ATC: an Advanced Tucker-based Compression library for multidimensional data – User manual

Wouter Baert

ATC is a compression library for numerical data on grids of 3 dimensions or higher, aimed at optimizing compression rates, compression/decompression time, memory usage and error control. It mainly relies on the Tucker decomposition, which exploits correlations along each mode (dimension) but takes some time to compute. Furthermore, ATC controls the error of the resulting tensor in terms of the Euclidean/Frobenius norm, in contrast to some other compressors which enforce an elementwise error bound or use other error metrics.

For an elaborate description of the algorithms used in ATC, as well as various implementation aspects, please consult the our paper.

ATC was developed by Wouter Baert as part of a research project under supervision of Nick Vannieuwenhoven at KU Leuven, Belgium.

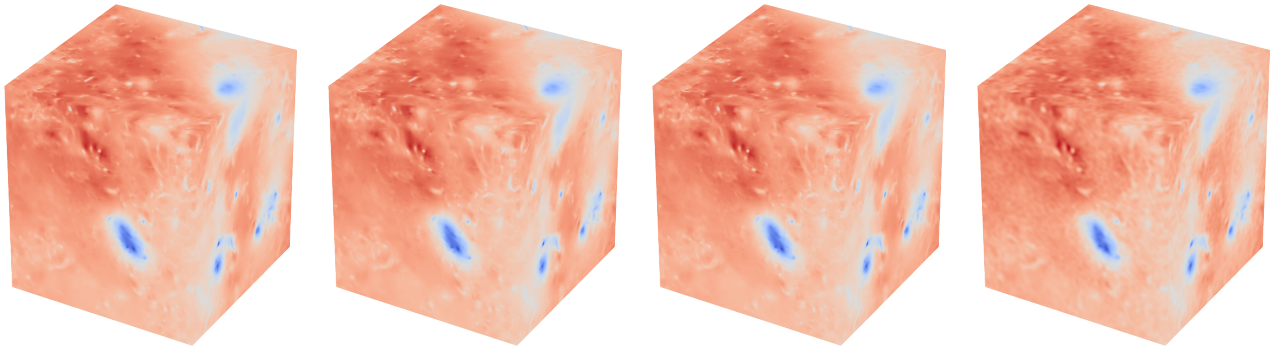
Main features

- Interfaces for C++ (using STL containers), C and command-line
- Error control in terms of the Frobenius norm of the error tensor (represented as relative error, RMSE or PSNR)
- Multi-threading support for linear algebra, DCT's, tensor transpositions as well as quantization and encoding
- Support for efficient decompression of specific subsets of the original data, using cut-outs and optionally downsampling
- Support for compression/decompression to/from files (C++/C/CLI) or in-memory (only C++/C)
- Support for 10 input data types (`uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float32`, `float64`)
- Support for single and double precision for floating-point arithmetic

Main algorithmic components

- ST-HOSVD-based truncated Tucker decomposition
- Parallellized quantization and encoding based on split bit plane truncation
- Precise error control using dequantization correction prediction

Compression examples



Original	Compression factor: 153 Relative error: 1.00% PSNR: 63.9 dB	Compression factor: 639 Relative error: 3.00% PSNR: 54.3 dB	Compression factor: 4948 Relative error: 10.1% PSNR: 43.7 dB

Dataset: A 1x512x512x512 cut-out, selecting the subtensor with the lowest coordinates, of the "Isotropic 1024 Fine" dataset from the Johns Hopkins Turbulence Database [1].

Usage

Requirements

ATC is written in C++ using features from C++17 and lower, so first of all a compatible compiler is required (e.g. GCC version 7 or higher). CMake is used to generate the Make build system. Furthermore, the following software must be installed on the system in a place where CMake will find them:

- Eigen
- Optional, to enable BLAS [2] routines: a BLAS implementation (only used when using `-DATC_-BLAS=ON`, more information below)
- Optional, to enable multi-threading: OpenMP [3]
- Optional, to compile the command-line interface executable: Boost.Program_Options
- Optional, to enable T-HODCT preprocessing: FFTW [4]

The following libraries are also used in ATC but are included already and do not require any further set-up from the user:

- HPTT [5]
- Parts of TTHRESH [6] (with permission from the author)

Installation

You can perform an out-of-source build by executing the following commands in the root directory:

```
mkdir build
cd build
cmake ..
make cli -j4
```

This will build the library itself as well as the command-line interface executable. If one of the aforementioned libraries is installed on your system, but the CMake configuration process does not find them (as shown in the CMake output), you can add a macro definition to the `cmake ..` command in order to point CMake towards the installation directory. Specifically, it should point to a directory containing

the relevant CMake files, such as `<package>Config.cmake` or `<package>-config.cmake`. For example, if Eigen is not found, you might need to add `-DEigen3_DIR=<path>/eigen3/cmake/`, because this is where `Eigen3Config.cmake` is located. Note that for the optional libraries, if CMake doesn't find them, it will not print a message as to how to help it locate them. For detailed information, see the relevant CMake documentation.

