We present ATC, a C++ library for advanced Tucker-based lossy compression of dense multidimensional numerical data in a shared-memory parallel setting, based on the sequentially truncated higher-order singular value decomposition (ST-HOSVD) and bit plane truncation. Several techniques are proposed to improve speed, memory usage, error control and compression rate. First, a hybrid truncation scheme is described which combines Tucker rank truncation and TTHRESH quantization. We derive a novel expression to approximate the error of truncated Tucker decompositions in the case of core and factor perturbations. We parallelize the quantization and encoding scheme and adjust this phase to improve error control. Implementation aspects are described, such as an ST-HOSVD procedure using only a single transpose. We also discuss several usability features of ATC, including the presence of multiple interfaces, extensive data type support, and integrated downsampling of the decompressed data. Numerical results show that ATC maintains state-of-the-art Tucker compression rates while providing average speed-up factors of 2.2 to 3.5 and halving memory usage. Our compressor provides precise error control, deviating only 1.4% from the requested error on average. Finally, ATC often achieves higher compression than non-Tucker-based compressors in the high-error domain.

CCS Concepts: • Theory of computation → Data compression; • Mathematics of computing → Mathematical software performance; • Computing methodologies → Linear algebra algorithms; • Human-centered computing → Scientific visualization; • Applied computing → Physical sciences and engineering;

Additional Key Words and Phrases: Data compression, tensors, Tucker decomposition, ST-HOSVD, bit plane truncation

ACM Reference format:
Wouter Baert and Nick Vannieuwenhoven. 2023. Algorithm 1036: ATC, An Advanced Tucker Compression Library for Multidimensional Data.
ACM Trans. Math. Softw. 49, 2, Article 21 (June 2023), 25 pages.
https://doi.org/10.1145/3585514

1 INTRODUCTION
Many scientific, industrial, and medical applications generate numerical data on multidimensional grids, such as hyperspectral imaging [3], diffusion tensor imaging [28], X-ray scans [27], and simulations of various kinds [9, 13, 23]. As these datasets increase in size, so does the need for compression to reduce network and storage costs. Specifically, when dealing with floating-point data, lossy
Fig. 1. A sample of spectral slices from the Moffett-Field hyperspectral image (indices are 0 based). For visibility, a different scaling factor was used for each visualization, yet all slices clearly exhibit the same structure, showing great potential for compression.

Fig. 2. ATC compression examples using the Isotropic-PT dataset. Each visualization shows only the first time slice of the data tensor, whereas the statistics in the captions represent the full data.

compression is most applicable because there is often no need to store the data in full precision. In fact, the original data may already be subject to measurement, simulation, and round-off errors of a certain magnitude, making storage beyond this level of accuracy redundant.

However, simply storing each data value within this limited precision is usually not the best approach. As shown in Figure 1, realistic datasets often contain a lot of redundancy, which can be removed during compression. When handling smooth datasets such as the one presented in Figure 2, high compression factors can often be achieved at a small cost in compression error.

In this article, we propose the Advanced Tucker Compressor (ATC), a lossy compression library for multidimensional numerical data, based on the sequentially truncated higher-order singular value decomposition (ST-HOSVD) [49] and an existing bit-plane-truncation-based quantization and encoding scheme from the TTHRESH compressor [5]. Our motivation is twofold. First, we aim to improve speed, memory usage, error control and compression rate where possible compared with existing Tucker-based [48] methods, such as TTHRESH [5] and TuckerMPI [4].

ACM Transactions on Mathematical Software, Vol. 49, No. 2, Article 21. Publication date: June 2023.
Table 1. General Performance Comparison in between Tucker-based Compressors over a Range of Compression Errors and Datasets

| Compressor         | Rel. comp. | Speed-up | Peak memory (bytes/element) | Error control |
|--------------------|------------|----------|-----------------------------|---------------|
|                    |            | Comp.    | Decompl.                    |               |
| TTHRESH            | -          | -        | -                           | -             |
| Baseline ATC       | +0.8%      | +27%     | -0.8%                       | -             |
| + Rank truncation  | -8.2%      | +54%     | +33%                        | +0.0%         |
| + Parallel quant./encoding | -0.4% | +77% | +61%                     | +1.4%         |
| + Predict dequant. correction | +4.0% | -3.1% | +1.5%                   | -0.1%         |
| + Householder compression | +1.2% | +5.7% | +1.9%                  | -0.1%         |
| + Mode storage order heuristic | +1.1% | -0.2% | +2.5%                  | +0.9%         |
| + Split bit plane truncation | +3.0% | -1.4% | -1.8%                 | +0.0%         |
| Full ATC           | +1.1%      | +249%    | +121%                       | +35%          |
| TuckerMPI          | -90%       | +308%    | +135%                       | +155%         |

Error control is expressed as the average relative deviation of the actual compression error from the target error. Each boldface compressor shows relative performance compared to TTHRESH; the other lines show incremental performance gains (green) or losses (red) achieved by a certain compressor optimization compared with the last line. For memory usage and error control, we also show the absolute performance metrics. Note that these metrics are averaged over a wide variety of settings, with certain optimizations having very different effects depending on the situation. The full methodology used is described in Section 6.

are interested in the compression of general, dense tensors, such as grid-based physical measurements or simulation data. Second, our objective is to create a Tucker-based compression library with state-of-the-art performance and usability features, such as multiple interfaces, broad support for data types, shared-memory parallelism and both file-based and in-memory compression or decompression.

In our view, these broad features are not yet available in existing tensor-based compressors. For example, TTHRESH lacks most of these features as well as a programming interface; therefore, it can be used only as an executable instead of a library. On the other hand, while TuckerMPI supports distributed-memory parallelism, it is solely aimed at computing Tucker decompositions. As such, it achieves much worse compression rates due to a lack of quantization and encoding methods (see Table 1).

Table 1 summarizes the performance gains achieved by the ATC optimizations compared with the other Tucker-based compressors TTHRESH and TuckerMPI. Thanks to an efficient implementation, for example, avoiding unnecessary copies, baseline ATC already achieves a roughly 27% compression speed-up and roughly halves memory usage compared with TTHRESH while being quite similar. Furthermore, the various algorithmic improvements made by ATC accumulate into an average compression and decompression speed-up factor of 3.5 and 2.2, respectively, as well as 24 times more precise error control, while maintaining similar compression rates. Moreover, the user can make certain trade-offs, for example, increasing compression while lowering speed by reducing the level of rank truncation. In summary, this shows the significant improvements made by ATC over existing work, both algorithmically and in terms of implementation.

In this article, we will discuss the following aspects. In Section 2, we describe several general compression techniques and alternative compressors. In Section 3, we present preliminary material that will be used throughout the article. In Section 4, we elaborate on our main algorithmic contributions. In Section 5, we summarize specific implementation aspects of the software. In Section 6, numerical results are presented and discussed, followed by concluding remarks and potential future lines of research in Section 7. In the supplementary materials, we discuss additional minor improvements and features.
2 RELATED WORK

Over the past few decades, a variety of methods were invented to address compression needs in diverse application domains [41]. In this section, we discuss the most relevant related methods.

2.1 Compression Basics

In data compression, methods are often divided into two categories: lossless compression, which compresses data while ensuring that it can be reconstructed exactly, and lossy compression, which introduces a small compression error in order to achieve much higher compression rates or compression factors, defined here as \( \frac{\text{original data size}}{\text{compressed data size}} \). Since our article is concerned with lossy compression, we will often express this error using one of the following metrics:

- The relative error (RE) \( \frac{\|A - \tilde{A}\|}{\|A\|} \), where \( \| \cdot \| \) denotes the Euclidean norm (see Section 3.1)
- The sum-of-squared-errors (SSE) \( \|A - \tilde{A}\|^2 \).

When processing numerical data, the floating-point format complicates efficient lossy compression. For example, the sign bit or exponent bits of a floating-point number will have a much larger influence on the represented number than the least significant bits of the mantissa. Therefore, quantization is required, in which each number is approximated within some discrete domain that is easier to represent in a compressed format. These quantized values can then be encoded, that is, stored as a bit stream, using lossless compression techniques such as the ones described in Section 2.2.

