Fast Operations on Compressed Arrays Without Decompression

Harvey Dam Ganesh Gopalakrishnan (advisor)

University of Utah

Abstract

In modern scientific computing and machine learning systems, data movement has overtaken compute as the performance bottleneck, thus motivating the wider adoption of lossy data compression. Unfortunately, state-of-the-art floating-point array compressors such as SZ [1] and ZFP [2] require decompression before operations can be performed on the data. In this work, we show that **compression methods can be designed to allow efficient operations on compressed arrays without having to first decompress**. In particular, compression methods that consist of only linear transformations and quantization allow certain operations on compressed arrays without decompression. We develop such a compression method, called PyBlaz, the first compression method we know that can **compress arbitrary-dimensional arrays** and **directly operate on the compressed representation**, with **all stages running on GPUs**.[a]

Designing PyBlaz for Compressed-Space Operations

Evolution of PyBlaz: We chose Blaz [3], a compression method that supports certain operations on compressed matrices, as a simple starting point, extending it to create PyBlaz. Our extensions consist of (1) adding a data type conversion step to encourage storing floating-point components of compressed arrays in low precision, (2) removing a differentiation step to preserve linearity to facilitate more compressed operations, (3) supporting arbitrary-dimensional arrays to allow wider application, and (4) using the GPU throughout.

Compared to SZ [1] and ZFP [2], PyBlaz supports operations on arrays of arbitrary dimensions while offering higher throughput. While its typical compression ratio of 4 to 8 falls short of that of ZFP and SZ, it exceeds that of lossless compressors, whose compression ratio is 1.5 to 4 [4].

The PyBlaz compression process (Figure 1) was designed to be composed entirely of linear transformations and quantization, with each step implemented for GPUs.

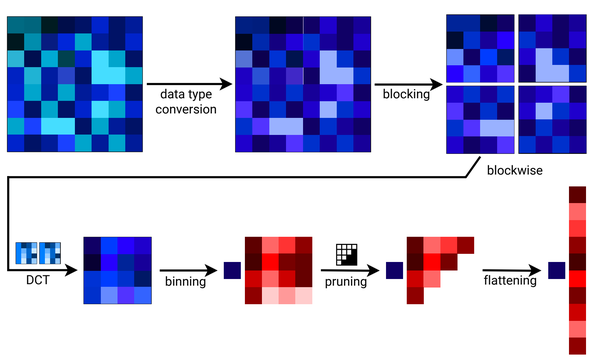

Figure 1. PyBlaz: From scientific data array to compressed-space representation. Blues and greens represent floating-point numbers, reds represent integers, and grays represent Boolean values.

[a] `github.com/damtharvey/pyblaz`

Operations on Compressed Arrays

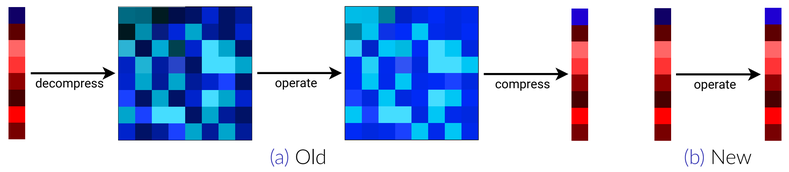

Figure 2. Operations on compressed arrays avoid decompression and recompression.

PyBlaz can perform the following operations in at most logarithmic time with sufficient threads. In most cases, the operation does not induce additional error.

Operation	Time Complexity	Result Type	Source of Error
Negation	$O(1)$	array	none
Element-wise addition	$O(\log n)$	array	rebinning
Addition of a scalar	$O(\log n)$	array	rebinning
Multiplication by a scalar	$O(1)$	array	none
Dot product	$O(\log n)$	scalar	none
Blockwise mean	$O(1)$	array	none
Mean	$O(\log n)$	scalar	none
Covariance	$O(\log n)$	scalar	none
Variance	$O(\log n)$	scalar	none
L_2 norm	$O(\log n)$	scalar	none
Cosine similarity	$O(\log n)$	scalar	none
Structural similarity index	$O(\log n)$	scalar	none

Table 1. Operations on compressed arrays.

