Compressed Matrix Computations

Matthieu Martel
LAMPS, University of Perpignan
52, avenue Paul Alduy, Perpignan, France
matthieu.martel@univ-perp.fr

***Abstract*—Frugal computing is becoming an important topic for environmental reasons. In this context, several techniques have been proposed to reduce the storage of scientific data by dedicated compression methods specially tailored for arrays of floating-point numbers. While these techniques are quite efficient to save memory, they introduce additional computations to compress and decompress data before processing them. In this article, we introduce a new lossy, fixed-rate compression technique for 2D-arrays of floating-point numbers which allows one to compute directly on the compressed data, without decompressing them. We obtain important speedups since less operations are needed to compute among the compressed data and since no decompression and re-compression is needed. More precisely, our technique makes it possible to perform basic linear algebra operations such as addition, multiplication by a constant among compressed matrices and dot product and matrix multiplication among partly uncompressed matrices. This work has been implemented into a tool named blaz and we present a comparison with the well-known compressor zfp in terms of execution-time and accuracy.**

***Index Terms*—Green computing, Frugal Computing, Lossy compression, Floating-point numbers, Linear algebra.**

I. Introduction

At the beginning of this new decade, more than ten percent of the world's total electricity production is consumed by information and communication technology systems [15]. This huge consumption can be roughly decomposed into three equivalent parts: storage (30%), computing (30%) and networking (40%). This distribution imposes to whoever wants to develop frugal computing solutions, for ecological or maybe economical reasons, to attack these three points all together: to reduce i) storage, ii) computing and iii) networking, simultaneously. In this article, we propose a solution to this problem for the special case of matrix computations [7] which is ubiquitous in computer science.

Clearly, at the beginning of this new decade, scientific data remain the most important part of the world stored data, before Enterprise Resource Plannings (ERPs) or entertainment data for example [9]. So it is urgent to look for ways to process scientific data which offer simultaneously gains in terms of storage, computing and networking. In this article, the scientific data that we consider are large matrices of floating-point numbers [1] used to represent smooth curves. We will assume that there exist correlations between adjacent elements of matrices. This assumptions is also done by other kinds of compressors, for images for example [10].

Because usual compression techniques [10], [12] are not efficient enough on scientific data, specific (lossy) compressors have been developed such as `zfp` [19] or `sz` [5]. The performance of these tools is impressive in terms of data size reduction and accuracy, allowing to reduce storage by 10 to 20 without significant loss of relevant information. Doing so, they help to reduce the footprint of scientific data storage and communications. However, they increase the computing resources needed to solve a problem, since data must be uncompressed before processing and re-compressed after.

In this article, we introduce a new lossy compression technique for scientific data which makes it possible to compute directly among the compressed data, without decompression. While our method is not as efficient as `zfp` or `sz` in terms of compression rate and accuracy, it avoids a huge amount of operations since i) data do not need to be decompressed and re-compressed and ii) the compressed data being smaller, less operations are needed to compute with them. More precisely, our technique operates upon two-dimensional arrays storing matrices and allows one to perform basic linear algebra operations such as addition and multiplication by a constant on compressed matrices and dot product and matrix multiplication on partly decompressed matrices. To our knowledge, this is the first attempt to define a compression technique enabling one to compute directly on compressed data.

We can give a bird's eye view of our technique as follows. Our compressor is lossy, fixed-rate (11.37 compression rate) and operates on 8×8 blocks of IEEE754 `binary64` floating-point numbers [1]. After a block-splitting stage, we perform successive stages of normalization, prediction, transformation and, finally, quantization, all described in details in Section II.

A key-point is that all these stages act like linear maps (at the notable exception of the quantization done at the end of the scheme), i.e. the transformation t applied at any stage (but quantization) of our scheme satisfies $t(\mathbf{A}+\mathbf{B}) = t(\mathbf{A})+t(\mathbf{B})$ and $t(c\mathbf{A}) = ct(\mathbf{A})$ where $\mathbf{A}$ and $\mathbf{B}$ are matrices represented in the adequate data structures and where c is a scalar constant. These properties are used in Section III to design algorithms to add compressed matrices and to multiply them by constants without decompression. Additionally, we will also see in Section III that the dot product among lines and columns of compressed matrices as well as the product $\mathbf{A}\times\mathbf{B}$ of matrices can be computed on partially uncompressed matrices.

