JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 1383-1403 (2018)
DOI: 10.6688/JISE.201811_34(6).0003

Cache-Aware Out-of-Core Tensor Decomposition on GPUs

YU-TING TSAI, WEI-JHIH WANG AND TZU-YUAN KAO
Department of Computer Science and Engineering
Yuan Ze University
Taoyuan, 320 Taiwan
E-mail: hieicis91@hotmail.com; {plok00125; s815l7za}@gmail.com

For compressing large-scale multidimensional datasets, out-of-core tensor decomposition often consumes a lot of time. This article particularly presents a method based on two key ideas to improve its performance. First, cache-aware static scheduling schemes are employed to reduce the total number of disk accesses. Second, we take advantage of the massively parallel computing power and large memory size of modern GPUs to accelerate linear algebra operations of tensor decomposition. Our experiments demonstrate that the proposed method can achieve speedups of 11~16 over a naive implementation and 2.5~5.3 over previous work [43] for practical data-driven rendering applications.

Keywords: data-driven photorealistic rendering, multidimensional data analysis, tensor decomposition, out-of-core computation, general-purpose GPU computing

1. INTRODUCTION

Recently, large-scale multidimensional datasets, such as *bidirectional texture functions* (BTFs) [6], *multispectral BTFs* (MSBTFs) [32], time- and spatially-varying BRDFs [12], time-varying light fields [2], have caught a lot of attention in computer graphics. They provide an example-based approach that can synthesize higher quality 3D images than traditional analytic models. To reduce the enormous amount of datasets for efficient data-driven rendering/processing, various novel compression algorithms [10, 28, 37] have been proposed during the past decades. Among modern compression algorithms, tensor decomposition, specifically *higher-order singular value decomposition* (HOSVD) [7, 40, 43], has been proved flexible and effective for analyzing and compressing large-scale multidimensional datasets.

When a tensor is too large to fit into host memory, out-of-core HOSVD can be employed for decomposition by exploiting external memory devices for data storage, such as hard disks and solid state drives. Nevertheless, it is often slow if the data amount is really huge. For a MSBTF dataset with a size of 48 GB, the algorithm of Wang *et al.* [43] may take several hours to decompose the MSBTF. As the data amount increases rapidly with time, constructing a database composed of various multidimensional datasets becomes time-consuming and impractical for real-world applications.

In this article, we present a method to improve the performance of out-of-core HOSVD. The proposed algorithm is based on two key insights:

- **Careful Scheduling of Blockwise Operations:** Out-of-core HOSVD often partitions a dataset into smaller blocks for efficient computation [43]. By carefully scheduling the

Received April 12, 2017; revised October 28, 2017; accepted December 18, 2017.
Communicated by Yung-Yu Chuang.

blockwise operations of HOSVD, not only the order of block accesses can be known in advance, but also the probability of reusing recently accessed blocks can be increased with a cache.
- **Heavy Linear Algebra Computations:** HOSVD mainly consists of linear algebra computations, including matrix multiplication and eigen-decomposition, on large/medium-size matrices.

The first insight suggests that the adopted block cache would operate similar to a simple first-in-first-out queue. After reexamining the algorithm flow of out-of-core HOSVD, we propose two forward and backward scheduling schemes to increase the hit rate of the block cache. The second insight indicates that the computation kernels of HOSVD are particularly suitable for GPU acceleration. By utilizing concurrent asynchronous CPU/GPU executions, matrix computations can be offloaded from CPUs to GPUs and overlapped with block accesses to hide related latencies and increase parallelism. Moreover, the two insights together imply that GPU memory can be regarded as an extra cache level on top of host memory. To store more in-core blocks (namely in host/GPU memory), we apply the exclusive caching technique, so that a block is constrained to reside in either the host or the GPU block cache, but not both.

In brief, this article makes following contributions:

- Propose two cache-aware static scheduling schemes to reduce the total number of disk accesses for out-of-core tensor decomposition.
- Introduce an efficient GPU-based out-of-core tensor decomposition framework.
- Apply the proposed method to decompose BTFs and MSBTFs for efficient rendering.

2. RELATED WORK

2.1 Tensor Decomposition

Tensor decomposition [7, 16], also called multilinear models or multiway analysis, was reported successful in compressing multidimensional datasets in computer graphics [40]. Its intrinsically multiway and flexible characteristics particularly allow high-quality data-driven rendering/processing. An out-of-core algorithm [43] was soon introduced to improve its performance for large-scale datasets. After that, various extensions have been proposed [31, 34-36, 38, 39, 45] to overcome the drawbacks of traditional methods. Interested readers may refer to the tensor survey/tutorial [2, 24] and references therein. Ballester-Ripoll *et al.* [1] also reviewed modern tensor models and conducted a detailed analysis on their decomposition and reconstruction performance.

Nevertheless, previous tensor models have focused on decomposition quality, reconstruction time, and/or compression ratios. Very few articles have addressed decomposition time for huge multidimensional datasets. For example, Wang *et al.* [43] proposed to partition a tensor into smaller blocks for efficient computation and employ an acceleration technique for reducing disk access time, which is conceptually similar to chunk-based matricization [12] and memory-efficient tensor mode products [25]. In this article, we further apply cache-aware static scheduling schemes and GPUs to substantially improve the performance of out-of-core HOSVD.

Moreover, GPUTensor [46], which also implements tensor decomposition on GPUs, is perhaps the most relevant to our work. Nevertheless, it is only suitable for sparse tensors and does not exploit host and GPU memory to cache tensor blocks. By contrast, our approach can handle both dense and sparse tensors and schedule blockwise operations to further reduce disk access time that usually dominates total decomposition time.

2.2 Out-of-Core Computation

Nowadays, a dataset is often very large such that it cannot fit into host memory and must be stored on the disk. To reduce disk access time, scheduling techniques are often employed to increase the probabilities of data reuse and sequential accesses [41]. A comprehensive survey on out-of-core computation is beyond the scope of this article. In the following paragraphs, we only briefly discuss out-of-core algorithms in linear algebra and computer graphics.

Marqués *et al.* [20] utilized GPUs to solve out-of-core linear systems, and suggested to employ a software cache and overlap computation with I/O transfers. Recently, Quintana-Ortí *et al.* [27] argued that it may not be necessary to find an optimal I/O schedule by tracking task dependencies. They also claimed that near-optimal performance would be achieved if data associated to a task could be prefetched while executing other tasks. This concept inspired us that cache-aware *static* scheduling, instead of more complex *dynamic* scheduling, may be adequate to our goal.

In computer graphics, the PantaRay system [26] was developed to efficiently perform GPU-based ray tracing on massive scenes with hierarchical acceleration structures. Wang *et al.* [44] demonstrated an out-of-core many-lights framework for rendering global illumination on GPUs, where data management was reformulated as a graph traversal problem for efficient processing. Günther and Grosch [13] presented a stochastic progressive photon mapping algorithm on CPUs and/or GPUs for out-of-core scenes by subdividing scenes and distributing workloads with coalesced tracing jobs across computers in a cluster. Nevertheless, most previous out-of-core algorithms in computer graphics tackle a specific application. We instead focus on general-purpose out-of-core computation on GPUs in this article.

2.3 Scheduling Techniques

Scheduling plays an important role not only in out-of-core algorithms but also in many other topics, such as databases [5], compilers [8], operating systems [21, 29], and distributed/cloud computing [4, 46]. Ceri *et al.* [5] improved the performance of detached rule scheduling for active database systems by employing dedicated threads to periodically execute detached rules. They also developed a performance model to adaptively determine the optimal number of threads and execution frequency at runtime. Eriksson and Kessler [8] integrated three code generation phases, including instruction selection (with cluster assignment), instruction scheduling, and register allocation, for clustered very long instruction word architectures. By formulating the problem into an integer linear programming model, one can explore more optimization opportunities for acyclic code and modulo scheduled loops.

Merkel *et al.* [21] proposed a resource-conscious operating system scheduler to improve both performance and energy efficiency for multicore processors by combining

tasks that can result in less resource contention. Ramaprasad and Mueller [29] analyzed the performance of data caches in a multi-task preemptive environment and derived tight upper bounds for real-time tasks. The estimated bounds thus can be utilized by a static or dynamic scheduler to significantly reduce the number of preemptions, the worse-case execution time, and the response time of a task. For distributed computing, Cao *et al.* [4] presented a heuristic scheduling technique for the directed acyclic graph workflow job in a Grid environment. They combined static task mapping and runtime dependable execution to achieve efficient performance and high resource utilization rates, while also providing fault tolerance. Recently, Zhou and He [46] introduced a flexible cloud computing framework for different offerings, workflows, and user requirements by identifying six basic transformation operations and effectively estimating their monetary costs and execution times for workflow optimization.

In this article, we focus on scheduling techniques for reducing the disk accesses time of out-of-core tensor decomposition. Our key idea is to employ a software data cache and schedule out-of-core operations in order to increase the cache hit rate. Since all out-of-core operations are known in advance before decomposition, it is adequate to just apply static scheduling from our experience.