Furthermore, if you want ATC to use BLAS, you should add the macro definition `-DATC_BLAS=ON` to the `cmake ..` command. In this case, CMake will automatically select a BLAS implementation found on the system, or report an error if none are found. If you want to ensure that a certain vendor is used, you can specify this using `-DBLA_VENDOR=<vendor>`, where `<vendor>` can be `OpenBLAS`, `Intel10_64lp`, `Goto`, ... For more information, see the relevant CMake documentation.

Finally, if you want to install ATC globally on your machine (or at the user level), you may want to add the flag `-DCMAKE_INSTALL_PREFIX=<path>` to the `cmake ..` command. For example, on Unix systems, CMake will set this variable to `/usr/local` by default. Check the target `install` below for more information.

You can also build other targets from the following list:

Target	Description	Result
`atc`	The ATC library itself.	`lib/libatc.so`
`cli`	The command-line interface (requires Boost.Program_Options).	`bin/atc`
`install`	Builds the targets `atc` and `cli` (if valid) and copies the executable, library and public header files (both C and C++) to the location specified during the CMake configuration process.	`${CMAKE_INSTALL_PREFIX}/bin/atc`, `${CMAKE_INSTALL_PREFIX}/lib/libatc.so`, `${CMAKE_INSTALL_PREFIX}/include/atc.hpp`, `${CMAKE_INSTALL_PREFIX}/include/atc.h`, `${CMAKE_INSTALL_PREFIX}/include/atc/*`
`uninstall`	Removes all files generated by `make install` based on the `install_manifest.txt`. Only works on Unix systems and if the `install_manifest.txt` file is still in the build directory.	Removes the results of `make install`
`test-suite`	The C++ test suite.	`bin/test-suite`
`example`	The C++ example driver.	`bin/example`
`example-c`	The C example driver.	`bin/example-c`
`examples`	Both the C++ and C example drivers.	`bin/example` and `bin/example-c`
`doc`	The ATC documentation (requires Python [7], Doxygen, Sphinx and more, see below).	`doc/build/html/index.html`, ...
`readme`	The repository's README (requires Python).	`README.md`, ...
`manual`	The user manual (requires Python, Pandoc and pdflatex).	`user-manual.pdf`

Note that, unfortunately, compiling the HPTT source file `transpose.cpp` can take a long time, leading to full ATC compilation times of 2-4 minutes. Furthermore, we observed that, when upgrading g++ from version 7 to version 9 or 10, compilation slowed down even further to 10-15 minutes. However, since this effect wasn't observed with other compilers, this seems to be a GCC-specific problem. Therefore, if you think this takes too long, we recommend using a different compiler.

Uncompressed/decompressed data format

The uncompressed/decompressed data should/will be stored in the "brick-of-floats" format. This simply means that the tensor is stored in a flattened format, with each scalar being stored using its binary representation in little-endian (e.g. a single-precision floating-point number is stored using 4 bytes), without any header or footer (although in the case of uncompressed data, there can be a header as long as you skip it properly using the `atc::Options.skip_bytes` option). If the `atc::Options.storage_order` is set to `atc::StorageOrder::MSI` (default), then the flattened tensor will be interpreted with the last index varying the fastest (the tensor equivalent of row-major storage). Conversely, if it is set to `atc::StorageOrder::LSI`, the first index will vary the fastest.

For example, consider a 2x2x2 tensor `A` with `A(i, j, k) = 100*i + 10*j + k` using 1-based indexing. When flattened using the MSI storage order, we get the following vector:

```
[111, 112, 121, 122, 211, 212, 221, 222]
```

When flattened using the LSI storage order, we get the following vector:

```
[111, 211, 121, 221, 112, 212, 122, 222]
```

Therefore, in order to achieve good compression, **it is crucial to use the correct storage order or else the tensor will be interpreted in a completely wrong way**. In case you're still unsure after reading this explanation, you can simply try both storage orders once on a data set of your choice. On decent-sized smooth data sets, this will make a drastic difference in terms of compression rate.