Last but not least, our compression scheme has been implemented into an open-source library named `blaz` and experimental results are given in Section IV. They show that the loss of accuracy introduced by our technique remains small, compared to the gains obtained in terms of execution

time and storage. For instance, the addition of matrices of size 2000×2000 with `blaz` is more than 50 times faster than with standard, uncompressed matrices and more than 5000 times faster than with `zfp` (see Section IV-A). On the other hand, The relative errors introduced by `blaz` are less than 10 times greater than with `zfp` for our basis of examples (see Section IV-B). This corresponds to the loss of one more decimal digit than with `zfp`. We strongly believe that this compromise is quite acceptable in many contexts. We also show in Section IV-B that `blaz` efficiently compresses climate simulations data coming from the Scientific Data Reduction Benchmarks [25].

As already mentioned, this article is organized as follows. The compression scheme used by `blaz` is detailed in Section II and the algorithms used to compute among compressed matrices are introduced in Section III. Experimental results are given in Section IV, Section V presents related work and Section VI concludes.

II. Compression Scheme

In this section, we introduce the five stages of our compression scheme which perform block-splitting, normalization, prediction, transformation and quantization. The originality of our compression scheme is to be compatible with the basic linear algebra operations that can be performed directly on the compressed data structures using the algorithms introduced in Section III. A summary of this scheme is given in Figure 1 and we detail the different steps hereafter.

a) Block Splitting: First of all, as in most fixed-rate compressors [19], we split the original matrix into blocks. In practice, our blocks have size 8×8.

Example II.1 *All along this section, we are going to illustrate how our compressor works using the following* 8×8 *matrix*

$$\mathbf{A} = \begin{pmatrix} 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.010 & 0.020 & 0.030 & 0.040 & 0.050 & 0.060 & 0.070 \\ 0.000 & 0.020 & 0.040 & 0.060 & 0.080 & 0.100 & 0.120 & 0.140 \\ 0.000 & 0.030 & 0.060 & 0.090 & 0.120 & 0.150 & 0.180 & 0.210 \\ 0.000 & 0.040 & 0.080 & 0.120 & 0.160 & 0.200 & 0.240 & 0.280 \\ 0.000 & 0.050 & 0.100 & 0.150 & 0.200 & 0.250 & 0.300 & 0.350 \\ 0.000 & 0.060 & 0.120 & 0.180 & 0.240 & 0.300 & 0.360 & 0.420 \\ 0.000 & 0.070 & 0.140 & 0.210 & 0.280 & 0.350 & 0.420 & 0.490 \end{pmatrix}.$$

We may assume that $\mathbf{A}$ *is one of the blocks obtained by splitting a larger original matrix. The values of* $\mathbf{A}$ *correspond to the function* $f(x, y) = x \times y$ *for* x *and* y *starting at* 0 *with a step of* 0.1*. The graphical representation of Block* $\mathbf{A}$ *is given in Figure 3.* ■

b) Block Normalization: After block splitting, the next step of our scheme is to normalize values. This stage is twofold. First, for each block element, we compute the difference between itself and its preceding elements. More precisely,

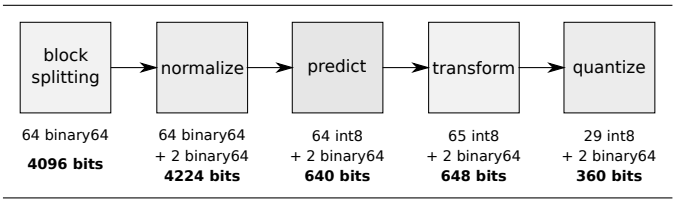

Fig. 1. Overview of `blaz` compression scheme.