2.2 Lossless Compression

When no error on the original data can be tolerated, it can be compressed using lossless compression. Many techniques for this purpose were proposed [41], resulting in general lossless compressors such as zlib [20] and Zstandard [12]. Next, we briefly describe two techniques that are relevant to our own work. Lossless compression can also be used as a component of a lossy compression pipeline.

Run-length encoding. When storing a sequence of identical symbols, it is more frugal to represent it as a tuple containing this symbol and the length of the sequence, that is, its run-length. For example, this technique is used in JPEG (Joint Photographic Experts Group) [8] image compression to compress sequences of zero coefficients.

Entropy coding. In information theory, the entropy of an information source represented by a discrete probability distribution \( X \) consisting of the probabilities \( p_1, \ldots, p_n \) is defined as \( H(X) = -\sum_i p_i \log_2 p_i \) [43]. Furthermore, Shannon’s source coding theorem implies that, when encoding a stream of symbols identically and independently distributed according to \( X \), one needs at least \( H(X) \) bits per symbol on average [43]. While this sets an upper limit to compression efficiency within this setting, it does not provide us with a constructive algorithm to approach this bound.

One approach to entropy coding consists of representing each symbol as a particular sequence of bits, its code word, and concatenating all code words corresponding to the symbols from the original stream. To allow for unambiguous decoding, the code must be a prefix code, that is, each code word of length \( l \) must be different from the first \( l \) bits of any other code word. Huffman coding [25] provides a simple algorithm to produce an optimal prefix code tree and has become widely used since its invention [41]. However, all prefix-code-based methods suffer from the constraint that each code word must consist of an integer number of bits. In contrast, the entropy of a particular symbol, \( -\log_2 p_i \), is almost always non-integer. Therefore, this discrepancy decreases the compression efficiency of such methods.
Arithmetic coding [40] solves this issue by representing all symbols in the stream combined as an interval bounded by two long fractions. The compressed output then consists of the binary representation of an arbitrary number within this interval. Since certain bits in this sequence contain information on more than one symbol, the aforementioned limitation of prefix codes is avoided. In fact, when the number of encoded symbols goes to infinity, the theoretical compression efficiency approaches the entropy limit [40].

More recently, asymmetric numeral systems [15] were proposed as a faster alternative to arithmetic coding while maintaining optimal theoretical compression efficiency. Because of this speed advantage, asymmetric numeral systems have become widely used in state-of-the-art compressors [12, 21, 26].

2.3 Lossy Compression

When the original data do not need to be reconstructed exactly, lossy compression can be used. This is especially relevant for data that is intended for human interpretation, such as images, sound, and video. Because of this, many lossy compressors choose which information to discard based on a model of human perception [8, 35]. Alternative error metrics, such as the structural similarity index measure (SSIM) [51] and video multmethod assessment fusion (VMAF) [30], were proposed to address this. Nevertheless, in this work, we will employ the usual Euclidean (or Frobenius) norm as our error metric because it is simple and has useful properties with regard to multilinear algebra.

A first class of lossy compressors can be described as predictive in the sense that values are predicted using, for example, neighboring values or a fitting function [16]. If the predictor performs well, the deviation of each value from its prediction should be small, which makes them easier to encode than the original values. Before encoding, these deviations are quantized in an appropriate way to achieve the desired level of accuracy. For our application, the compression of numerical multidimensional arrays, predictive-based compressors include SZ [31, 53, 54] and FPZIP [33].

Some compressors apply a certain invertible transformation to the data before quantization. By exploiting the structured nature of the most relevant data, these transformations can significantly increase the sparsity of the coefficients in the transformed domain, improving compressibility. The discrete cosine transform (DCT) [1] is a widely used tool for this purpose [10], being used in various image and video compressors such as JPEG, x265 [35] and AV1 [2]. For our application, transforms are used in compressors such as ZFP [32], TTHRESH [5], and SSEM [42].

Transform-based methods usually rely on fixed transformations, which removes the need to store the transform but might lead to less efficient compression than if a data-dependent transform were used. For example, when compressing an \( n \times n \) matrix \( A \) with singular value decomposition (SVD) \( A = USV^T \), it can be interesting to use its bases of left and right singular vectors \( U \) and \( V \) as data-dependent transforms applied to the column and row spaces, respectively, since the Eckart-Young theorem [17] states that this provides us with the best low-rank approximation in terms of the Euclidean norm. However, even when truncating to rank \( r \), the compressed representation would require \( 2nr + r \) scalars to store the largest \( r \) singular values and the \( r \) corresponding columns of \( U \) and \( V \) compared with \( r^2 \) when using the same truncation rank for a fixed transform, such as a 2-dimensional DCT, due to transform storage cost.

In the case of data with 3 or more modes, the orthogonal Tucker decomposition [48] reduces the relative storage overhead of the transformation by representing an \( n_1 \times n_2 \times \cdots \times n_d \) tensor as a core tensor of the same size and orthogonal matrices \( U_i \in \mathbb{R}^{n_i \times n_i} \). Like a truncated SVD, we can truncate the matrices \( U_i \) to the first \( r_i \) columns and retain only the \( r_1 \times r_2 \times \cdots \times r_d \) core tensor corresponding to this selection of basis vectors. This reduces the storage cost to \( r_1 r_2 \cdots r_d \) scalars for the transformed coefficients and \( \sum_{i=1}^d n_i r_i \) scalars for the factor matrices. As such, the number of transformed coefficients grows exponentially in the number of modes \( d \), while the
transform storage cost only increases linearly with \( d \). Due to this relatively low transform storage cost, Tucker decomposition can still be competitive compared with methods using fixed transforms. In the field of tensor decompositions, this decomposition is also commonly used for compression of tensors of moderate order \([29, 38]\). For example, the TuckerMPI software efficiently computes Tucker decompositions of massive datasets in a distributed-memory parallel setting.

Only a few publications discuss the quantization and encoding of such decompositions \([5, 6, 45, 46]\). To our knowledge, the Tucker-based TTHRESH compressor employs the most advanced quantization scheme among these, using bit plane truncation, which will be discussed further in Section 3.3. We will use a variant of this strategy in ATC.

Tensor train \([37]\) and hierarchical Tucker \([22]\) decompositions are also used for compression of high-order tensors \([38]\). In our targeted application we are dealing with dense tensors, that is, tensors that store each element explicitly. Thus, the order of tensor is typically not high. Therefore, these alternative decompositions will not be discussed further.

Finally, a new category of low-rank tensor formats based on subpartitioning is also emerging \([18, 34]\). Many data compressors already split the input data into subblocks before compression \([8, 31, 32, 52]\), not only to reduce the complexity of most transform-based methods but also to improve compression when adaptively partitioning the data into blocks to diminish intra-block discontinuities. While it is promising to apply this approach to tensor formats, this is out of scope for the current version of ATC.

3 PRELIMINARIES

3.1 Tensor Concepts and Notation

A tensor \( \mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) is represented in bases by a multidimensional array containing numerical values indexed with \( d \) integers. \( n_1 \times \cdots \times n_d \) is called the size of the tensor, and \( d \) is called the order. The Euclidean inner product of two tensors \( \langle \mathcal{X}, \mathcal{Y} \rangle \) is the sum over their element-wise products. The induced Euclidean norm is \( ||\mathcal{X}|| = \sqrt{\langle \mathcal{X}, \mathcal{X} \rangle} \).

We will use the MATLAB notation \( a : b \) as a tensor index to select a subrange along the corresponding mode from index \( a \) up to and including index \( b \), or simply \( i \) when all indices are selected. For example, \( U_{:,1:2} \) refers to the submatrix containing all rows but only the first \( r \) columns of \( U \). A mode-\( k \) fiber of a tensor \( \mathcal{X} \) is then defined as a vector obtained by fixing all indices in the tensor apart from the \( k \)-th index, that is, \( \mathcal{X}_{i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_d} \in \mathbb{R}^{n_k} \). Conversely, a mode-\( k \) slice is a tensor of order \( d - 1 \) obtained by fixing only the \( k \)-th index of the tensor, that is, \( \mathcal{X}_{\cdot,\ldots,i_k,\ldots} \in \mathbb{R}^{n_1 \times \cdots \times n_{k-1} \times n_{k+1} \times \cdots \times n_d} \).