2.4 General-Purpose GPU Computing

The massively parallel computing power of GPUs has driven a trend towards broader applications beyond computer graphics [22, 23]. For general-purpose computing, a GPU is regarded as a high-performance many-core processor. Through specialized programming languages, such as CUDA and OpenCL, single or multiple GPUs can be utilized to accelerate a wide variety of algorithms. This concept has been applied to solve many scientific/engineering problems. Due to length limitation, we only review previous work on linear/multilinear algebra operations, especially matrix multiplication and eigen-decomposition.

Krüger and Westermann [18] introduced a GPU-based linear algebra framework to provide a foundation for complex numerical algorithms. Fatahalian *et al.* [9] investigated that the slow performance of dense matrix multiplication on GPUs at that time was due to the inefficient use of GPU caches. Volkov and Demmel [42] proposed to improve the performance of dense matrix multiplication/factorization by matrix blocking and heterogeneous computing on CPUs and GPUs. Lahabar and Narayanan [19] presented a GPU-based implementation of singular value decomposition for dense matrices. Suter *et al.* [34] employed GPUs to accelerate multiscale tensor reconstruction and perform volume ray casting in realtime. Haidar *et al.* [14] developed a high-performance multi-GPU eigen-solver by reducing synchronization and data transfers among GPUs, at the expense of more compute-intensive tasks. Note that most previous GPU-based linear algebra methods only consider in-core operations, but this article further addresses out-of-core computation.

3. BACKGROUND

3.1 Notations

The transpose of a matrix $\mathbf{U} \in \mathbb{R}^{I \times J}$ is denoted by $\mathbf{U}^T$. The entry in row i and column j

of a matrix $\mathbf{U}$ is written as $(\mathbf{U})_{i,j}$; similarly, the entry of an Nth order tensor $\mathcal{A}\in\mathbb{R}^{I_1\times\cdots\times I_N}$ as $(\mathcal{A})_{i_1,\ldots,i_N}$. For a matrix $\mathbf{U}$ partitioned into uniform blocks with a size of $I'\times J'$, the submatrix $\mathbf{U}_{(i,j)}$ denotes the block (i, j) of $\mathbf{U}$, whose entries are

$$(\mathbf{U}_{(i,j)})_{k,l} = (\mathbf{U})_{(i-1)I'+k,(j-1)J'+l}. \tag{1}$$

The symbol $\langle\mathcal{A}, \mathcal{B}\rangle$ is the scalar product of two Nth order tensors $\mathcal{A}, \mathcal{B}\in\mathbb{R}^{I_1\times\cdots\times I_N}$. The Frobenius norm of an Nth order tensor $\mathcal{A}$ is written as $\|\mathcal{A}\|_F = \sqrt{\langle\mathcal{A},\mathcal{A}\rangle}$. Let $uf_n(\mathcal{A})\in\mathbb{R}^{I_n\times(I_{n+1}\cdots I_N I_1\cdots I_{n-1})}$ denote the mode-n unfolded matrix of $\mathcal{A}$, which is derived by retaining the nth mode of $\mathcal{A}$ and flattening the others [40, Fig. 2]. Namely, $uf_n(\mathcal{A})$ contains $(\mathcal{A})_{i_1,\ldots,i_N}$ in its row i_n and column j_n, where

$$j_n = i_{n+1} + \sum_{n_1=n+2}^{N}\left(i_{n_1}-1\right)I_{n+1}\cdots I_{n_1-1} + \sum_{n_1=1}^{n-1}\left(i_{n_1}-1\right)I_{n+1}\cdots I_N\prod_{n_2=1}^{n_1-1}I_{n_2}. \tag{2}$$

Refolding $uf_n(\mathcal{A})$ back into $\mathcal{A}$ is written as $uf_n^{-1}(uf_n(\mathcal{A}))$. The mode-$n$ product between a tensor $\mathcal{A}$ and a matrix $\mathbf{U}\in\mathbb{R}^{J_n\times I_n}$ is represented by $\mathcal{B}=\mathcal{A}\times_n\mathbf{U}$, where $\mathcal{B}\in\mathbb{R}^{I_1\times\cdots\times I_{n-1}\times J_n\times I_{n+1}\times\cdots\times I_N}$ is an Nth order tensor whose entries are

$$(\mathcal{A})_{i_1,\ldots,i_{n-1},j_n,i_{n+1},\ldots,i_N} = \sum_{i_n}(\mathcal{A})_{i_1,\ldots,i_N}(\mathbf{U})_{j_n,i_n}. \tag{3}$$

The mode-n product also can be rewritten in the matrix form as $uf_n(\mathcal{B}) = \mathbf{U}\cdot uf_n(\mathcal{A})$.

3.2 Out-of-Core HOSVD

HOSVD [7] decomposes an Nth order tensor $\mathcal{A}$ into an Nth order core tensor $\mathcal{Z}$ and a set of N basis matrices. Specifically, it can be formulated as the following constrained least-squares optimization problem:

$$\min_{\left\{\mathcal{Z},\{\mathbf{U}_n\}_{n=1}^{N}\right\}}\left\|\mathcal{A}-\mathcal{Z}\times_1\mathbf{U}_1^T\cdots\times_N\mathbf{U}_N^T\right\|_F^2, \text{s.t.}\ \forall n, \mathbf{U}_n\mathbf{U}_n^T = \mathbf{I}_{R_n}, \tag{4}$$

where $R_n\in\mathbb{Z}^+$ is the mode-n reduced rank, $\mathbf{U}_n\in\mathbb{R}^{R_n\times I_n}$ specifies the mode-n basis matrix, $\mathcal{Z}\in\mathbb{R}^{R_1\times\cdots\times R_N}$ denotes the decomposed core tensor, and $\mathbf{I}_{R_n}\in\mathbb{R}^{R_n\times R_n}$ represents the identity matrix of size $R_n\times R_n$. When $R_1, \ldots, R_N$ are sufficiently small, $\mathcal{Z}$ and $\{\mathbf{U}_n\}_{n=1}^{N}$ will give a compact representation for $\mathcal{A}$. A locally optimal solution to Eq. (4) can be derived by an iterative alternating least-squares algorithm, whose pseudocode is shown in Algorithm 1. To improve performance for a huge tensor stored on the disk, out-of-core HOSVD often partitions the tensor into smaller blocks [43], so that each block can fit into host memory. Thus, the original HOSVD operations on out-of-core tensors ($\mathcal{A}$, $\mathcal{A}_n$, and $\mathcal{Z}$ in Algorithm 1) must be performed blockwise. Since the size of the Gram matrix $\mathcal{A}_n$ is usually not large, the eigen-decomposition of $\mathcal{A}_n$ (line 8 in Algorithm 1) can be computed in core, leaving tensor unfolding/refolding and matrix multiplication (lines 6, 7, 11) as the most critical out-of-core operations. Note that as described in Section 3.1, the mode-n products in lines 6 and 11 are computed in the matrix form.

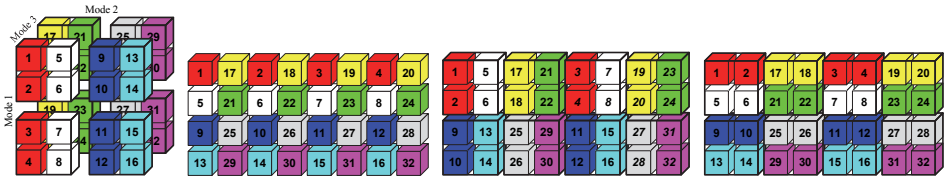
Similar to many out-of-core algorithms, out-of-core HOSVD also suffers from long disk access latencies that may take up to 40% of total decomposition time from our ex-

periments. Carefully-designed scheduling and parallel/heterogeneous computing are common techniques to solve this problem and hide latencies. We thus employ a block cache, schedule block accesses for matrix multiplication with special forward and backward orders to increase the cache hit rate, and utilize concurrent asynchronous CPU/GPU executions to hide block access latencies as many as possible. Sections 4 and 5 will respectively present our key ideas of scheduling and GPU acceleration in detail.

Algorithm 1: The HOSVD Algorithm

Input: $\mathcal{A}$, $\{R_n\}_{n=1}^{N}$, initial guess for $\{\mathcal{Z}, \{\mathbf{U}_n\}_{n=1}^{N}\}$, and convergence threshold ε.
Output: $\mathcal{Z}$ and $\{\mathbf{U}_n\}_{n=1}^{N}$.

