compression method of Müller et al. [MMK03] to a CPU implemenation of the LocalPCA algorithm using the full data matrix and the EM-PCA method.

Our method achieves roughly a speed increase by a factor between 20 and 35. At the same time the increase in reconstruction error does not exceed 0.11% for the full-matrix-factorization. The relative error is more unstable for the LocalPCA, but this is mainly due to the jitter of the clustering step. In Figure 4 and 6 we compare renderings of the materials compressed with both full-matrix techniques and in Figure 5 we make the same comparsion for the two LocalPCA implementations. As it can be noticed, there is no visible difference between the version compressed with our techniques and the serial CPU implementations.

It should be noted, that the runtimes in Table 1 include the necessary IO times, which dominate the runtime of our algorithm for large datasets since caching by the operating system is no longer possible for them. For example for the large dataset Leather1 with 26 GB matrix size one complete IO pass required about 700 seconds. Thus, already more than half of the 2398 seconds runtime is spent on read operations during the mean calculation and the final computation to determine $\mathbf{V}$. The block factorizations have small additional IO cost, because we perform the IO asynchronously. Only the IO for the first block is performed synchronously, on the 26 GB matrix it takes about 40 seconds. Except for the second block, where still 10 additional IO seconds are needed,

Material	Resolution	Size [GB]	#Blocks	Block-PCA		EM-PCA		Speedup	Rel. error increase
				Time[s]	$\overline{RMSE}$	Time[s]	$\overline{RMSE}$		
Leather1	256^2x95^2	6.61	12	317	0.0236689	11659	0.0236529	36.78	0.068%
Leather1	512^2x95^2	26.44	48	2398	0.078524	47019	0.0785006	19.61	0.030%
Leather2	128^2x151^2	4.17	11	280	0.0113223	7711	0.0113162	27.54	0.054%
Leather3	256^2x81^2	4.81	9	261	0.0143584	8109	0.0143554	31.07	0.021%
Pulli	256^2x81^2	4.81	9	266	0.0282213	8129	0.0282085	30.56	0.045%
Fabric	256^2x81^2	4.81	9	223	0.0060206	8146	0.0060141	36.53	0.108%
				LPCA with Block-PCA		LPCA with EM-PCA			
				Time[s]	$\overline{RMSE}$	Time[s]	$\overline{RMSE}$		
Leather1	256^2x95^2	6.61	12	858	0.0368971	19104	0.0370494	22.27	-0.41%
Pulli	256^2x81^2	4.81	9	546	0.0374273	13573	0.0373398	24.86	0.23%

Table 1: *Upper part: Runtime and reconstruction error comparison between our method and a non-parallel EM-PCA with $k = 120$. Lower part: Comparison between our modified LocalPCA method and LocalPCA based on EM-PCA.*

the IO for all further blocks is completely asynchronous and only one second is required after the calculation step to fetch the data for the next block.

We investigated the runtime of our algorithm with increasing matrix size $m \times n$ and increasing number of components k. As it can be seen in Figure 2, the runtime is linear in $m \times n$ as expected. Figure 3 shows the runtime in dependence on k. We performed cubic regression to determine the contribution of the $O\left(k^3\right)$ operations to the total runtime. The coefficients for the quadratic and cubic part are very low compared to the linear part. This shows that the matrix multiplications with the large matrices dominate the total runtime of the algorithm as it was stated in Section 3.

6. Conclusion

We presented a method which accelerates the factorization of large data matrices, as they can be found in BTF compression, by exploiting the massive parallel computing power offered by modern GPUs.

This is achieved by first subdividing the input matrix into blocks, which are factorized independently using the EM-PCA algorithm, and then merging the resulting eigenspaces to obtain the final result. This technique allows to process matrices of nearly arbitrary size. We evaluated our technique by applying it to the compression of full BTF matrices. Here, it achieves speedups between 20-35, without increasing the reconstruction error by more than 0.11%, when compared to a out-of-core CPU implementation of the EM-PCA algorithm. This considerable accereleration enables the practical processing of BTF datasets with high angular and spatial resolution.