By arranging all mode-\( k \) fibers as columns in a matrix in a consistent manner, we obtain the mode-\( k \) matricization of \( \mathcal{X} \), denoted by \( X_{(k)} \in \mathbb{R}^{n_k \times \prod_{l \neq k} n_l} \). We then define the mode-\( k \) matrix-tensor product as follows:

\[
\mathcal{Y} = U \cdot_k \mathcal{X} \iff Y_{(k)} = UX_{(k)}.
\]

In other words, multiplying a tensor by a matrix \( U \) along mode \( k \) is equivalent to transforming each mode-\( k \) fiber using \( U \). This operation has several useful properties:

- The multilinear product commutes along different modes: \( U \cdot_i U' \cdot_j \mathcal{X} = U' \cdot_j U \cdot_i \mathcal{X} \) for \( i \neq j \).
- Multilinear products along the same mode can be composed: \( U \cdot_i U' \cdot_i \mathcal{X} = (UU') \cdot_i \mathcal{X} \).
- Because multiplying a matrix by an orthogonal matrix preserves its Euclidean norm, multiplying a tensor by an orthogonal matrix along any mode also preserves its Euclidean norm.

3.2 ST-HOSVD Rank Truncation

Using the mode-\( k \) product, we define a Tucker decomposition \([48]\) of \( \mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) as

\[
(U_1, \ldots, U_d) \cdot \mathcal{B} = U_1 \cdot_1 \cdots \cdot_{d-1} U_d \cdot_d \mathcal{B},
\]

ACM Transactions on Mathematical Software, Vol. 49, No. 2, Article 21. Publication date: June 2023.
where \( \mathcal{B} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) is called the core tensor and \((U_1, \ldots, U_d) \in \mathbb{R}^{n_1 \times n_1} \times \cdots \times \mathbb{R}^{n_d \times n_d} \) are called the factor matrices. A common method to compute a Tucker decomposition of a tensor \( \mathcal{A} \) is higher-order singular value decomposition (HOSVD) [14], in which each orthogonal factor matrix \( U_i \) is chosen as the matrix of left singular vectors of \( A_{(i)} \). A truncated HOSVD can then be obtained by retaining only the first \( r_i \leq n_i \) columns, leading to a truncated core \( \tilde{\mathcal{B}} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) and factors \( U_i \in \mathbb{R}^{n_i \times r_i} \). Alternatively, the ST-HOSVD [49] algorithm can also be used, which interleaves the factor computation steps with the rank truncation and projection steps. The full procedure is shown in Algorithm 1. Due to the data reduction in each iteration, this method significantly speeds up as more truncation is applied. The resulting compression error is almost always less than or equal to the one obtained by the truncated HOSVD [49].

For the remainder of this article, we assume that the original data to be compressed is an order-
\( d \) tensor \( \mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) with total size \( N = \prod_{i=1}^{d} n_i \). This tensor will be approximated by the multilinear rank \((r_1, \ldots, r_d)\) ST-HOSVD \((U_1, \ldots, U_d) \cdot \mathcal{B}\), where \( \mathcal{B} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) and the \( U_i \in \mathbb{R}^{n_i \times r_i} \) have orthonormal columns. The final approximation \( \tilde{\mathcal{A}} \) produced by the proposed ATC compressor will be denoted by \((\tilde{U}_1, \ldots, \tilde{U}_d) \cdot \tilde{\mathcal{B}}\), where \( \tilde{\mathcal{B}} \in \mathbb{R}^{n_1 \times \cdots \times n_d} \) and \( \tilde{U}_i \in \mathbb{R}^{n_i \times r_i} \).

### 3.3 TTHRESH Bit Plane Truncation and Encoding

While compression can be achieved through the aforementioned concept of rank truncation, another strategy is to compute the full HOSVD \( \mathcal{A} = (U_1, \ldots, U_d) \cdot \mathcal{B} \) and then store the resulting coefficients with limited precision. This is the essence of the bit plane truncation scheme employed by the Tucker-based compressor TTHRESH [5], which then encodes the remaining data using a custom procedure described later. Table 2 demonstrates this process with an example in which \( \mathcal{B} \in \mathbb{R}^{2 \times 2 \times 2} \).

First, all entries of \( \mathcal{B} \) are scaled by \( 2^k \), where \( k \in \mathbb{N} \) such that \( 2^{63} \leq 2^k \| \mathcal{B} \|_{\text{max}} < 2^{64} \), where \( \| \cdot \|_{\text{max}} \) denotes the max-norm: the largest absolute value of the entries in the tensor. This scaling

| Sign | Absolute value (binary) |
|------|-------------------------|
| \( b_{111} \) | 0 1 0 0 0 0 0 0 1 1 1 ... |
| \( b_{112} \) | 0 0 0 0 0 0 0 0 0 1 1 ... |
| \( b_{121} \) | 0 0 0 0 0 0 0 0 0 0 0 ... |
| \( b_{122} \) | 1 0 0 0 1 1 0 0 1 0 1 ... |
| \( b_{211} \) | 0 0 0 0 0 0 0 0 1 0 0 ... |
| \( b_{212} \) | 1 0 0 0 1 0 0 0 1 1 0 ... |
| \( b_{221} \) | 0 0 0 0 1 0 0 1 1 0 0 ... |
| \( b_{222} \) | 0 0 0 0 0 1 0 1 0 0 1 ... |

**ALGORITHM 1:** The ST-HOSVD [49]

**Data:** input tensor \( \mathcal{A} \) of order \( d \)

**Result:** truncated core \( \mathcal{B} \), truncated factors \( \tilde{U}_1, \ldots, \tilde{U}_d \)

1. \( \mathcal{B} = \mathcal{A} \);
2. for \( i = 1, \ldots, d \) do
   3. [Compute a singular value decomposition \( \tilde{B}_{(i)} = U_i \Sigma_i V_i^T \);]
   4. [Choose truncation rank \( r_i \);]
   5. \( \tilde{U}_i = U_{i,1:r_i} \);
   6. \( \tilde{B}_{(i)} = \Sigma_{1:r_i,1:r_i} V_{1:r_i,1:r_i}^T \);
factor is stored in the compressed output so that the process can be inverted during decompression. Then, the core coefficients of $2^k B$ are rounded to the nearest integer. By vectorizing the core coefficients into a column vector of length $\Pi_i r_i$ and considering the binary representation of each coefficient’s absolute value, we obtain a bit matrix in which each row represents a single coefficient. The first few columns of this matrix are shown in Table 2. We then iterate over each column of this matrix, that is, each bit plane, starting from the left, in order of significance. Within each bit plane, we process each bit one by one and track the core quantization error that would be achieved by encoding all bits up till this point. Due to the orthogonality of the Tucker factors, ignoring perturbations introduced during factor compression, this core error will equal the final compression error. Therefore, when the target compression error is reached, the procedure ends. We define the point in the bit matrix where this happens as the **breakpoint** of the bit plane truncation process. In Table 2, the breakpoint is at position 4 on the 7th-highest bit plane. We define all coefficients with at least one encoded 1-bit as **significant**, with all other coefficients being **insignificant**. In Table 2, coefficients $b_{112}$, $b_{121}$, and $b_{211}$ are insignificant, whereas all others are significant.

All bits in the vectorized core up to the breakpoint are included in the encoded core. Not all bits are encoded in the same way. The **leading bits** of each coefficient, marked in italics in Table 2, mostly consist of zeroes. Therefore, run-length encoding is applied to all sequences of zeroes within each bit plane, that is, within each column of the bit matrix. Because the resulting run lengths are highly non-uniformly distributed [5], they are then compressed further using arithmetic coding. The **sign bits** of the significant coefficients as well as the **trailing bits** (marked in bold) are almost uniformly distributed in general; thus, they are encoded without compression.