1. **repeat**
2. $z \leftarrow \|\mathcal{Z}\|_F^2$
3. **for** $n \leftarrow 1$ **to** N **do** *// Update basis matrices*
4. $\mathcal{A}_n \leftarrow \mathcal{A}$
5. **for** $n' \leftarrow 1$ **to** N, $n' \neq n$ **do**
6. $\mathcal{A}_n \leftarrow uf_{n'}^{-1}(\mathbf{U}_{n'} \cdot uf_{n'}(\mathcal{A}_n))$ *// Mode-n' product*
7. $\mathbf{A}_n \leftarrow uf_n(\mathcal{A}_n) \cdot uf_n(\mathcal{A}_n)^T$ *// Compute Gram matrix*
8. Update $\mathbf{U}_n$ with the R_n dominant eigenvectors of $\mathbf{A}_n$
9. $\mathcal{Z} \leftarrow \mathcal{A}$
10. **for** $n \leftarrow 1$ **to** N **do** *// Update core tensor*
11. $\mathcal{Z} \leftarrow uf_n^{-1}(\mathbf{U}_n \cdot uf_n(\mathcal{Z}))$ *// Mode-n product*
12. **until** $\frac{\left|\|\mathcal{Z}\|_F^2 - z\right|}{\|\mathcal{A}\|_F^2} < \varepsilon$



(a) (b) (c) (d)

Fig. 1. *True* and *false* mode-2 unfolding for a third order tensor; (a) The tensor is partitioned into eight blocks, each of which is highlighted by a bold solid line cube; (b) True mode-2 unfolding reorganizes tensor entries regardless of blocks; (c) False mode-2 unfolding instead first reorganizes individual blocks; (d) It then only unfolds entries within each block.

3.3 False Unfolding and Refolding

To reduce disk access time, Wang *et al.* [43] proposed an acceleration technique that we called *false* unfolding/refolding. It is conceptually similar to chunk-based matricization [15] and memory-efficient tensor mode products [30]. As illustrated in Fig. 1, false unfolding regards each block in the tensor as a *big* tensor entry, then unfolds these big entries (Fig. 1 (c)), and only unfolds entries within each block (Fig. 1 (d)). Since unfolding big tensor entries is equivalent to reorganizing individual blocks (namely changing their indices), each block is guaranteed to be accessed only once and unnecessary

block reads/writes are avoided. Moreover, false refolding can be similarly implemented by reversing the process of false unfolding. Note that we also formally prove that false unfolding can lead to the same result of a mode-n product or the Gram matrix in the appendix, while the equivalence proof was not presented in previous work [43].

4. CACHE-AWARE STATIC SCHEDULING

The disk access time for blockwise operations (lines 6, 7, 11 in Algorithm 1) is often the most significant bottleneck in out-of-core HOSVD. Although dynamic scheduling can be applied to reduce the number of disk accesses, we have found that static scheduling is adequate. In general, static scheduling is much easier to implement. If an optimal schedule is unnecessary, the complex implementation of dynamic scheduling may incur too much runtime overhead [27]. We thus propose two static scheduling schemes for different types of multiplications. One scheme is designed for the mode-n product (Section 4.1) and the other for Gram matrix computation (Section 4.2). The two schemes are cache-aware due to the forward and backward orders for block accesses, which will be explained in detail in the following subsections.

4.1 Mode-n Product

In our implementation, a mode-n product $uf_n(\mathcal{C}) = \mathbf{U} \cdot uf_n(\mathcal{D})$ (in the matrix form, such as lines 6 and 11 in Algorithm 1) is performed on an in-core basis matrix $\mathbf{U}$ and an out-of-core unfolded matrix $uf_n(\mathcal{D})$ to obtain an out-of-core unfolded matrix $uf_n(\mathcal{C})$. We propose two scheduling sequences for different cases of the mode-n product. Sequence I reads each block in $uf_n(\mathcal{D})$ only once, but may read/write each block in $uf_n(\mathcal{C})$ multiple times (Section 4.1.1). Sequence II may read each block in $uf_n(\mathcal{D})$ multiple times, but instead writes each block in $uf_n(\mathcal{C})$ only once (Section 4.1.2). Each time before executing a mode-n product, an appropriate sequence is automatically determined (Section 4.1.3). Note that all basis matrices are stored in core, since their sizes are usually small.

4.1.1 Sequence I

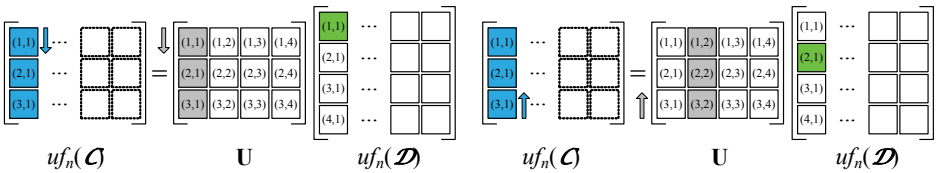
In Sequence I, each block in the input $uf_n(\mathcal{D})$ is read only once and streamed for multiplication with blocks in $\mathbf{U}$. The multiplied results are then accumulated with associated blocks in the output $uf_n(\mathcal{C})$. Specifically, blocks in $uf_n(\mathcal{C})$ are computed by

$$\forall i, \forall j, uf_n(\mathcal{C})_{(i,j)} = \sum_k uf_n(\mathcal{C})^{(k)}_{(i,j)}, \tag{5}$$

$$uf_n(\mathcal{C})^{(k)}_{(i,j)} = \mathbf{U}_{(i,k)} \cdot uf_n(\mathcal{D})_{(k,j)}, \tag{6}$$

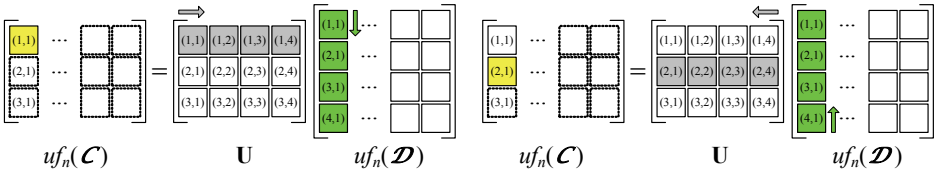
where $uf_n(\mathcal{C})^{(k)}_{(i,j)}$ denotes the kth partial multiplied result of $uf_n(\mathcal{C})_{(i,j)}$. In practice, we iterate the index j to obtain blocks $uf_n(\mathcal{C})_{(1,j)}$, …, $uf_n(\mathcal{C})_{(B^{\mathcal{C}}_n,j)}$ at each iteration, where $B^{\mathcal{C}}_n$ is the number of blocks along the nth mode of $\mathcal{C}$, and Fig. 2 illustrates Sequence I with the special forward and backward orders when $j = 1$. At the jth iteration, the block $uf_n(\mathcal{D})_{(1,j)}$ is first read from the disk and multiplied with $\mathbf{U}_{(1,1)}$, …, $\mathbf{U}_{(B^{\mathcal{C}}_n,1)}$ (Fig. 2 (a)). The first partial results $uf_n(\mathcal{C})^{(1)}_{(1,j)}$, …, $uf_n(\mathcal{C})^{(1)}_{(B^{\mathcal{C}}_n,j)}$ are then written to associated blocks in $uf_n(\mathcal{C})$ in the

(forward) order of $uf_n(\mathcal{C})_{(1,j)}$, …, $uf_n(\mathcal{C})_{(B_n^{\mathcal{C}},j)}$. After that, the next block $uf_n(\mathcal{D})_{(2,j)}$ is read for multiplication with $\mathbf{U}_{(1,2)}$, …, $\mathbf{U}_{(B_n^{\mathcal{C}},2)}$, and the second partial results $uf_n(\mathcal{C})^{(2)}_{(1,j)}$, …, $uf_n(\mathcal{C})^{(2)}_{(B_n^{\mathcal{C}},j)}$ are accumulated in the (backward) order of $uf_n(\mathcal{C})_{(B_n^{\mathcal{C}},j)}$, …, $uf_n(\mathcal{C})_{(1,j)}$ (Fig. 2 (b)). Again, blocks $uf_n(\mathcal{C})_{(1,j)}$, …, $uf_n(\mathcal{C})_{(B_n^{\mathcal{C}},j)}$ are updated forward, backward, and so on, until their final results are obtained. This scheduling technique could immediately reuse recently accessed blocks in $uf_n(\mathcal{C})$ when switching from forward to backward, and vice versa, and the cache hit rate would be effectively increased. If the employed cache can hold at most E blocks in $uf_n(\mathcal{C})$, there will be only $B_n^{\mathcal{C}} - E$ misses when accessing blocks in the backward order as Fig. 2 (b).



(a) Forward order (b) Backward order

Fig. 2. Sequence I for a mode-n product. Each arrow shows the order of block accesses. Cyan blocks are first read from the disk (if needed), updated, and then written to the disk. Green blocks indicate that they are read from the disk, and gray ones are read from memory. Solid line blocks are valid ones stored in memory or on the disk, while dotted line blocks are invalid ones that have not been generated.



(a) Forward order (b) Backward order

Fig. 3. Sequence II for a mode-n product; Yellow blocks indicate that they are written to the disk. Refer to Fig. 2 for the meanings of other objects and colors.