Testing the installation

You can test ATC by compiling and executing the example program (`examples/example.cpp`) using differents compilers and hardware. You can do this with a custom compiler by running the following commands in the ATC root directory:

```
rm -rf build
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=<g++|clang|...> ..
make example -j4
```

The example can then be run using the following commands:

```
cd ../bin
./example
```

The output should be similar to:

```
ATC version 1.1.2
Compressed file size: 20709
Compression factor: 3.16461
Target relative error: 0.001
Actual relative error: 0.00101744
```

where the compression factor is the ratio in between the original file size and the compressed file size, while the relative error is the Frobenius norm of the difference in between the original and decompressed tensor, divided by the Frobenius norm of the original tensor. You can also test the C example program (`examples/example.c`) by replacing the above `make example` command with `make examples` and then executing `./example-c` in the `bin` directory. If you want to choose the C compiler, make sure to add the flags `-DCMAKE_C_COMPILER=<gcc|clang|...>` to the aforementioned `cmake` command. Note that, because both targets are configured using the same CMake project, the vendor of both compilers should be the same in this set-up. Of course you should be able to compile your own applications using the C interface with a compiler from a different vendor (the interface is meant to be portable after all), but it would have to be configured separately.

Finally, you can also build and run the target `test-suite` to check if all automated tests execute successfully.

C++ API

You can perform compression and decompression by calling the functions `atc::compress` and `atc::decompress`. These functions are overloaded to cover many different situations. Here we will demonstrate how to use them to compress and decompress a file:

```
atc::compress("path/to/original_data", atc::DataType::float32, {10, 20, 30}, 80,
    "path/to/compressed_data");
```

This will let ATC compress the original data at `path/to/original_data`, a 10x20x30 tensor of 32-bit floating-point numbers stored in a raw binary format, with a target PSNR (i.e. peak signal-to-noise ratio) of 80 dB and store the result in `path/to/compressed_data`. Since the tensor metadata is stored in the compressed file, we can simply decompress it to the file `path/to/decompressed_data` with the following function call:

```
atc::decompress("path/to/compressed_data", "path/to/decompressed_data");
```

For a more extensive example, check out the source file `examples/example.cpp`. For more information, such as the use of `atc::Options`, `atc::Results`, `atc::compress_decompress` or other API functions, please consult the relevant documentation.

C API

As an alternative to the C++ interface, we also provide a C interface. To understand when to use or not to use this interface, check the relevant documentation. The aforementioned compress-and-decompress-files scenario would be translated as follows:

```
// Compression
size_t sizes[3] = {10, 20, 30};
atc_compress_ff("path/to/original_data", ATC_DATA_TYPE_FLOAT32, 3, sizes, 80,
    "path/to/compressed_data"); // "ff" means file-to-file
// Decompression
atc_decompress_ff("path/to/compressed_data", "path/to/decompressed_data");
```

For a more extensive example, check out the source file `examples/example.c`. For more information, please consult the relevant documentation.

Command-line interface

General compression command:

```
./atc --of <original-file> --type <data-type> --sizes <tensor-sizes> --<error-metric>
    <target-error> --cf <compressed-file> [OPTIONS]
```

General decompression command:

```
./atc --cf <compressed-file> --df <decompressed-file> [OPTIONS]
```

You can also combine compression and decompression into a single command, allowing you to use the `--ce` flag to calculate the compression error:

```
./atc --of <original-file> --type <data-type> --sizes <tensor-sizes> --<error-metric>
    <target-error> --cf <compressed-file> --df <decompressed-file> [OPTIONS]
```

Furthermore, you can use INI config files to store and reuse sets of command-line options using the `--config` flag. For examples, check out the Shell script `examples/example.sh`. Some common options you may want to set:

- `--rank-truncation <arg>`: Sets the RTMSS parameter (in the range 0-1), which affects the trade-off in between speed and rate-distortion. Typically, 0 leads to the highest compression rates, while 1 leads to much faster compression (default: 0.5). For details, check out our paper.
- `--threads <arg>`: The maximal amount of threads that will be used. 0 (default) means that ATC will attempt to use the amount of cores available on the system.
- `--verbose`: Prints a lot of information about the compression/decompression process during execution.

For more information on the command-line options, run `./atc --help` or check the relevant documentation.

Python example

Although we do not provide full Python bindings, you can find an example that performs ATC compression and decompression using ctypes at `examples/example.py`, which demonstrates how ATC can be dynamically loaded and called through its C wrapper from other languages. If you want to use ATC in Python, this example is probably a good starting point.