Note that all insignificant coefficients will be decoded as zero; thus, their signs do not need to be stored. It is possible that after bit plane truncation, certain core slices contain only insignificant coefficients, which are represented by 0 in the quantized core. Therefore, the corresponding factor columns will not be encoded. For further details, such as the quantization and encoding of the factors, we refer the reader to [5].

Ballester-Ripoll and Pajarola [6] concluded that a variant of the aforementioned thresholding scheme consistently leads to better compression than Tucker rank truncation followed by a simple quantization scheme. Rank truncation indiscriminately removes all coefficients from low-energy slices, whereas bit plane truncation preserves the most significant components from all slices. We will use a variation of this approach in ATC.

4 THE ATC PIPELINE

In this section we describe the ATC pipeline, which is summarized in Figure 3. This diagram serves as a point of reference throughout this section. While the decompression pipeline also has a few distinctive components, it is very similar to the compression procedure in reverse. Therefore, it is not documented in a separate diagram.

In the first phase of the compression process, the data are processed by an ST-HOSVD. We describe the motivation for ST-HOSVD rank truncation and the implications for error control when combined with bit plane truncation in Section 4.1. During the second phase, the resulting core tensor and factors are quantized and encoded. Although this is based on TTHRESH’s bit plane truncation scheme, we applied several improvements, two of which will be discussed in Sections 4.2 and 4.3. Various minor improvements are discussed in the supplementary material.

4.1 Hybrid Truncation

As discussed in Section 3, previous research suggests that while the ST-HOSVD can compute truncated Tucker decompositions relatively quickly, TTHRESH’s bit plane truncation approach

ACM Transactions on Mathematical Software, Vol. 49, No. 2, Article 21. Publication date: June 2023.
achieves superior compression. Combining both strategies was suggested by [5, 6] as a trade-off between compression rate and execution time.

In our approach, we first apply rank truncation during the ST-HOSVD (see Algorithm 1) to approximate a certain target rank truncation error. Then, the truncated core and factors are processed during the quantization and encoding phase as in TTHRESH, introducing a quantization error as well.

### 4.1.1 ST-HOSVD Mode Processing Order

Due to the truncation performed in each step of the ST-HOSVD, the mode processing order can significantly influence execution time. In ATC, the modes are processed in order of increasing mode length during compression by default, following the heuristic described in [49]. However, ATC also allows the user to specify a processing order. For example, if a particular mode is known to be highly compressible a priori, it is advisable to process it first since this will lead to a large reduction of the core size for the subsequent steps.

During decompression, the mode processing order $p$ is determined by minimizing the approximate number of operations required, estimated as $\sum_{l=1}^{d} n_{p_l} \ldots n_{p_l} r_{p_l} \ldots r_{p_d}$ based on the
The target error for ST-HOSVD is determined by the parameter $p$, as described in Section 8 of [4], we did not implement such an algorithm due to the very low value of $d$.

4.1.2 Circular Mode Shift. To minimize the number of tensor transpositions needed in the ST-HOSVD, we employ a circular mode shift trick, which lets us compute the ST-HOSVD with arbitrary mode processing order using only a single transposition regardless of the order of the tensor. Algorithm 2 describes the full procedure. We start by transposing the input tensor such that the core is initially stored with its modes ordered in the same way as the mode processing order $p$. In the first iteration, we matricize the tensor by “merging” all but the first mode, resulting in an $n_{p_1} \times n_{p_3} \times \ldots \times n_{p_d}$ matrix (line 3). Mode $p_1$ can then be processed. After selecting an appropriate factor matrix, we project the core while transposing it, which results in an $n_{p_2} \times \ldots \times n_{p_d} \times r_{p_1}$ matrix (line 5). The matrix can now be reshaped into a tensor with mode order $(p_2, \ldots, p_d, p_1)$ (line 6). Therefore, each iteration effectively applies a circular shift to the mode order. By initially transposing this mode order to the order $p$, we ensure that at the start of the $i$-th step, mode $p_i$ will be at the front of the mode order, allowing it to be processed.

Note that these reshaping operations simply reinterpret the same memory as matrices with different dimensions and do not result in any data movement or poor memory access patterns, leading to a negligible runtime cost. The Eigen linear algebra software library [24], used in ATC, evaluates the transposition and matrix multiplication on line 5 simultaneously using expression templates. Thus, effectively, no transposition needs to be processed apart from the initial one on line 1. We implemented a similar procedure for the decompression process as well.

4.1.3 Error Control. The target error for ST-HOSVD is determined by the parameter $\text{rank}_\text{truncation_{max}} \cdot \text{sse}_{\text{share}}$ (RTMSS) as follows:

$$\text{TargetSSE}_{\text{ST-HOSVD}} = \text{RTMSS} \cdot \text{TargetSSE}_{\text{total}}, \quad \text{where} \quad 0 \leq \text{RTMSS} \leq 1.$$ 

Using this target, the truncation rank for each mode is determined dynamically using the strategy described in [49, Section 6.3]. The actual rank truncation SSE can simply be computed as the sum of the squares of the singular values discarded across all steps of the ST-HOSVD, as stated in Theorem 6.4 from [49].

To control the final error of the decompressed tensor, we will now approximately separate this error into multiple components. Based on the ST-HOSVD $A \approx (\overline{U}_1, \ldots, \overline{U}_d) \cdot \overline{B}$, we define the full factors $U_i$ as arbitrary orthogonal matrices such that $(U_i)_{1:1 r_i} = \overline{U}_i$, with the corresponding full core $B = (U_1^T, \ldots, U_d^T) \cdot A$. To simplify arithmetic, we also introduce the padded truncated factors $\overline{U}_i' \in \mathbb{R}^{n_i \times n_i}$ with $(\overline{U}_i')_{1:1 r_i} = \overline{U}_i$ and $(\overline{U}_i')_{r_i + 1:1 n_i} = 0$. We define the padded quantized core $B' \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ as a tensor containing the quantized core coefficients in $B'_{1:1 r_i, \ldots, 1:1 n_d}$ and 0 everywhere else. This tensor will be decomposed as the sum of the full core $B$, the truncation
error tensor $\overrightarrow{B} - B$, and the quantization error tensor $\delta B = \overrightarrow{B} - \overrightarrow{B}$. The quantized factors $\overrightarrow{U}_i \in \mathbb{R}^{n_i \times r_i}$ are matrices with the first $r_i$ columns consisting of the quantized factor coefficients and 0 elsewhere. We define $\delta U_i = \overrightarrow{U}_i - \overrightarrow{U}_i$. Note that due to the non-zero pattern in $\overrightarrow{B}$, this means that $\overrightarrow{U}_i \cdot i \overrightarrow{B} = U_{i \cdot i} \overrightarrow{B}$ for $i = 1, \ldots, d$. Recall that $\overrightarrow{A}$ is the final approximation. We start with the following expression:

$$\|\overrightarrow{A} - A\|^2 = \|(\overrightarrow{U}_1' + \delta U_1, \ldots, \overrightarrow{U}_d' + \delta U_d) \cdot \overrightarrow{B}' - (U_1, \ldots, U_d) \cdot B\|^2.$$  

Discarding higher-order factor quantization errors, we get that

$$\|\overrightarrow{A} - A\|^2 \approx \|\overrightarrow{B}' - B + \sum_{i=1}^{d} (U_i' \delta U_i) \cdot i \overrightarrow{B}'\|^2$$

$$= \|\overrightarrow{B}' - B + \sum_{i=1}^{d} (U_i^T \delta U_i) \cdot i \overrightarrow{B}'\|^2$$

$$= \|\overrightarrow{B}' - B\|^2 + \|\delta B\|^2 + \sum_{i=1}^{d} \|((U_i^T \delta U_i) \cdot i \overrightarrow{B}', (U_j^T \delta U_j) \cdot j \overrightarrow{B}')\|$$

where we used that the truncation and quantization error tensors ($\overrightarrow{B} - B$) and $\delta B$ have complimentary non-zero patterns; thus, their inner product is zero. Using the Cauchy-Schwartz inequality, we can provide an upper bound:

$$\|\overrightarrow{A} - A\|^2 \leq \|\overrightarrow{B}' - B\|^2 + \|\delta B\|^2 + \sum_{i=1}^{d} \|\delta U_i \cdot i \overrightarrow{B}'\|^2 + 2 \sum_{i=1}^{d} \sum_{j=1}^{i-1} \|((U_i^T \delta U_i) \cdot i \overrightarrow{B}', (U_j^T \delta U_j) \cdot j \overrightarrow{B}').$$

However, for the general, dense tensors in our setting, this bound is very loose in practice. This can be explained by considering that due to the mostly uniform quantization errors, the error tensor components are not aligned in a particular direction. Because these tensors live in a very high-dimensional space, they are almost orthogonal to each other. Thus, the inner products in Equation (1) are dominated by the squared norms in practice.