4.1.2 Sequence II

In this sequence, each block in the output $uf_n(\mathcal{C})$ is respectively computed by reading associated blocks in the input $uf_n(\mathcal{D})$ for multiplication with blocks in $\mathbf{U}$ and then written to the disk only once. Similar to Sequence I, Sequence II also switches between the forward and backward block access orders, but only blocks in $uf_n(\mathcal{D})$ are cached instead. Formally, blocks in $uf_n(\mathcal{C})$ are computed by

$$\forall i, \forall j, uf_n(\mathcal{C})_{(i,j)} = \sum_k \mathbf{U}_{(i,k)} \cdot uf_n(\mathcal{D})_{(k,j)}. \tag{7}$$

Sequence II iterates the index j to obtain blocks in $uf_n(\mathcal{C})$, and Fig. 3 shows how it works on an example when $j = 1$. At the jth iteration, the block $uf_n(\mathcal{C})_{(i,j)}$ is first derived by reading associated blocks in $uf_n(\mathcal{D})$ in the (forward) order of $uf_n(\mathcal{D})_{(1,j)}$, …, $uf_n(\mathcal{D})_{(B_n^{\mathcal{D}},j)}$, where $B_n^{\mathcal{D}}$ is the number of blocks along nth mode of $\mathcal{D}$, and multiplying them respectively with $\mathbf{U}_{(1,1)}$, …, $\mathbf{U}_{(1,B_n^{\mathcal{D}})}$. The next block $uf_n(\mathcal{C})_{(i+1,j)}$ is then obtained by reading blocks in the (backward) order of $uf_n(\mathcal{D})_{(B_n^{\mathcal{D}},j)}$, …, $uf_n(\mathcal{D})_{(1,j)}$ and multiplying them with $\mathbf{U}_{(2,1)}$, …,

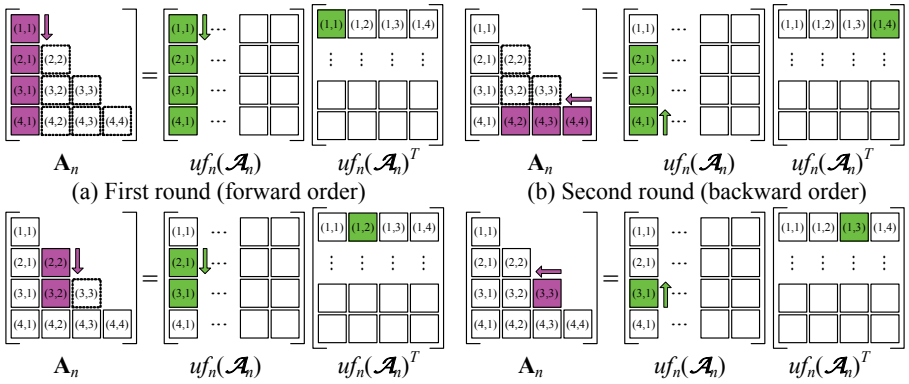
$\mathbf{U}_{(2,B_2^{\mathcal{D}})}$ (Fig. 3 (b)). After that, Sequence II turns back to the forward order again for computing $uf_n(\mathcal{C})_{(i+2,j)}$, and switches between forward and backward orders for reading $uf_n(\mathcal{D})_{(1,j)}$, …, $uf_n(\mathcal{D})_{(B_2^{\mathcal{D}},j)}$ until $uf_n(\mathcal{C})_{(B_2^{\mathcal{C}},j)}$ is derived. The recently accessed blocks in $uf_n(\mathcal{D})$ thus could be immediately reused when the reading order changes.

4.1.3 Automatic sequence selection

With static scheduling, we can mathematically analyze the total numbers of required block accesses for both sequences (as listed in Table 1) and automatically select an appropriate one that would access blocks less frequently. Specifically, each time before executing a mode-n product, the numbers of block reads and writes are computed for each sequence. We then select the sequence with the smallest value of the metric: $L_s + \alpha W_s$, where L_s and W_s respectively denote the number of block reads and writes for a sequence s, and α is a constant that accounts for the average time ratio of a block write over a block read. The value of α may vary from system to system and can be determined offline by conducting some experiments. Note that we ignore hardware caching and memory latency issues for sequence selection, since disk access time is more dominant.

Table 1. Number of block accesses for a mode-n product. The scalars $B_j^{\mathcal{C}}$ and $B_j^{\mathcal{D}}$ denote the numbers of blocks along the jth mode of $\mathcal{C}$ and $\mathcal{D}$, respectively. The scalar E represents the number of maximal blocks in the cache. The output of the function $\max_0(a)$ is the maximum of the scalar a and 0.

Sequence	Operation	Matrix	Number of block accesses
I	Read	$uf_n(\mathcal{C})$	$(B_n^{\mathcal{D}}-1)\cdot\max_0(B_j^{\mathcal{C}}-E)\cdot\prod_{j\neq n}B_j^{\mathcal{D}}$
		$uf_n(\mathcal{D})$	$\prod_j B_j^{\mathcal{D}}$
	Write	$uf_n(\mathcal{C})$	$((B_n^{\mathcal{D}}-1)\cdot\max_0(B_n^{\mathcal{C}}-E)+B_n^{\mathcal{C}})\cdot\prod_{j\neq n}B_j^{\mathcal{D}}$
II	Read	$uf_n(\mathcal{D})$	$((B_n^{\mathcal{C}}-1)\cdot\max_0(B_n^{\mathcal{D}}-E)+B_n^{\mathcal{D}})\cdot\prod_{j\neq n}B_j^{\mathcal{D}}$
	Write	$uf_n(\mathcal{C})$	$B_n^{\mathcal{C}}\cdot\prod_{j\neq n}B_j^{\mathcal{D}}$



(a) First round (forward order) (b) Second round (backward order)

(a) First round (forward order) (b) Second round (backward order)

Fig. 4. The proposed scheduling scheme for Gram matrix computation. Magenta blocks are first read from memory (if needed), updated, and then written to memory. Refer to Fig. 2 for the meanings of other objects and colors.

Sequence I is generally suitable when the number of rows in $uf_n(\mathcal{C})$ is smaller than that in $uf_n(\mathcal{D})$, since the employed cache is more likely to hold the required blocks in $uf_n(\mathcal{C})$. By contrast, Sequence II is suitable for the opposite case. It is worth noting that both sequences are respectively related to the GEBP and GEPDOT operations that are the building kernels of matrix multiplication in GotoBLAS [11]. Nevertheless, GotoBLAS only considers in-core matrix multiplication, while out-of-core multiplication is encountered in our case.

4.2 Gram Matrix Computation

Computing the Gram matrix (line 7 in Algorithm 1) is one of the critical operations in out-of-core HOSVD, where there are N total computations at each *repeat* iteration in Algorithm 1. For the nth mode, Gram matrix computation corresponds to the matrix multiplication between an out-of-core unfolded matrix $uf_n(\mathcal{A}_n)$ and its transpose $uf_n(\mathcal{A}_n)^T$ to obtain an in-core Gram matrix $\mathbf{A}_n$ (whose size is usually not large). The proposed scheduling scheme for computing $\mathbf{A}_n$ also employs the forward and backward orders to read blocks in $uf_n(\mathcal{A}_n)$. Since $\mathbf{A}_n$ is a symmetric matrix, we can only compute its lower triangular part to further reduce the number of block accesses. Specifically, the lower triangular blocks in $\mathbf{A}_n$ are computed by

$$\forall i, \forall j \le i, (\mathbf{A}_n)_{(i,j)} = \sum_k (\mathbf{A}_n)^{(k)}_{(i,j)}, \tag{8}$$

$$(\mathbf{A}_n)^{(k)}_{(i,j)} = uf_n(\mathcal{A}_n)_{(i,k)} \cdot \left(uf_n(\mathcal{A}_n)^T\right)_{(k,j)}, \tag{9}$$