The computation time for each block is not dependent on the contained data and the individual blocks can be processed independently and in arbitrary order. Therefore, we think it will be quite easy to parallelize the algorithm to multiple GPUs, because the load balancing between the execution threads should not be too complex. However, at the moment this would require to implement the basic linear algebra functions in CUDA since CUBLAS does not support the execution on multiple GPUs at the moment.

Though our technique has been developed for the compression of BTFs, the factorization of big matrices is a problem that arises in a large number of applications and thus its use in other areas might be worth further investigation in the future. Therefore, the source code is available at http://cg.cs.uni-bonn.de.

7. Acknowledgments

This work was supported by the German Science Foundation (DFG) under research grant KL 1142/4-1.

References

[ABD*90] ANDERSON E., BAI Z., DONGARRA J., GREENBAUM A., MCKENNEY A., CROZ J. D., HAMMERLING S., DEMMEL J., BISCHOF C., SORENSEN D.: LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing* (Los Alamitos, CA, USA, 1990), IEEE Computer Society Press, pp. 2–11. http://www.netlib.org/lapack/.

[Bra02] BRAND M.: Incremental singular value decomposition of uncertain data with missing values. In *Computer Vision - ECCV 2002, 7th European Conference on Computer Vision, Copenhagen, Denmark, May 28-31, 2002, Proceedings, Part I* (2002), Heyden A., Sparr G., Nielsen M., Johansen P., (Eds.), vol. 2350 of *Lecture Notes in Computer Science*, Springer, pp. 707–720.

[DvGNK97] DANA K. J., VAN GINNEKEN B., NAYAR S. K., KOENDERINK J. J.: Reflectance and texture of real-world surfaces. In *IEEE Conference on Computer Vision and Pattern Recognition* (1997), pp. 151–157.

[GL96] GOLUB G. H., LOAN C. F. V.: *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[HMM00] HALL P., MARSHALL D., MARTIN R.: Merging and splitting eigenspace models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 22*, 9 (Sep 2000), 1042–1049.

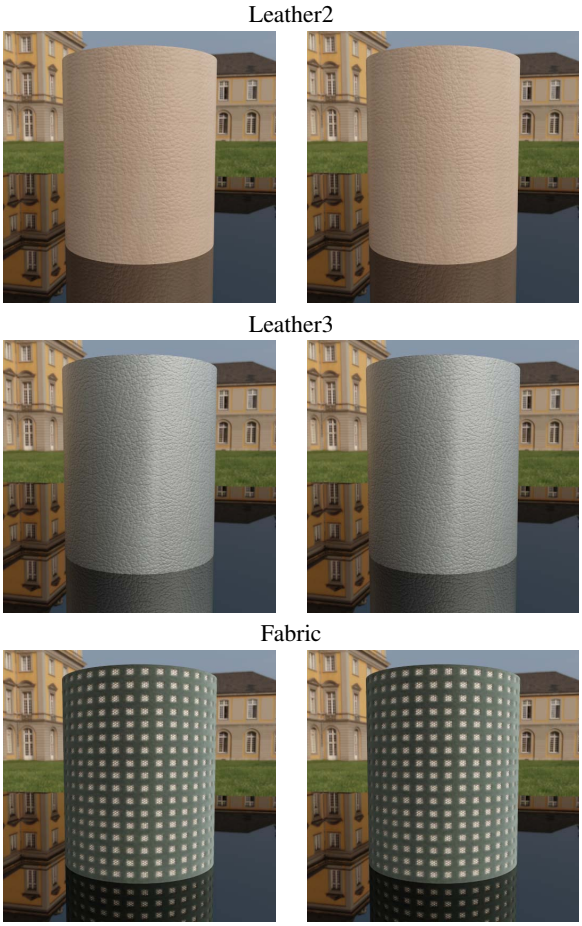

Figure 4: *Visual comparison between our factorization method (left row) and out-of-core EM-PCA (right row)*