As a result, we propose to ignore the inner products from Equation (1) to obtain a more useful estimate for the compression error. Note that the core tensor $C$ produced by an HOSVD is all-orthogonal, that is, all mode-$i$ slices are orthogonal to each other for each $i$ [14]. This implies that $\|\delta U_i \cdot i \overrightarrow{C}\| = \|\delta U_i \Sigma_i\|_F$, where $\Sigma_i$ is a diagonal matrix containing the mode-$i$ core slice norms of $C$. Although this property does not hold exactly for $\overrightarrow{B}$ due to truncation and quantization errors, we observed that it is mostly all orthogonal in practice. Therefore, we approximate $\|\delta U_i \cdot i \overrightarrow{B}'\|$ by $\|\delta U_i \Sigma_i\|_F$. This leads to the following error approximation:

$$\|\overrightarrow{A} - A\|^2 \approx \|\overrightarrow{B}' - B\|^2 + \|\delta B\|^2 + \sum_{i=1}^{d} \|\delta U_i \Sigma_i\|_F^2.$$  

(2)

When compressing several datasets from Section 6.1 with various RTMSS values and target relative errors $10^{-2}$ and $10^{-3}$, we found that the relative difference in between the left-hand and
right-hand side of Equation (2) never exceeded 0.15%. At a target relative error of $10^{-1}$, this deviation increased to 8.5%. We conclude that the approximation is usually very accurate; thus, we will use it to predict the total error. Finally, as in THRESH, we do not consider the factor errors while quantizing the core because they are not yet known. Therefore, we simply set the target core quantization SSE to $\text{TargetSSE}_{\text{total}} - \text{ActualSSE}_{\text{ST-HOSVD}}$.

Figure 4 shows the effect of the RTMSS parameter on the compression factor, with 0 and 1 corresponding to almost no and almost only rank truncation, respectively. Surprisingly, some level of hybrid truncation improves the compression rate in certain cases in addition to improving the execution time, although the optimal RTMSS parameter depends on the dataset and compression error. This improvement can be attributed to a very large reduction in encoded leading zeroes, which generally also decreases their cost after run length and entropy coding. We chose 0.5 as the default RTMSS parameter value, which we believe to be a reasonable trade-off between compression efficiency and speed.

4.2 Parallel Core Quantization and Encoding
Since the core quantization and encoding process represents a significant share of the total compression time, we adapted it to support multi-threading. This is achieved by splitting the vectorized core into a series of equally sized blocks, with each block containing a contiguous series of quantized core coefficients. The number of blocks is at least as large as the number of threads. Each thread can then process and encode the bits of the current bit plane inside one or more blocks independently of the other blocks. This procedure is executed sequentially for all bit planes until the encoding breakpoint is reached.

A minor complication is dynamically selecting a breakpoint for approximating the desired quantization error in parallel. To avoid synchronization overhead, we process bit planes in full without synchronization and check the total error reduction achieved at the end of the bit plane. If this exceeds the target error reduction, we process the bit plane again with a limited number of threads and synchronized error checks. When one thread determines that the target error has been reached, all threads will stop encoding bits, leading to a separate breakpoint for each thread. Therefore, the full bit plane will consist of several alternating sections of encoded and non-encoded bits. Note that only the last encoded bit plane needs to be processed twice, as the global target error reduction can be exceeded only once.

For each block, the bits of the current bit plane are then encoded independently and written to temporary buffers in memory. These compressed blocks are then sequentially written to the
compressed output, including the size of each block. Although these sizes could be determined during decompression, explicitly storing them allows ATC to quickly read and separate all blocks into temporary buffers during decompression before parsing them in parallel, leading to parallelized dequantization and decoding as well.

### 4.3 Improving Error Control by Predicting the Dequantization Correction

During dequantization, if a decoded coefficient \( \tilde{a} \) contains bits down to bit plane \( p \), we know that the original value \( a \) was located in the interval \([\tilde{a}, \tilde{a} + 2^p - 1]\) (ignoring signs). To decrease the quantization error, we therefore approximate \( a \) as \( \tilde{a} + 2^p - 1 \). While this correction was already applied in TTHRESH, we significantly improved error control by considering this during the quantization error tracking process as well.

To demonstrate the significance of this correction, assume that \( a \) is drawn from the uniform distribution \( U[\tilde{a}, \tilde{a} + 2^p] \). Then, we have that

\[
E_{a \sim U[\tilde{a}, \tilde{a} + 2^p]} [(a - \tilde{a})^2] = \frac{2^{2p}}{3} \quad \text{and} \quad E_{a \sim U[\tilde{a}, \tilde{a} + 2^p]} [(a - \tilde{a} - 2^{p-1})^2] = \frac{2^{2p}}{12}.
\]

Therefore, if we guess that \( a \) is (approximately) in the middle of the uncertainty interval rather than at its lower edge, we reduce the SSE by a factor of 4 when \( p \) is large. While the values \( a \) are not exactly uniformly distributed in practice, this nevertheless shows that dequantization correction has a significant impact on the final error and, therefore, needs to be considered during quantization to achieve precise error control.

### 5 SOFTWARE IMPLEMENTATION

ATC is implemented in C++17 and is mainly designed to optimize compression efficiency, speed, error control and memory usage. Certain mathematical software aspects, such as the usability of the library, were also taken into account. In this section, we discuss the handling of cut-outs, the interfaces of the library, shared-memory parallelism, libraries, support for various types, I/O, and general software design considerations. When referring to library version numbers throughout this section, we will specify the lowest tested version.

**Cut-outs and downsampling.** In some settings, only a part of the decompressed tensor may be needed. In such situations, Tucker decomposition can be efficiently combined with the extraction of a cut-out from the decompressed tensor. Although any linear filter could be used to downscale the granularity of the tensor grid, we simply decided to reuse the downsampling, box, and Lanczos filtering methods from TTHRESH, as described in Section 5 of [5]. ATC is implemented in such a way that new filters could be added relatively easily if necessary.

**Interfaces.** ATC’s native interface is written in C++. However, due to the use of the Standard Template Library (STL) containers, this interface may cause compatibility problems when the library and user code are not compiled in exactly the same setting. To address this, we provide a C wrapper, which also serves as an interface to C users as well as users who may want to connect ATC to other languages, such as Python. Finally, we implemented an executable that also offers a command-line interface.

**Shared-memory parallelism.** The user can specify the threads parameter to determine how many threads can be used for linear algebra and tensor transpositions, as these steps are performed using external libraries that support shared-memory parallelism. The core quantization and encoding process is parallelized using OpenMP 4.5 [36], as discussed in Section 4.2, and can be controlled using the same parameter.
Libraries used. Tensor transpositions are performed using the High-Performance Tensor Transpose (HPTT) library version 1.0.5 [44]. This library supports arbitrary mode permutations, allowing ATC to use a custom mode processing order during the ST-HOSVD, as well as a custom mode storage order for quantization and encoding. We observed that HPTT appreciably sped up these parts of the compression pipeline. Some code from TTHRESH [5] was reused with permission from the author to implement certain shared features, such as the handling of cut-outs.