where $(\mathbf{A}_n)^{(k)}_{(i,j)}$ specifies the kth partial multiplied result of $(\mathbf{A}_n)_{(i,j)}$. In this scheme, we iterate the index k to accumulate the lower triangular blocks in $\mathbf{A}_n$, and Fig. 4 demonstrates an example of the proposed scheme when $k = 1$. At the kth iteration, blocks $(\mathbf{A}_n)_{(1,1)}$, …, $(\mathbf{A}_n)_{(B_n^{\mathcal{A}_n},1)}$, where $B_n^{\mathcal{A}_n}$ is the number of blocks along nth mode of $\mathcal{A}_n$, are first updated by reading associated blocks in $uf_n(\mathcal{A}_n)$ in the forward order of $uf_n(\mathcal{A}_n)_{(1,k)}$, …, $uf_n(\mathcal{A}_n)_{(B_n^{\mathcal{A}_n},k)}$ and multiplying them with $(uf_n(\mathcal{A}_n)^T)_{(k,1)}$ (Fig. 4 (a)). Next, blocks $uf_n(\mathcal{A}_n)_{(2,k)}$, …, $uf_n(\mathcal{A}_n)_{(B_n^{\mathcal{A}_n},k)}$, without $uf_n(\mathcal{A}_n)_{(1,k)}$ at this time, are read in the backward order and multiplied with $(uf_n(\mathcal{A}_n)^T)_{(k,\,B_n^{\mathcal{A}_n})}$ to update associated blocks in $\mathbf{A}_n$, namely $(\mathbf{A}_n)_{(B_n^{\mathcal{A}_n},2)}$, …, $(\mathbf{A}_n)_{(B_n^{\mathcal{A}_n},\,B_n^{\mathcal{A}_n})}$ (Fig. 4 (b)). Then, the above process is similarly applied until all lower triangular blocks in $\mathbf{A}_n$ are updated (Figs. 4 (c) and (d)). Note that each time when switching the reading order, one block in $uf_n(\mathcal{A}_n)$ is not needed henceforth, and the required block in $uf_n(\mathcal{A}_n)^T$ is already in the cache. This style of reading blocks in $uf_n(\mathcal{A}_n)$ is analogous to a *underdamped* spring/oscillator. The total number of block reads for $uf_n(\mathcal{A}_n)$ can be estimated by

$$\left(B_n^{\mathcal{A}_n} + \frac{\max_0\left(B_n^{\mathcal{A}_n} - E\right)\max_0\left(B_n^{\mathcal{A}_n} - E - 1\right)}{2}\right) \cdot \prod_{j \ne n} B_j^{\mathcal{A}_n}, \tag{10}$$

where we use the same notation as in Table 1.

5. GPU ACCELERATION

From Section 3.2 and Algorithm 1, one can easily find out that most operations in

out-of-core HOSVD are linear algebra computations on large/medium-size matrices, including matrix multiplication (lines 6, 7, 11) and eigen-decomposition (line 8), which are particularly suitable for GPU acceleration. Our implementation thus employs asynchronous GPU executions to overlap operations on CPUs and GPUs. Specifically, we only computes blockwise *submatrix* multiplication and eigen-decomposition asynchronously on GPUs and utilizes CPUs to handle other tasks, such as program flow control, disk accesses, and block cache management. To further increase GPU parallelism, concurrent data transfers and GPU kernel executions are enabled by batch processing [25]. Namely, operations are divided into batches, each of which includes the data transfers of different submatrices between host and GPU memory and the corresponding submatrix multiplications, such as Eq. (6) for Sequence I, each term of the summation in Eq. (7) for Sequence II, and Eq. (9) for Gram matrix computation. Note that synchronization among different batches/GPUs must be carefully coordinated to avoid data corruption. Fig. 5 illustrates the concept of concurrent asynchronous CPU/GPU executions and batch processing in our system.

Moreover, modern GPUs are equipped with a large amount of memory. Since blockwise operations often consume just a portion of GPU memory, we can regard unused GPU memory as an extra cache level of the memory hierarchy to reduce unnecessary data transfers between host and GPU memory. In our implementation, cache blocks in GPU memory are different from those in host memory, namely exclusive contents in the two caches. This allows more cache blocks in the memory hierarchy and improves overall decomposition performance, since disk accesses are much more time-consuming than data transfers between host and GPU memory.

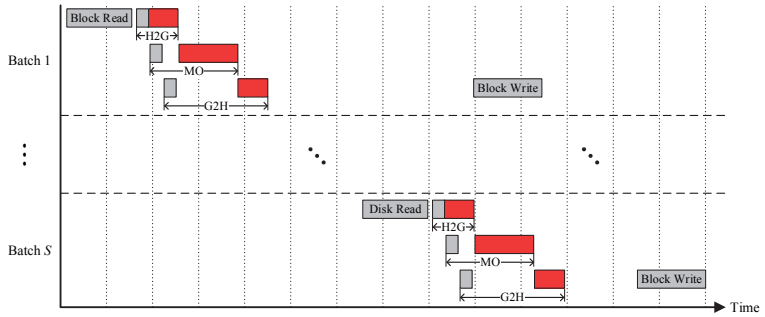


Fig. 5. We employ asynchronous GPU executions to overlap CPU/GPU operations and divide them into multiple batches to maximize the throughput of our system. Gray rectangles are tasks handled by CPUs, while red ones are executed on GPUs. "H2G" stands for data transfer from host to GPU memory, "G2H" for transfer from GPU to host, and "MO" specifies (blockwise) matrix operations, such as multiplication and/or eigen-decomposition. Block reads/writes are handled by CPUs and implicitly include cache/disk accesses.

6. EXPERIMENTAL RESULTS

This section shows the experimental results of the proposed method on BTFs (Section 6.1) and MSBTFs (Section 6.2), which are two common types of large-scale multi-dimensional datasets in computer graphics. The following parameters were adopted in

our experiments unless otherwise specified. An input tensor was partitioned along various modes into multiple blocks, each with a size of 60.75 MB. For GPU acceleration, the number of batches was set to 3. For cache-aware scheduling, the total number of maximal blocks in the cache was set to 9 (6 in host memory and 3 in GPU memory), leading to a memory footprint of around 1 GB (60.75MB9×cache blocks = 546.75MB, plus about 400 MB temporary data, where each GPU batch needs 2 additional blocks, one in host memory and the other in GPU memory, for concurrent data transfers and CPU/GPU asynchronous executions). Decomposition quality is evaluated by the *signal-to-noise ratio* (SNR). Timings were measured on a workstation with an Intel Core i7-4930K CPU, an NVIDIA GeForce GTX TITAN graphics card (with 6 GB GPU memory), and 16 GB host memory. The raw and decomposed data were stored using 32-bit and 16-bit floating point numbers, respectively.

6.1 Bidirectional Texture Functions

A BTF [6] is a 6D function that describes the reflectance of a planar surface point when lit by an incident ray in the illumination direction ω_l and reflected in the view direction ω_v, where the surface point is usually represented as a texel **t** with 2D spatial coordinates (x, y). HOSVD was proved as an efficient and flexible BTF compression method [40, 43]. Although it may not be the most efficient one, the flexibility of individually reducing each mode is the main reason that we use it. Nevertheless, out-of-core HOSVD may consume substantial time for decomposition.

Experimental settings A BTF is organized as a fourth order tensor $\mathcal{A} \in \mathbb{R}^{I_{\omega_l} \times I_{\omega_v} \times I_x \times I_y}$ for decomposition. To achieve fast rendering performance, we only decompose its illumination and view modes. This allows texture filtering for the x and y modes on GPUs and also reflects the real memory usage at runtime. The utilized BTF datasets were collected from the UBO2003 Datasets [33] in BTF Database Bonn (http://cg.cs.uni-bonn.de/projects/btfdbb/download/ubo2003/) and the Volumetric Surface Texture Database [17] (http://vision.ucsd.edu/kriegman-grp/research/vst/).

Results: Table 2 shows the statistics of BTF decomposition using five methods. Readers may refer to our accompanying video and supplemental materials for the reconstructed/rendered results. The performance of the proposed method can be reduced up to almost 71% when compared with Wang *et al.* [43]. For a BTF, the SNRs of CPU- and GPU-based methods should be theoretically identical, but are slightly different due to adopted eigen-solvers and numerical issues, with almost equivalent rendering rates, the same amount of decomposed data, and indisquinshable visual quality of reconstructed/rendered images (please refer to our supplemental materials). For eigen-decomposition, we currently employ "culaDeviceSsyev" in CULA (http://www.culatools.com/) for GPU-based methods and "eig" in MATLAB for others. Fig. 6 compares the speedups over Wang *et al.* [43] among different configurations of the proposed method. For the BTF "Lego", we changed the mode-ω_l or mode-ω_v reduced rank from 8 to 88 and conducted total 121 experiments for different combinations of the two parameters. The speedup generally changes a lot with the increase in the mode-ω_v reduced rank R_{ω_v}, but first raises and then decreases with the increase in the mode-ω_l reduced rank R_{ω_l}.

Table 2. Statistics of BTF decomposition using five methods, including the naive block-wise implementation with true unfolding/refolding (N), Wang *et al.* [43] (with false unfolding/refolding), cache-aware scheduling with false unfolding/refolding (C), GPU acceleration with false unfolding/refolding (G), and all the proposed acceleration techniques (C+G). The rendering rates were measured under 3 directional lights with a screen resolution of 800×600.