Illustration of a Compressed Operation: Mean of an Array

A compressed array representation contains its original shape $\mathbf{s}$, its block shape $\mathbf{i}$, the biggest coefficient for each block N, and the flattened specified bin indices F.

Algorithm 1: Mean

Data: compressed array $\{\mathbf{s}, \mathbf{i}, N, F\}$
Result: the mean of the array
$b \leftarrow$ the number of bits in the element type of F;
$r \leftarrow 2^{b-1} - 1$;
$\hat{C} \leftarrow N \odot F_{\ldots 1} \oslash r$;
return $\text{mean}(\hat{C}) / \prod(\mathbf{i}^{\frac{1}{2}})$;

To get the mean, multiply N blockwise with the corresponding first flattened bin indices, and divide the result by the radius, which half the number of values supported by the bin index type minus one. Then divide the mean of the result by the product of the element-wise square root of the length of a block in each direction.

Time Performance Benchmarks

PyBlaz is faster than the most comparable compression method ZFP [2]. Unlike PyBlaz, ZFP's compression process contains many non-linear transformations, making operations on compressed arrays impractical. The CUDA-accelerated version of ZFP is limited to fixed-rate mode and to 3 dimensions, without the usual features of ZFP for CPU, such as a user-defined error bound.

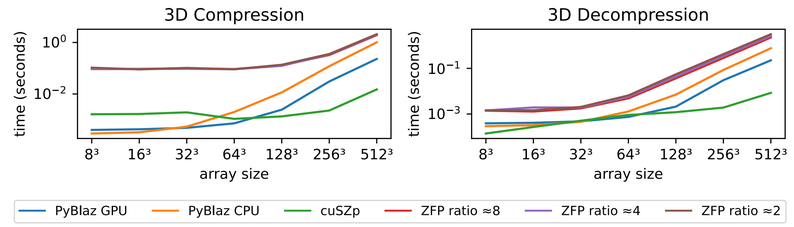

Figure 3. Time to compress or decompress 3-dimensional arrays compared to SZ and ZFP.

Example Application: Diffing Simulation Outputs

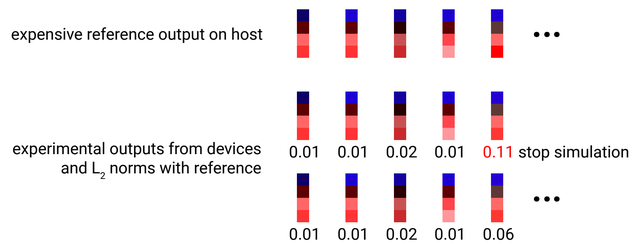

Figure 4. Scientific simulations often trade precision for speed. The compressed outputs of several fast experimental simulations can be compared with that of an expensive reference, stopping simulations that diverge too much. This way, PyBlaz can support efficient ensemble searches while keeping simulation outputs compact, yet still allowing comparison against reference compressed baselines.

Limitations and Future Work

While we have characterized stage-wise error, further analysis is needed to provide end-to-end error bounds.

We envisage PyBlaz will find many uses as a light-weight compressor usable in settings where overall reliability is important, but exact error bounds are less important, such as for compressing artificial neural network inputs and parameters.

References

[1] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An efficient transformation scheme for lossy data compression with point-wise relative error bound," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 179–189, 2018.

[2] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, 08 2014.

[3] M. Martel, "Compressed matrix computations," in *IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2022*, pp. 68–76, IEEE, 2022.

[4] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE transactions on visualization and computer graphics*, vol. 12, pp. 1245–50, 09 2006.

harvey.dam@utah.edu

Acknowledgements: DOE ASCR Award Number DE-SC0022252 and NSF CISE Awards 1956106, 2124100, and 2217154.