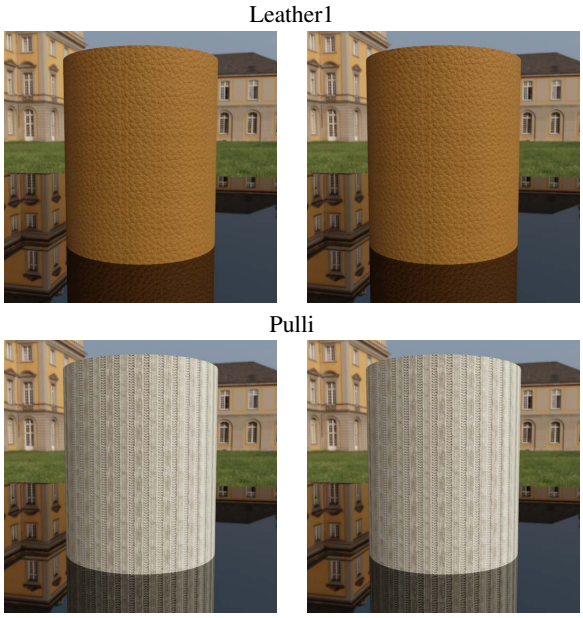

Figure 5: *Visual comparison between our modification of the LocalPCA method (left) and the original algorithm (right).*

[KM03] KOUDELKA M. L., MAGDA S.: Acquisition, compression, and synthesis of bidirectional texture functions. In *In Proc. 3rd Int. Workshop on Texture Analysis and Synthesis (Oct* (2003), pp. 59–64.

[LHZ*04] LIU X., HU Y., ZHANG J., TONG X., GUO B., SHUM H.-Y.: Synthesis and rendering of bidirectional texture functions on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics 10*, 3 (2004), 278–289.

[LWWT07] LIU L., WANG Y., WANG Q., TAN T.: Fast principal component analysis using eigenspace merging. In *ICIP (6)* (2007), pp. 457–460.

[Mül08] MÜLLER G.: Data-driven methods for compression and editing of spatially varying appearance, 2008. PhD thesis.

[MMK03] MÜLLER G., MESETH J., KLEIN R.: Compression and real-time rendering of measured btfs using local pca. In *Vision, Modeling and Visualisation 2003* (November 2003), Ertl T., Girod B., Greiner G., Niemann H., Seidel H.-P., Steinbach E., Westermann R., (Eds.), Akademische Verlagsgesellschaft Aka GmbH, Berlin, pp. 271–280.

[NVI08] NVIDIA: CUDA CUBLAS library, reference manual v2.0, March 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf.

[Row98] ROWEIS S.: Em algorithms for pca and spca. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10* (Cambridge, MA, USA, 1998), MIT Press, pp. 626–632.

[SBLD03] SUYKENS F., BERGE K. V., LAGAE A., DUTRÉ P.: Interactive rendering with bidirectional texture functions. *Computer Graphics Forum 22* (2003), 463–472.

[Ska03] SKARBEK W.: Merging subspace models for face recognition. In *CAIP* (2003), pp. 606–613.

[SSK03] SATTLER M., SARLETTE R., KLEIN R.: Efficient and realistic visualization of cloth. *Proceedings of the Eurographics Symposium on Rendering 2003* (2003).

[VT04] VASILESCU M. A. O., TERZOPOULOS D.: Tensortextures: multilinear image-based rendering. *ACM Trans. Graph. 23*, 3 (2004), 336–342.

[WWS*05] WANG H., WU Q., SHI L., YU Y., AHUJA N.: Out-of-core tensor approximation of multi-dimensional matrices of visual data. *ACM Trans. Graph. 24*, 3 (2005), 527–535.

[WXC*08] WU Q., XIA T., CHEN C., LIN H.-Y. S., WANG H., YU Y.: Hierarchical tensor approximation of multi-dimensional visual data. *IEEE Transactions on Visualization and Computer Graphics 14*, 1 (2008), 186–199.