BTF Object model	Corduroy Armadillo					Impalla Room					Lego Teapot					Proposte Dragon				
$I_{\omega_l} \times I_{\omega_v} \times I_x \times I_y$ Raw data (GB)	81 × 81 × 256 × 256 4.81					81 × 81 × 256 × 256 4.81					120 × 90 × 256 × 256 7.91					81 × 81 × 256 × 256 4.81				
$R_{\omega_l} \times R_{\omega_v}$ Decomposed data (MB)	24 × 32 96.02					32 × 36 144.02					32 × 48 192.03					24 × 40 120.02				
Frames per second	~50.37					~14.29					~35.87					~33.27				
Method	N	[43]	C	G	C+G	N	[43]	C	G	C+G	N	[43]	C	G	C+G	N	[43]	C	G	C+G
SNR (dB)	19.37	19.37	19.37	19.31	19.31	18.91	18.91	18.91	18.85	18.85	17.25	17.25	17.25	17.24	17.24	23.69	23.69	23.69	23.33	23.33
Performance (min.)	10.27	3.53	2.32	1.12	0.78	30.02	10.53	6.75	3.1	2.05	30.85	9.93	6.15	3.92	3.08	33.35	12.05	7.47	3.35	2.28
Speedup over N	-	2.91	4.43	9.19	13.11	-	2.85	4.45	9.68	14.64	-	3.11	5.02	7.88	10.01	-	2.77	4.47	9.96	14.61
Speedup over [43]	0.34	-	1.53	3.16	4.51	0.35	-	1.56	3.4	5.14	0.32	-	1.62	2.54	3.22	0.36	-	1.61	3.6	5.28

BTF Object model	Pulli Horse					Sponge Bunny					Wool Cloth				
$I_{\omega_l} \times I_{\omega_v} \times I_x \times I_y$ Raw data (GB)	81 × 81 × 256 × 256 4.81					120 × 90 × 256 × 256 7.91					120 × 90 × 256 × 256 4.81				
$R_{\omega_l} \times R_{\omega_v}$ Decomposed data (MB)	24 × 48 144.02					20 × 24 60.02					24 × 28 84.02				
Frames per second	~26.73					~86.29					~38.23				
Method	N	[43]	C	G	C+G	N	[43]	C	G	C+G	N	[43]	C	G	C+G
SNR (dB)	20.21	20.21	20.21	20.15	20.15	25.72	25.72	25.72	25.23	25.23	20.65	20.65	20.65	20.55	20.55
Performance (min.)	8.63	3.33	2.07	1.05	0.75	19.08	4.73	3.15	1.83	1.47	31.42	10.32	6.42	3.05	2.0
Speedup over N	–	2.59	4.18	8.22	11.51	–	4.03	6.06	10.41	13.01	–	3.05	4.9	10.3	15.71
Speedup over [43]	0.39	–	1.61	3.17	4.44	0.25	–	1.5	2.58	3.23	0.33	–	1.61	3.38	5.16

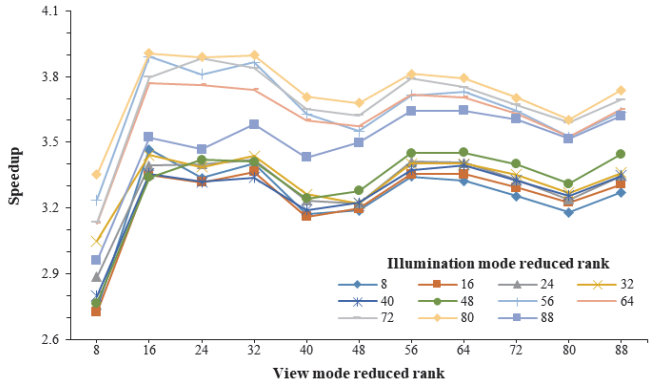


Fig. 6. Speedups over Wang *et al.* [43] for the BTF "Lego" based on different configurations of the proposed method (*C*+*G*).

6.2 Multispectral Bidirectional Texture Functions

A MSBTF [32] is a generalization of the BTF with multispectral responses. It is a 7D function of the illumination direction ω_l, the view direction ω_v, the wavelength λ, and

the 2D spatial coordinates (x, y) of a texel $\mathbf{t}$. The additional dependency on wavelengths further increases the required amount of each raw MSBTF dataset to tens of GB. It is thus time-consuming to compress such a huge dataset using out-of-core HOSVD.

Experimental settings A MSBTF is organized as a fifth order tensor $\mathcal{A} \in \mathbb{R}^{I_{\omega_l} \times I_{\omega_v} \times I_\lambda \times I_x \times I_y}$, and we only decompose its illumination, view, and wavelength modes. The adopted MSBTF datasets were collected from the Spectral Datasets [32] in BTF Database Bonn (http://cg.cs.uni-bonn.de/projects/btfdbb/download/spectral/).

Rendering process The MSBTF rendering process is similar to BTF, except that we should additionally reconstruct the wavelength mode and apply a color matching function $C(\cdot)$ to convert the spectral power distribution into RGB color values. Specifically, we obtain the reflected RGB values from a MSBTF texel $\mathbf{t}$ by

$$\sum_i [C(\mathbf{l}_i \circ (\mathbf{U}_\lambda)_{1,*}^T)...C(\mathbf{l}_i \circ (\mathbf{U}_\lambda)_{R_\lambda,*}^T)] \cdot uf_\lambda(\mathcal{Z}^{(\mathbf{t})} \times_{\omega_l} (\mathbf{u}_{\omega_l}^{(i)})^T \times_{\omega_v} \mathbf{u}_{\omega_v}^T), \tag{11}$$

where $\mathbf{l}_i \in \mathbb{R}^{I_\lambda}$ is the incident spectral power distribution from the ith light source, the symbol $\circ$ denotes the Hadamard (entrywise) product, $(\mathbf{U}_\lambda)_{j,*}$ specifies the jth row vector of $\mathbf{U}_\lambda$, $\mathbf{u}_{\omega_l}^{(i)} \in \mathbb{R}^{R_{\omega_l}}$ and $\mathbf{u}_{\omega_v} \in \mathbb{R}^{R_{\omega_v}}$ are respectively sampled from $\mathbf{U}_{\omega_l}$ and $\mathbf{U}_{\omega_v}$ according to the ith illumination and current view directions, and $\mathcal{Z} \in \mathbb{R}^{R_{\omega_l} \times R_{\omega_v} \times R_\lambda}$ is the associated entries of $\mathbf{t}$ in $\mathcal{Z}$.

To increase runtime rendering rates, we precompute $\{C(\mathbf{l}_i \circ (\mathbf{U}_\lambda)_{j,*}^T)\}_{j=1}^{R_\lambda}$ for each light source and store the results in a texture. This may also reduce memory requirements as long as the number of lights is small (2 in our experiments). Nevertheless, the texture needs to be updated each time when the spectral emissive power distribution of a light source changes.

Table 3. Statistics of MSBTF decomposition using five methods (please refer to Table 2 for abbreviations). The rendering rates were measured under 2 directional lights with a screen resolution of 640×480.

MSBTF	Colorchecker					Lego Bricks					Red Fabric					Wallpaper				
Object model	Plane					Cloth					Bunny					Teapot				
$I_{\omega_l} \times I_{\omega_v} \times I_\lambda \times I_x \times I_y$	81 × 81 × 30 × 256 × 256					81 × 81 × 30 × 256 × 256					81 × 81 × 30 × 256 × 256					81 × 81 × 29 × 256 × 256				
Raw data (GB)	48.05					48.05					48.05					46.45				
$R_{\omega_l} \times R_{\omega_v} \times R_\lambda$	44 × 52 × 3					36 × 56 × 3					28 × 48 × 3					40 × 48 × 3				
Decomposed data (MB)	858.02					756.01					504.01					720.01				
Frames per second	~8.2					~7.83					~15.03					~11.84				
Method	N	[43]	C	G	C+G	N	[43]	C	G	C+G	N	[43]	C	G	C+G	N	[43]	C	G	C+G
SNR (dB)	12.12	12.12	12.12	12.12	12.12	11.62	11.62	11.62	11.62	11.62	15.47	15.26	15.26	15.4	15.4	17.78	17.67	17.67	17.72	17.72
Performance (hr.)	24.86	5.81	4.93	2.67	2.27	9.79	2.3	1.69	1.07	0.87	10.99	2.43	1.81	1.18	0.94	9.6	2.25	1.87	1.05	0.86
Speedup over N	–	4.28	5.04	9.3	10.93	–	4.27	5.78	9.14	11.21	–	4.53	6.06	9.35	11.68	–	4.26	5.13	9.11	11.13
Speedup over [43]	0.23	–	1.18	2.17	2.56	0.23	–	1.36	2.14	2.63	0.22	–	1.34	2.07	2.58	0.23	–	1.2	2.14	2.61

Results Table 3 lists the statistics of MSBTF decomposition, where we also compare five methods. The reconstructed/rendered images can be found in our accompanying video and supplemental materials. GPU acceleration typically achieves a higher speedup than cache-aware scheduling, but the performance gains of cache-aware scheduling still

cannot be ignored. Fig. 7 plots the speedup over Wang *et al.* [43] of the proposed method versus the block size. It shows that the proposed method can achieve a considerable speedup regardless of the block size. Although increasing the block size (while being less than 200 MB) would reduce the processing time of Wang *et al.* [43], it usually does not have a strong effect on the proposed method. Note that increasing the block size also reduces the total number of blocks in the input tensor, resulting in fewer number of disk accesses but longer access time for each block. The limited amount of host/GPU memory usually prevents users from employing a too large block size.