Linear algebra is performed using Eigen 3.3.4, with an option to use Basic Linear Algebra Subprograms (BLAS) [7] for matrix multiplications. To enable an optional higher-order DCT preprocessing phase, the Fastest Fourier Transform in the West (FFTW) 3 library [19] is needed. If the user wishes to compile the ATC command-line interface, the Boost.Program_options library (version 2) [39] is also required. Finally, to enable multi-threading in some parts of the pipeline as discussed in the previous paragraph, OpenMP 4.5 [36] is required.

Supported types and I/O. ATC supports a wide range of data types for the input tensor: 8-, 16-, 32- and 64-bit integers, signed and unsigned, as well as 32- and 64-bit floating-point numbers. The tensor can be provided either as a file or as a buffer in memory. Similarly, both the compressed and decompressed output can be stored using either method. The original and decompressed tensors are stored in a flattened, binary format. Metadata such as the data type and mode sizes are passed as separate arguments during compression and are stored in the compressed file for use during decompression. If the original data file contains a header, this can be ignored using the optional skip_bytes parameter.

Internally, floating-point arithmetic is performed using 64-bit precision by default, although the user can switch to 32-bit precision if desired. We provide a similar option for choosing in between 32- and 64-bit integer types for quantization and encoding. Our experiments indicate that in some high-error cases, lower precision suffices to achieve roughly the same compression factor while significantly improving speed.

Software design. To support the various types described in earlier, ATC uses C++ templates throughout most of its code. While this keeps the implementation concise and improves modifiability, templated code must be written in header files by default so that the compiler can instantiate the code only for the requested types when compiling the user code. Therefore, the library templated code needs to be recompiled by the user whenever the application is modified, which can lead to long compilation times and can slow down development. Since ATC supports only a fixed set of types, we addressed this issue by explicitly instantiating the relevant code for each of the valid types for most type parameters.

However, instantiating all code in terms of each type parameter combination would lead to excessively large compiled binaries. ATC solves this problem by templating most code only in terms of the floating-point type parameter while using inheritance in some cases to connect components for which types are known only at runtime. Although this adds the performance overhead of dynamic dispatch, ATC is specifically designed to invoke this mechanism only in non-critical parts of the code, thus, making the performance cost negligible.

6 RESULTS

In this section, we will empirically evaluate the performance of ATC in terms of compression rate, speed, memory usage, and error control across datasets of different sizes from various application domains. These metrics will be compared with other state-of-the-art numerical data compressors.
Table 3. The Datasets Used Along with Their Properties

| Dataset          | Size   | Mode sizes          | Data type   | Domain                      | Modes                          |
|------------------|--------|---------------------|-------------|-----------------------------|--------------------------------|
| Foot [27]        | 16.0 MB| 256 × 256 × 256     | uint8       | X-ray scan                  | Space (3)                      |
| Brain [28]       | 103 MB | 160 × 190 × 148 × 6 | float32     | Diffusion tensor image       | Space (3) × component          |
| Moffett-Field [3] | 224 MB | 512 × 1024 × 224    | int16       | Hyperspectral image          | Space (2) × wavelength         |
| Isotropic-PT [23]| 800 MB | 100 × 128 × 128 × 128| float32     | CFD simulation               | Time × space (3)               |
| Isotropic-V [23] | 1.50 GB| 512 × 512 × 512 × 3 | float32     | Climate simulation           | Variable × time × space (3)    |
| Deforest-8 [13]  | 3.05 GB| 20 × 79 × 8 × 180 × 360|            |                             |                                |
| Deforest-33 [13] | 12.0 GB| 19 × 79 × 33 × 180 × 360|            |                             |                                |
| Hurricane [9]    | 24.2 GB| 13 × 20 × 100 × 300 × 300|            |                             |                                |

“Space (X)” represents a set of X spatial modes. “Components” represents the unique diffusion tensor elements per voxel in the case of Brain and the velocity components per voxel in the case of Isotropic-V.

6.1 Datasets

We selected a diverse collection of datasets across a range of sizes and application domains, as listed in Table 3. Some of these were already used as benchmarks in other publications [5, 31]. Certain datasets were preprocessed as follows:

- **Brain**: The original dataset consists of 7 values per voxel: the 6 unique elements of the corresponding diffusion tensor along with a “confidence” value in the interval [0, 1], indicating the accuracy of the measurements for this voxel. We extracted the 6 actual components and multiplied them by the corresponding confidence value. This effectively removes the distorted values, which represent missing elements, by mapping them to 0, which makes them easy to compress while improving smoothness by not applying a hard thresholding scheme.

- **Isotropic-PT, Isotropic-V**: These are cut-outs of the “Isotropic 1024 Fine” dataset [23]. We extracted the subtensors from the corner with the smallest coordinates in the full tensor. In the case of Isotropic-V, the time mode effectively has size 1.

- **Deforest-8, Deforest-33**: These are part of the “deforest-globe” dataset (variant r1i1p1f1) [13]. To acquire an appropriate amount of data, we considered only .gr data files with version 20191122 and annually sampled variables. The filtered data files were then merged into two separate tensors based on their number of levels, that is, the size of the vertical spatial mode, with 20 variables using 8 levels and 19 variables using 33 levels. The original data contains extreme but constant values in voxels located over land. Due to a lack of documentation, we assumed that these values represent missing values and should not be stored. Therefore, we subtracted the first time slice from all other time slices. The resulting tensor represents the change in each variable since the start of the simulation, setting these missing values to zero and reducing their effect on the compression process. Finally, we normalized each variable by scaling it such that its maximum absolute value equals 1.

- **Hurricane**: To not exceed the available amount of memory on our hardware, we select only the first 20 time slices. Since the missing values in this dataset were all set to a particular constant, we simply set all of these values to zero and did not apply the difference coding step, unlike the previous case. Each variable slice was then normalized in the manner discussed earlier.

6.2 Compressors

For our experimental comparison, we selected several lossy data compressors based on a recent survey by Duwe et al. [16, Section 1.2.2]. We limited ourselves to compressors with at least support for datasets of order 3 and 4 as well as a publicly available implementation, including a command-line interface, since all experiments are performed using this interface. The resulting compressors are listed in Table 4. Considering the rate-distortion comparisons performed in [5] and [31], this
Table 4. The Compressors and Their Corresponding Information Used in the Following Compression Comparison

| Compressor | Version/commit date | Cores | Notes |
|------------|---------------------|-------|-------|
| ATC        | 1.1.2               | 36    | Using OpenBLAS 0.2.20 for matrix multiplications. |
| TTHRESH    | 20 Feb. 2022        | 36    | No support for datasets with 5 or more modes (through the command-line interface). |
| SZ         | 12 Apr. 2022        | 36    | No support for datasets with 5 or more modes (through the command-line interface). |
| ZFP        | 27 Jan. 2022        | 36 (compr.) 1 (decompr.) | The tensor is sliced into 36 parts along the longest mode. Each part is handled by a single-threaded MPI process. Since the data are stored in single precision, we use the single-precision driver. Using OpenMPI 2.1.2 with shared-memory optimization flags --mca btl self,sm,tcp. |
| FPZIP      | 1.3.0               | 1     | Based on the x265 video codec [35], see details in text. |
| TuckerMPI  | 31 Jan. 2022        | 36    | The tensor is sliced into 36 parts along the longest mode. Each part is handled by a single-threaded MPI process. Since the data are stored in single precision, we use the single-precision driver. Using OpenMPI 2.1.2 with shared-memory optimization flags --mca btl self,sm,tcp. |
| x265       | 2.6                 | 16    | Based on the x265 video codec [35], see details in text. |

The last compressor in this list, x265, consists of a Python script that preprocesses the dataset and passes it onto the x265 video codec for compression using Fast Forward Moving Picture Experts Group (FFmpeg) version 4.1 and compiled with GCC 6.4.0 [47]. We used the options -preset veryslow -tune psnr, leading to high compression in terms of relative error but longer runtimes. Furthermore, the error was controlled using the parameter crf to optimize compression results. Although we are not aware of any literature considering video compressors for compressing general tensor data, we include x265 in our comparison due to its high performance on video data [11, 50].