Documentation

You can find the ATC documentation here. Alternatively, you can generate the ATC documentation yourself if you have installed the following dependencies:

- Python
- Doxygen
- Sphinx
- Breathe
- recommonmark
- sphinx-markdown-tables
- sphinx-rtd-theme

by running the following commands:

```
cd build
cmake ..
make doc
```

The documentation can then be viewed by opening the file `doc/build/html/index.html` in a web browser. Note that, due to software installation limitations, this set-up was only tested using Sphinx v1.8.5. For newer Sphinx versions, the Markdown compatibility should probably be handled by MyST instead of recommonmark, but this is not supported at the moment.

To generate the user manual, you can build the target `manual` as long as you have installed the following dependencies:

- Python
- Pandoc
- pdflatex

Benchmarks

In our experiments, we found that with default settings, ATC maintains about the same compression rate as the the Tucker-based TTHRESH compressor, while providing average speed-ups of 3.5 and 2.2 during compression and decompression. Furthermore, peak memory usage is about halved. Moreover, compression can be slightly improved by lowering the `rank_truncation_max_sse_share` parameter at the cost of speed. Finally, ATC achieves very fine error control, on average deviating only 1.4% from the requested compression error. Compared to the Tucker-based TuckerMPI [8] compressor, we achieve about 10 times higher compression due to the lack of quantization and encoding in TuckerMPI.

Compared to non-Tucker-based compressors such as SZ [9], ZFP [10] and FPZIP [11] (as well as our own x265 wrapper), we also observed that ATC often outperforms all alternatives in terms of compression rate in the high-error domain (e.g. 1%-10% relative error). However, due to the expensive Tucker decomposition, ATC takes significantly more time and memory to run compared to e.g. SZ.

For more detailed experimental results, including an extensive series of graphs, we refer to our paper.

Issues

If you find a bug in ATC, please open a new issue or report it to wouter.baert@kuleuven.be.

Acknowledgements

I'd like to thank the authors of TTHRESH, which initially served as the basis of the ATC implementation. Furthermore, ATC's quantization and encoding scheme is heavily based on their work.

Citation

If you use our work in an academic publication, please cite our paper (accepted at the ACM Transactions on Mathematical Software, pre-print available on arXiv) using the following BibTeX reference:

```
@misc{https://doi.org/10.48550/arxiv.2107.01384,
  doi = {10.48550/ARXIV.2107.01384},
  url = {https://arxiv.org/abs/2107.01384},
  author = {Baert, Wouter and Vannieuwenhoven, Nick},
  keywords = {Mathematical Software (cs.MS), FOS: Computer and information sciences,
    FOS: Computer and information sciences},
  title = {ATC: an Advanced Tucker Compression library for multidimensional data},
  publisher = {arXiv},
  year = {2021},
  copyright = {arXiv.org perpetual, non-exclusive license}
}
```

References

[1] E. Perlman, R. Burns, Y. Li and C. Meneveau. "Data Exploration of Turbulence Simulations using a Database Cluster". Supercomputing SC07, ACM, IEEE, 2007. DOI

[2] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel and others. "An updated set of basic linear algebra subprograms (BLAS)". ACM Transactions on Mathematical Software, 28(2), 135–151, 2002. DOI

[3] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46-55, Jan.-March 1998. DOI

[4] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". Proceedings of the IEEE 93 (2), 216–231, 2005. DOI

[5] P. Springer, T. Su and P. Bientinesi. "HPTT: A High-Performance Tensor Transposition C++ Library". Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ACM, 2017. DOI

[6] R. Ballester-Ripoll, P. Lindstrom and R. Pajarola. "TTHRESH: Tensor Compression for Multi-dimensional Visual Data". IEEE Transaction on Visualization and Computer Graphics, IEEE, 2019. DOI

[7] G. van Rossum. "Python tutorial". Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), May 1995.

[8] G. Ballard and A. Klinvex and T. G. Kolda. "TuckerMPI: A Parallel C++/MPI Software Package for Large-scale Data Compression via the Tucker Tensor Decomposition". ACM Transactions on Mathematical Software, 46(2), 2020. DOI

[9] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello. "SZ3: A modular framework for composing prediction-based error-bounded lossy compressors". IEEE Transactions on Big Data. DOI

[10] P. Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". IEEE Transactions on Visualization and Computer Graphics 20, 12, 2674–2683, 2014. DOI

[11] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data". IEEE Transactions on Visualization and Computer Graphics 12, 5, 1245–1250, 2006. DOI