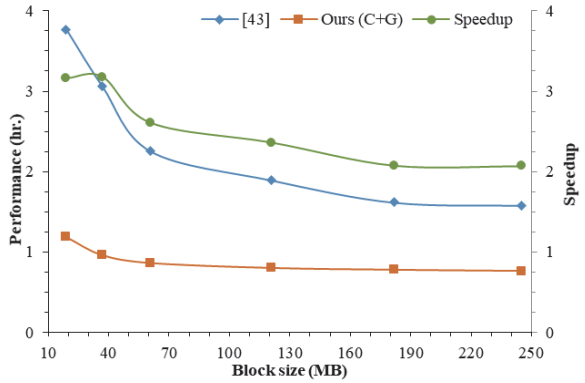


Fig. 7. Speedups over Wang *et al.* [43] for the MSBTF "Wallpaper" based on different block sizes of the proposed method.

Table 4. Statistics of decomposing the MSBTF "Wallpaper" with various numbers of maximal cache blocks.

Method	[43]	C+G			
Block size (MB)	60.75	60.75			
Blocks in host cache	–	3	6	6	9
Blocks in GPU cache	–	3	3	6	3
Performance (min.)	135.12	51.82	51.77	50.7	51.7
Speedup	–	2.61	2.61	2.67	2.61

Table 4 lists the statistics with various numbers of maximal cache blocks for the proposed method. Increasing the cache block number often improves performance, but when the total block number in both caches exceeds necessity, the overhead of maintaining redundant cache blocks may instead reduce performance (as the last case in Table 4). Raising the block number in the GPU cache also provides more performance gains than increasing that in the host cache. Regarding unused GPU memory as an extra cache level indeed unleashes additional performance due to less data transfers between host and GPU memory. Table 5 shows the statistics with similar memory footprints. For each configuration, we adjust the block size and the number of maximal blocks in the host/GPU cache to have a memory footprint of about 1 GB. The performance of different configurations generally improves with the increase in the block size, but their differ-

Table 5. Statistics of decomposing the MSBTF "Wallpaper" with similar memory footprints.

Method	[43]	C+G	[43]	C+G	[43]	C+G
Block size (MB)	48	48	60.75	60.75	75	75
Blocks in host cache	–	9	–	6	–	3
Blocks in GPU cache	–	5	–	3	–	3
Performance (min.)	159.03	55.03	135.12	51.77	148.2	51.35
Speedup	–	2.89	–	2.61	–	2.89

Table 6. Statistics of decomposing the MSBTF "Red Fabric" with different numbers of GPU batches.

Method	[43]	C	C+G	C	C+G	C	C+G
Number of batches	–	1	1	2	2	3	3
Performance (min.)	145.68	78.17	64.95	71.28	60.4	70.53	56.45
Speedup	–	1.86	2.24	2.04	2.41	2.07	2.58

ences are rather small. This implies that the influence of parameter settings is insignificant with fixed and limited memory usage. Table 6 compares the results with different numbers of GPU batches. A large batch number usually increases performance, but also consumes more memory space.

6.3 Discussions

The proposed method can achieve a considerable performance gain over Wang *et al.* [43] for BTF and MSBTF decomposition. Cache-aware scheduling provides a speedup of 1.2~1.6, GPU acceleration gives 2.1~3.6, and the two techniques can be seamlessly integrated to achieve 2.5~5.3. For cache-aware scheduling, the proposed forward and backward schemes are likely to reuse recently accessed blocks. A significant number of disk accesses thus could be avoided. Nevertheless, the speedup for a MSBTF is usually lower than that for a BTF. Since the data amount of a MSBTF is much larger, disk access time becomes the dominant bottleneck over other operations. The numbers of compulsory and capacity misses of the block cache also rapidly increase, leading to poorer cache efficiency and a lower speedup (including GPU acceleration). Although raising the number of maximal cache blocks would alleviate this issue, compulsory misses still have a great performance impact.

It is worth noting that the scheduling scheme for the mode-n product in Wang *et al.* [43] is actually close to Sequence II (Section 4.1.2) by writing each block in the output tensor only once, but without the host/GPU block cache and the special forward and backward orders for block accesses. By contrast, our system further employs two cache-aware scheduling sequences to avoid unnecessary disk accesses and can automatically select an appropriate sequence at runtime. According to our experience, Sequence I sometimes can lead to less disk accesses than Sequence II. With automatic sequence selection, our system is more likely to find a better scheduling sequence for the mode-n product. Moreover, our GPU-based computations and the extra GPU block cache also have a great impact on the speedup over Wang *et al.* [43].

We currently rely on concurrent asynchronous CPU/GPU executions and exclusive CPU/GPU block caches to balance disk accesses and data transfers between host and GPU memory. By applying asynchronous CPU/GPU executions, matrix computations can be overlapped with block accesses to hide related latencies. Exclusive block caching also effectively utilizes the capacity of host/GPU memory and the bandwidth between them. Our experiments in Table 4 further shows that by changing the numbers of maximal blocks in the host and GPU caches, one may find a good balance of disk accesses and data transfers between host and GPU memory. When the total block number in both caches does not exceed necessity, raising the block number in the GPU cache often provides more performance gains than increasing that in the host cache. This implies that the latencies of disk accesses can be well hidden by asynchronous CPU/GPU executions.

7. CONCLUSIONS

This article presents a novel method to improve the performance of out-of-core tensor decomposition. Our key ideas include cache-aware scheduling and a GPU-based decomposition framework. The proposed scheduling schemes significantly reduce the total number of disk accesses for computing the mode-n product or the Gram matrix. Various linear algebra operations of tensor decomposition also can be effectively accelerated on GPUs. For BTF and MSBTF decomposition, the proposed method can achieve speedups of 11~16 over a naive implementation and 2.5~5.3 over previous work [43].

This article is just an initial step in addressing the scalability issue of multilinear models for massive multidimensional datasets. In the future, we are interested in extending the proposed method to accelerate advanced multilinear models, such as MK-CTA [36]. We would also like to investigate the performance challenge of multilinear models for big data analysis, including CPU-GPU heterogeneous computing and efficient distributed algorithms.

APPEXNDIX

This appendix proves that false unfolding/refolding (Section 3.3) can lead to the same result of a mode-n product with true unfolding/refolding. The proof for Gram matrix computation is omitted, since it can be similarly derived.

A mode-n product can be written in the matrix form as $uf_n(\mathcal{A}\times_n\mathbf{U}) = \mathbf{U}\cdot uf_n(\mathcal{A})$ (Section 3.1). From the definition of true mode-n unfolding in Eq. (2) and Fig. 1, it is easy to find out that false mode-n unfolding can be regarded as permuting the columns of $uf_n(\mathcal{A})$. Let $\mathbf{P}_n$ denote such permutation matrix. We have

$$\mathbf{U}\cdot uf_n(\mathcal{A}) = \mathbf{U}\cdot uf_n(\mathcal{A})\cdot\mathbf{P}_n^T\mathbf{P}_n = \mathbf{U}\cdot uf_n'(\mathcal{A})\cdot\mathbf{P}_n, \tag{12}$$

where we use the identity $\mathbf{P}_n^{-1} = \mathbf{P}_n^T$, and $uf_n'(\mathcal{A})$ represents the false mode-n unfolded matrix of $\mathcal{A}$. Eq. (12) implies that the same result of the mode-n product can be obtained by permuting the columns of $\mathbf{U}\cdot uf_n'(\mathcal{A})$ back in the order as those of $uf_n(\mathcal{A})$ and then applying true mode-n refolding. In fact, this operation is equivalent to false mode-n refolding.

ACKNOWLEDGEMENT

This work was supported in part by the Ministry of Science and Technology of Taiwan under Grant Numbers NSC101-2221-E-155-065, MOST103-2221-E-155-031, and MOST106-2221-E-155-058.