Because this compressor is intended for video formats, we added preprocessing steps to support compression of general tensors of order 3 or higher. First, for each dataset, we designated two of its spatial modes as the video’s width and height mode and one mode as the video’s time mode. This time mode was chosen as the dataset’s own time mode or some other very smooth mode. All other remaining modes were then merged with the time mode by transposing and reshaping the tensor. Finally, all values were shifted, scaled, and rounded to fit into the domain \{0, 1, 2, \ldots, 255\}. Note that due to this limited domain, there will always be a significant quantization error; Thus, our x265-based compressor cannot produce low compression errors and will be relevant only in the high-error domain.

Finally, while we included FPZIP in all of our experiments, we found that its compression performance was almost always worse than ZFP while being much slower. To reduce the number of data points in the figures in this section, we will not show results from this compressor.

6.3 Experimental Set-up

All experiments were performed on a computing node with two Xeon Gold 6240 central processing units (2.6 GHz clock speed with 18 physical cores and 36 threads per CPU) and 180 GB main memory running CentOS 7.9.2009. I/O was performed using local solid-state drives (SSDs) with theoretical reading and writing speeds of 1 GB/s and 0.6 GB/s, respectively. We let each compressor use the maximal number of supported threads or processes, described in Table 4.

Wall-clock times and peak memory usage were measured using the Linux command /usr/bin/time. Due to the memory limitations of TTHRESH (see Section 6.6), we will not apply this compressor to the largest dataset, Hurricane. Furthermore, x265 results will not be shown for Deforest-8 and Deforest-33 since the corresponding compression errors are all larger than 10%.
Throughout this section, we will aggregate average gains in terms of compression rates, speed, memory usage and error control when comparing compressors. These averages refer to geometric means in the case of relative compression gains, speed-ups, and peak memory reduction, whereas in the case of error control, we use the arithmetic mean of the absolute values of the logarithms of the relative error deviations, as shown in the following formula:

$$\text{Average relative error deviation} = \exp\left(\frac{\sum_{i=1}^{n} |\ln\frac{\text{ActualError}_i}{\text{TargetError}_i}|}{n}\right)$$

Because these metrics are not all evaluated at the same error, we will instead compute the average in between interpolated curves. We obtain these curves by applying linear interpolation to data points consisting of the logarithm of the compression error as well as the peak memory usage or the logarithm of the compression factor, execution time, or deviation from the target error depending on the metric under consideration. Figures 5 and 7 to 9 show that by considering these metrics on a logarithmic scale, we obtain relatively smooth curves. We will use this interpolation method when plotting relative compression factors.

Note that some compressors do not accept all data types. Therefore, for datasets with integral data types, we cast them to a 32-bit floating-point format before compression. After decompression, we round the decompressed values back to the nearest value in the domain of the corresponding integer type, which can significantly reduce the compression error in low-error settings. Note that while ATC already performs this rounding step for integral data types internally, we perform it externally in all cases to prevent this simple feature from skewing our experimental results. Moreover, SZ, ZFP, and FPZIP do not support compression of order-5 tensors (through the command-line interface). For these datasets, we merge the first two modes and treat them as order-4 tensors.

6.4 Incremental Performance Gains Compared with TTHRESH

Table 1 shows how each modification to the baseline TTHRESH algorithm affects various performance metrics in our experiments. To clarify the impact of these changes, we briefly discuss our most important findings here:

- **Baseline ATC**: Due to a more efficient implementation, we achieve a compression speed-up of around 27% while halving memory usage.
- **Hybrid truncation**: This leads to compression and decompression speed-up factors of roughly 1.5 and 1.3, respectively, at the cost of an 8% decrease in compression rate. Note that these numbers are averages and can vary a lot in practice, depending on the compressibility of the dataset and the target error.
- **Parallel quantization and encoding**: This results in compression and decompression speed-up factors of roughly 1.8 and 1.6, respectively, while slightly reducing error control. Our experiments show that these speed-up factors are generally present but can become much smaller for well-compressible datasets and high target errors since in these cases the truncated core shrinks, reducing the time share of the quantization and encoding phase in the full compression process.
- **Predicting the dequantization correction during quantization**: This small change greatly improves error control, reducing the average error deviation from 27.7% to 1.4%, with little effect on other performance metrics.
- **Various minor improvements**: The three last modifications described in Table 1 slightly improve average compression rate, but in some cases at a minor speed cost. As such, their
relevance may depend on the use case. Therefore, while they are enabled by default, we provide user options to disable them.

6.5 Rate-Distortion Comparison

Figure 5 shows the rate-distortion curves for all compressors and datasets, with the highest curves representing the best compression rates. We observed that ATC and TTHRESH perform very well in some cases (Moffett-Field, Isotropic-PT, and Isotropic-V), adequate in other cases (Deforest-8, Deforest-33, and Hurricane) and poorly in some other cases (Foot and Brain). In particular, we note that these compressors outperform the rest in the high-compression domain, typically starting from a compression factor of 10 to 100. This usually corresponds with high errors, with ATC/TTHRESH often achieving the highest relative compression rates at errors over 1%. In fact,
Fig. 6. Compression factors of ATC, in terms of its RTMSS parameter value, relative to TTHRESH. Since we do not have TTHRESH data on Hurricane, we choose ATC with RTMSS = 0.4 as a reference instead, which still shows the relationship between the RTMSS parameter value and the compression factor.

while SZ slightly outperforms ATC for many errors, ATC’s gain is so high in other cases that on average, in our experiments, ATC achieves 98% higher compression.

When comparing ATC and TTHRESH, we can see that both perform similarly most of the time, with ATC achieving only 1.1% higher compression on average in our experiments. Figure 6 demonstrates that this can be attributed to ATC’s default RTMSS parameter value of 0.5, which controls rank truncation (see Section 4.1). If we consider the highest compression factor achieved by ATC across the different RTMSS settings for each error value, we obtain an average gain of 8.0% compared with TTHRESH at the cost of compression and decompression slowdowns of 24% and 21%, respectively, relative to the default RTMSS setting, which is still much faster than TTHRESH. For some datasets, this increases to 20% or higher for high errors, which Figure 5 shows is the most relevant domain for Tucker-based compressors. Note that this gain is accumulated over the optimal RTMSS setting for each individual error, which is often but not always 0. Furthermore, we observe that for most errors, decreasing the RTMSS parameter improves compression. In conclusion, we find that users who prioritize compression over speed might want to start with an RTMSS value of 0 and adjust it if necessary because this setting almost never leads to compression losses and can lead to significant gains compared with TTHRESH.

6.6 Time and Memory Usage

Figure 7 shows the compression and decompression times of each compressor. Note that the Tucker-based compressors generally need more time than the others because they apply a global transformation to the data (the Tucker decomposition), whereas most other compressors process
the data in blocks in some way, leading to a time complexity proportional to the amount of data. This discrepancy also manifests itself in Figure 8 in terms of peak memory usage.

However, when comparing ATC to TTHRESH, our compressor took 71% and 55% less time on average during compression and decompression, respectively. Yet, we observe that the Tucker decomposition incurs a significant speed cost, making the non-Tucker-based compressors SZ and ZFP much faster in most settings.

Finally, Figure 8 shows that ATC is more efficient in terms of memory usage than its Tucker-based counterpart TTHRESH, on average achieving reductions of 54% and 56% during compression and decompression, respectively. Yet, note that these averages are skewed by the memory overhead observed in the cases of small datasets such as Foot and Brain, whereas ATC achieves average memory usage of around 18 bytes per data element for larger datasets in contrast to TTHRESH’s typical 40 bytes per element. TuckerMPI uses even less memory since we are using its single-precision driver, whereas ATC stores the tensor in double precision by default.
Fig. 8. Peak memory usage of each compressor during compression and decompression. Since this is largely proportional to the number of elements in the original data, we express this metric in terms of the peak number of bytes used per element. To measure TuckerMPI’s peak memory usage, we run it using only one process. Because this process needs to read all data alone, it cannot process the largest three datasets due to the limitations of MPI I/O.