REFERENCES

1. R. Ballester-Ripoll, S. K. Suter, and R. B. Pajarola, "Analysis of tensor approximation for compression-domain volume visualization," *Computers and Graphics*, Vol. 47, 2015, pp. 34-47.
2. M. Balsa Rodríguez, E. Gobbetti, J. A. Iglesias Guitián, M. Makhinya, F. Marton, R. B. Pajarola, and S. K. Suter, "State-of-the-art in compressed GPU-based direct volume rendering," *Computer Graphics Forum*, Vol. 33, 2014, pp. 77-100.
3. Y. Bando, H. Holtzman, and R. Raskar, "Near-invariant blur for depth and 2D motion via time-varying light field analysis," *ACM Transactions on Graphics*, Vol. 32, 2013, pp. 13:1-13:15.
4. H. Cao, H. Jin, X. Wu, S. Wu, and X. Shi, "DAGMap: Efficient and dependable scheduling of DAG workflow job in grid," *The Journal of Supercomputing*, Vol. 51, 2010, pp. 201-223.
5. S. Ceri, C. Gennaro, S. Paraboschi, and G. Serazzi, "Effective scheduling of detached rules in active databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, 2003, pp. 2-13.
6. K. J. Dana, B. Van Ginneken, S. K. Nayar, and J. J. Koenderink, "Reflectance and texture of real-world Surfaces," *ACM Transactions on Graphics*, Vol. 18, 1999, pp. 1-34.
7. L. de Lathauwer, B. de Moor, and J. Vandewalle, "On the best rank-1 and rank-(R_1, R_2, …, R_n) approximation of higher-order tensors," *SIAM Journal on Matrix Analysis and Applications*, Vol. 21, 2000, pp. 1324-1342.
8. M. Eriksson and C. Kessler, "Integrated code generation for loops," *ACM Transactions on Embedded Computing Systems*, Vol. 11S, 2012, pp. 19:1-19:24.
9. K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of Graphics Hardware*, 2004, pp. 133-137.
10. J. Filip and M. Haindl, "Bidirectional texture function modeling: A state of the art survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 31, 2009, pp. 1921-1940.
11. K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, Vol. 34, 2008, pp. 12:1-12:25.
12. J. Gu, C. I. Tu, R. Ramamoorthi, P. N. Belhumeur, W. Matusik, and S. K. Nayar, "Time-varying surface appearance: Acquisition, modeling and rendering," *ACM Transactions on Graphics*, Vol. 25, 2006, pp. 762-771.
13. T. Günther and T. Grosch, "Distributed out-of-core stochastic progressive photon mapping," in *Proceedings of Computer Graphics Forum*, Vol. 33, 2014, pp. 154-166.

14. A. Haidar, M. Gates, S. Z. Tomov, and J. J. Dongarra, "Toward a scalable multi-GPU eigensolver via compute-intensive kernels and efficient communication," in *Proceedings of the 27th ACM International Conference on International Conference on supercomputing*, 2013, pp. 223-232.
15. M. Kim and K. S. Candan, "Efficient static and dynamic in-database tensor decompositions on chunk-based array stores," in *Proceedings of ACM International Conference on Information and Knowledge Management*, 2014, pp. 969-978.
16. T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, Vol. 51, 2009, pp. 455-500.
17. M. L. Koudelka, S. Magda, P. N. Belhumeur, and D. J. Kriegman, "Acquisition, compression, and synthesis of bidirectional texture functions," in *Proceedings of the 3rd International Workshop on Texture Analysis and Synthesis*, 2003, pp. 59-64.
18. J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics*, Vol. 22, 2003, pp. 908-916.
19. S. Lahabar and P. J. Narayanan, "Singular value decomposition on GPU using CUDA," in *Proceedings of IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1-10.
20. M. Marqués, G. Quintana-Ortí, E. S. Quintana-Ortí, and R. A. van de Geijn, "Using graphics processors to accelerate the solution of out-of-core linear systems," in *Proceedings of International Symposium on Parallel Distributed Processing*, 2009, pp. 168-176.
21. A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of EuroSys*, 2010, pp. 153-166.
22. J. R. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, Vol. 30, 2010, pp. 56-69.
23. J. D. Owens, D. P. Luebke, N. K. Govindaraju, M. J. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, Vol. 26, 2007, pp. 80-113.
24. R. B. Pajarola, S. K. Suter, and R. Ruiters, "Tensor approximation in visualization and computer graphics," in *Proceedings of Eurographics – Tutorials*, 2013, T6.
25. A. Pajot, L. Barthe, M. Paulin, and P. Poulin, "Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering," *Computer Graphics Forum*, Vol. 30, 2011, pp. 315-324.
26. J. Pantaleoni, L. Fascione, M. Hill, and T. Aila, "PantaRay: Fast ray-traced occlusion caching of massive scenes," *ACM Transactions on Graphics*, Vol. 29, 2010, pp. 37:1-37:10.
27. G. Quintana-Ortí, F. D. Igual, M. Marqués, E. S. Quintana-Ortí, and R. A. van de Geijn, "A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures," *ACM Transactions on Mathematical Software*, Vol. 38, 2012, pp. 25:1-25:25.
28. R. Ramamoorthi, "Precomputation-based rendering," *Foundations and Trends in Computer Graphics and Vision*, Vol. 3, 2009, pp. 281-369.
29. H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemptions," *ACM Transactions on Embedded Computing Systems*, Vol. 10, 2010, pp. 27:1-27:34.
30. N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis, "Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition," in

Proceedings of Asilomar Conference on Signals, Systems, and Computers, 2014, pp. 581-585.
31. R. Ruiters and R. Klein, "BTF compression via sparse tensor decomposition," in *Proceedings of Computer Graphics Forum*, Vol. 28, 2009, pp. 1181-1188.
32. M. Rump, R. Sarlette, and R. Klein, "Groundtruth data for multispectral bidirectional texture functions," in *Proceedings of International Conference on Computer, Graphics, Imaging, and Visualization*, 2010, pp. 326-331.
33. M. Sattler, R. Sarlette, and R. Klein, "Efficient and realistic visualization of cloth," in *Proceedings of Eurographics Workshop on Rendering*, 2003, pp. 167-178.
34. S. K. Suter, J. A. Iglesias Guitián, F. Marton, M. Agus, A. Elsener, C. P. E. Zollikofer, M. Gopi, E. Gobbetti, and R. B. Pajarola, "Interactive multiscale tensor reconstruction for multiresolution volume visualization," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 17, 2011, pp. 2135-2143.
35. S. K. Suter, M. Makhynia, and R. B. Pajarola, "TAMRESH – Tensor approximation multiresolution hierarchy for interactive volume visualization," in *Proceedings of Computer Graphics Forum*, Vol. 32, 2013, pp. 151-160.
36. Y. T. Tsai, "Multiway K-clustered tensor approximation: Toward high-performance photorealistic data-driven rendering," *ACM Transactions on Graphics*, Vol. 34, 2015, pp. 157:1-157:15.
37. Y. T. Tsai, K. L. Fang, W. C. Lin, and Z. C. Shih, "Modeling bidirectional texture functions with multivariate spherical radial basis functions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 33, 2011, pp. 1356-1369.
38. Y. T. Tsai and Z. C. Shih, "All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation," *ACM Transactions on Graphics*, Vol. 25, 2006, pp. 967-976.
39. Y. T. Tsai and Z. C. Shih, "K-clustered tensor approximation: A sparse multilinear model for real-time rendering," *ACM Transactions on Graphics*, Vol. 31, 2012, pp. 19:1-19:17.
40. M. A. O. Vasilescu and D. Terzopoulos, "TensorTextures: Multilinear image-based rendering," *ACM Transactions on Graphics*, Vol. 23, 2004, pp. 336-342.
41. J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Computing Surveys*, Vol. 33, 2001, pp. 209-271.
42. V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of ACM/IEEE Conference on Supercomputing*, 2008, pp. 1-11.
43. H. Wang, Q. Wu, L. Shi, Y. Yu, and N. Ahuja, "Out-of-core tensor approximation of multidimensional matrices of visual data," *ACM Transactions on Graphics*, Vol. 24, 2005, pp. 527-535.
44. R. Wang, Y. Huo, Y. Yuan, K. Zhou, W. Hua, and H. Bao, "GPU-based out-of-core many-lights rendering," *ACM Transactions on Graphics*, Vol. 32, 2013, pp. 210:1-210:10.
45. Q. Wu, T. Xia, C. Chen, H. Y. S. Lin, H. Wang, and Y. Yu, "Hierarchical tensor approximation of multidimensional visual data," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 14, 2008, pp. 186-199.
46. A. C. Zhou and B. He, "Transformation-based monetary cost optimizations for workflows in the cloud," *IEEE Transactions on Cloud Computing*, Vol. 2, 2014, pp. 85-98.

47. B. Zou, C. Li, L. Tan, and H. Chen, "GPUTENSOR: Efficient tensor factorization for context-aware recommendations," *Information Sciences*, Vol. 299, 2014, pp. 159-177.

Yu-Ting Tsai (蔡侑庭) received the B.S. and M.S. degrees in Electronics Engineering from National Chiao Tung University, Taiwan, in 2000 and 2002, respectively, and the Ph.D. degree in Computer Science from National Chiao Tung University, Taiwan, in 2009. Currently, he is an Assistant Professor in the Department of Computer Science and Engineering at Yuan Ze University. His research interests include computer graphics, computer vision, machine learning, and signal processing.

Wei-Jhih Wang (王暐智) received the B.S. degree in Computer Science from National Tsing Hua University, Taiwan. Currently, he is a Master student in the Department of Computer Science and Engineering at Yuan Ze University. His research interests include computer graphics and interactive games.

Tzu-Yuan Kao (高梓淵) received the B.S. degree in Computer Science and Engineering from Yuan Ze University, Taiwan. His research interests include computer graphics and interactive games.