6.7 Error Control

Lossy compressors can control the trade-off between compression rate and error in various ways. In the case of ATC and certain other compressors in our comparison, we say that they are error bounded since they attempt to approximate a given target error with the highest possible compression rate. When using such a compressor, it can be important that the actual compression error does not deviate much from the desired target error. Higher errors are undesirable whereas lower errors lead to an unnecessary loss of compression rate. As a result, we analyze the degree of error control of different compressors in Figure 9. ATC clearly performs very well, with an average relative deviation from the target error of 1.4% compared with TTHRESH’s 33.8% and SZ’s 31.7%.
Fig. 9. Error control per compressor, defined as the deviation of the compression error from the target error. Values below 1 indicate that the compressor achieved a lower error than requested. ZFP, FPZIP, and our x265 compressor do not support any Euclidean-norm-based target error metric.

7 CONCLUSIONS

We presented ATC, a novel Tucker-based numerical data compressor centered around the ST-HOSVD and bit plane truncation. Several techniques were described to improve speed, memory usage, error control and compression rate. Certain implementation and usability aspects were discussed as well.

Our experiments show that ATC on average maintains the compression rates of the state-of-the-art Tucker-based compressor TTHRESH while providing average speed-up factors of 3.5 and 2.2 during compression and decompression, respectively. Average peak memory usage was also reduced in our experiments by 54% and 56%, respectively. Moreover, ATC achieves very precise error control, on average only deviating 1.4% from the requested compression error.

Compared with non-Tucker-based compressors, ATC usually outperforms all alternatives in terms of rate distortion when targeting high relative errors, for example, above 1%. Although the state-of-the-art SZ compressor achieves slightly higher compression rates in many settings, ATC drastically outperforms it in other situations, leading to an average compression gain for ATC of 97% in our experiments. However, due to the costly Tucker decomposition, ATC uses significantly more time and memory.

ACKNOWLEDGMENTS

We thank the five anonymous reviewers for their extensive questions and remarks on earlier versions of this article, which greatly improved it. We are grateful to Rafael Ballester-Ripoll for allowing us to reuse certain parts of the TTHRESH source code in the implementation of ATC. We kindly thank Zitong Li (TuckerMPI) and Kai Zhao (SZ) for fixing our reported issues, allowing us to run our experiments using their compressors.
We also thank the sources of all datasets used in this paper: Philips Research and IAPR-TC18 (Foot), Gordon Kindlmann and Andrew Alexander (Brain), the NASA Jet Propulsion Laboratory (Moffett-Field), the Johns Hopkins Turbulence Databases (Isotropic-PT and Isotropic-V), the World Climate Research Programme and the Earth System Grid Federation (Deforest-8 and Deforest-33) as well as the NCAR and U.S. National Science Foundation (Hurricane).

Nick Vannieuwenhoven was partially supported by a Postdoctoral Fellowship of the Research Foundation—Flanders (FWO) with project 12E8119N.

REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao. 1974. Discrete cosine transform. IEEE Transactions on Computers C-23, 1 (1974), 90–93. https://doi.org/10.1109/T-C.1974.223784

[2] Alliance for Open Media. [n.d.]. AV1 Features. Retrieved March 12, 2023 from https://aomedia.org/av1-features/.

[3] AVIRIS. [n.d.]. Free Data. Retrieved March 12, 2023 from https://aviris.jpl.nasa.gov/data/free_data.html.

[4] G. Ballard, A. Klinvex, and T. G. Kolda. 2020. TuckerMPI: A parallel C++/MPI software package for large-scale data compression via the Tucker tensor decomposition. ACM Transactions on Mathematical Software 46, 2, Article 13 (Jun 2020), 31 pages. https://doi.org/10.1145/3378445

[5] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. 2020. TTHRESH: Tensor compression for multidimensional visual data. IEEE Transactions on Visualization and Computer Graphics 26, 9 (2020), 2891–2903. https://doi.org/10.1109/TVCG.2019.2904063

[6] R. Ballester-Ripoll and R. Pajarola. 2015. Lossy volume compression using Tucker truncation and thresholding. The Visual Computer (2015), 1–14.

[7] S. L. Blackford, A. Petitet, R. Pozo, K. Remington, C. R. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software 28, 2 (2002), 135–151.

[8] CCITT. 1992. Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines. Retrieved March 12, 2023 from https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[9] U.S. National Center for Atmospheric Research. [n.d.]. Hurricane Isabel WRF Model Data Visualization. Retrieved March 12, 2023 from https://www.earthsystemgrid.org/dataset/isabeldata.html.

[10] W. Chen. 2004. The Electrical Engineering Handbook. Elsevier, 906.

[11] J. De Cock, A. Mavlankar, A. Moorthy, and A. Aaron. 2016. A large-scale video codec comparison of x264, x265 and libvpx for practical VOD applications. In Applications of Digital Image Processing XXXIX, Andrew G. Tescher (Ed.), Vol. 9971. International Society for Optics and Photonics, SPIE, 363–379. https://doi.org/10.1117/12.2238495

[12] Y. Collet and C. Turner. 2016. Smaller and faster data compression with Zstandard. Retrieved March 12, 2023 from https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/.

[13] Gokhan Danabasoglu. 2019. NCAR CESM2 model output prepared for CMIP6 LUMIP deforest-globe. Earth System Grid Federation. https://doi.org/10.22033/ESGF/CMIP6.7574

[14] L. DeLathauwer, B. De Moor, and J. Vandewalle. 2000. A multilinear singular value decomposition. SIAM Journal on Matrix Analysis and Applications 21, 4 (2000), 1253–1278. https://doi.org/10.1137/S0895479896305696

[15] J. Duda. 2009. Asymmetric numeral systems. (2009). arXiv:0902.0271

[16] K. Duwe, J. Lüttgau, G. Mania, J. Squar, A. Fuchs, M. Kuhn, E. Betke, and T. Ludwig. 2020. State of the art and future trends in data reduction for high-performance computing. Supercomputing Frontiers and Innovations 7, 1 (2020). https://superfri.org/superfri/article/view/303.

[17] C. Eckart and G. Young. 1936. The approximation of one matrix by another of lower rank. Psychometrika 1, 3 (1936), 211–218.

[18] V. Ehrlicher, L. Grigori, D. Lombardi, and H. Song. 2021. Adaptive hierarchical subtensor partitioning for tensor compression. SIAM Journal on Scientific Computing 43, 1 (2021), A139–A163. https://doi.org/10.1137/19M128689X arXiv:https://doi.org/10.1137/19M128689X

[19] M. Frigo and S. Johnson. 2005. The design and implementation of FFTW3. Proceedings of the IEEE 93, 2 (2005), 216–231. Special issue on “Program Generation, Optimization, and Platform Adaptation.”

[20] J. Gailly and M. Adler. [n.d.]. zlib. Retrieved March 12, 2023 from http://zlib.net/.

[21] Google. [n.d.]. Draco 3D Graphics Compression. Retrieved March 12, 2023 from https://google.github.io/draco/.

[22] L. Grasedyck. 2010. Hierarchical singular value decomposition of tensors. SIAM Journal on Matrix Analysis and Applications 31, 4 (2010), 2029–2054. https://doi.org/10.1137/090764189 arXiv:https://doi.org/10.1137/090764189

[23] Turbulence Database Group. [n.d.]. Johns Hopkins Turbulence Database. Retrieved March 12, 2023 from http://turbulence.pha.jhu.edu/newcutout.aspx.
[52] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC video coding standard. 
IEEE Transactions on Circuits and Systems for Video Technology 13, 7 (2003), 560–576. https://doi.org/10.1109/TCSVT.2003.815165

[53] K. Zhao, S. Di, M. Dmitriev, T. D. Tonellot, Z. Chen, and F. Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In IEEE 37th International Conference on Data Engineering (ICDE’21). 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[54] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello. 2020. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization. In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (Stockholm, Sweden) (HPDC’20). ACM, New York, NY, 89–100. https://doi.org/10.1145/3369583.3392688

Received 6 May 2022; revised 17 November 2022; accepted 14 December 2022