taking as entry a 8×8 matrix $\mathbf{M}$ corresponding to a block, we compute the new block $\mathbf{\Delta}$ such that, for any $0 \leq i, j \leq 7$,

$$\mathbf{\Delta}_{ij} = \begin{cases} 0 & i = j = 0, \\ \mathbf{M}_{0,j} - \mathbf{M}_{0,j-1} & i = 0,\ j \neq 0, \\ \mathbf{M}_{i,0} - \mathbf{M}_{i-1,0} & i \neq 0,\ j = 0, \\ \frac{\left(\mathbf{M}_{i,j} - \mathbf{M}_{i-1,j}\right) + \left(\mathbf{M}_{i,j} - \mathbf{M}_{i,j-1}\right)}{2} & \text{otherwise.} \end{cases} \quad (1)$$

As in other compressors [5], [19], normalization relies on the assumption that block elements are correlated. The differences between consecutive values are then assumed to be small enough and this step aims at reducing the range of values occurring in the block. In addition to the matrix $\mathbf{\Delta}$, we have to store the value of the first element of the block $\mathbf{M}_{00}$ into a `binary64` number.

Example II.2 *By normalizing the matrix* $\mathbf{A}$ *of Example II.1 using the formula displayed in Equation (1), we obtain the new matrix*

$$\mathbf{A}_N = \begin{pmatrix} 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.010 & 0.015 & 0.020 & 0.025 & 0.030 & 0.035 & 0.040 \\ 0.000 & 0.015 & 0.020 & 0.025 & 0.030 & 0.035 & 0.040 & 0.045 \\ 0.000 & 0.020 & 0.025 & 0.030 & 0.035 & 0.040 & 0.045 & 0.050 \\ 0.000 & 0.025 & 0.030 & 0.035 & 0.040 & 0.045 & 0.050 & 0.055 \\ 0.000 & 0.030 & 0.035 & 0.040 & 0.045 & 0.050 & 0.055 & 0.060 \\ 0.000 & 0.035 & 0.040 & 0.045 & 0.050 & 0.055 & 0.060 & 0.065 \\ 0.000 & 0.040 & 0.045 & 0.050 & 0.055 & 0.060 & 0.065 & 0.070 \end{pmatrix}.$$

Let us observe that the values of $\mathbf{A}_N$ *are smaller than those of* $\mathbf{A}$*. We also store the element* $\mathbf{A}_{00} = 0.0$ *into a separate* `binary64` *number.* ■

The second step of the normalization stage consists of dividing the values in $\mathbf{\Delta}$ by the mean slope between consecutive values of the block $\mathbf{A}$ (considering only the non zero elements.) Since $\mathbf{\Delta}$ already contains the differences between adjacent elements of $\mathbf{A}$, we have

$$s = \frac{1}{K} \sum_{0 \leq i,j \leq 7} |\mathbf{\Delta}_{ij}| \quad (2)$$

with

$$K = \text{Card}\big(\{\mathbf{\Delta}_{ij},\ 0 \leq i, j \leq 7\ : \mathbf{\Delta}_{ij} \neq 0)\}\big) \quad (3)$$

and we compute

$$\mathbf{\Delta}' = \frac{1}{s} \cdot \mathbf{\Delta} \quad (4)$$

Note that the mean slope s must be stored into a `binary64` number by our compressor.

Example II.3 *Using the matrix $\mathbf{A}_N$ of Example II.2, we have $s = 0.04$ and the new matrix $\mathbf{A}'_N = \frac{1}{s} \cdot \mathbf{A}_N$ is*

$$\mathbf{A}'_N = \begin{pmatrix} 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.250 & 0.375 & 0.500 & 0.625 & 0.750 & 0.875 & 1.000 \\ 0.000 & 0.375 & 0.500 & 0.625 & 0.750 & 0.875 & 1.000 & 1.125 \\ 0.000 & 0.500 & 0.625 & 0.750 & 0.875 & 1.000 & 1.125 & 1.250 \\ 0.000 & 0.625 & 0.750 & 0.875 & 1.000 & 1.125 & 1.250 & 1.375 \\ 0.000 & 0.750 & 0.875 & 1.000 & 1.125 & 1.250 & 1.375 & 1.500 \\ 0.000 & 0.875 & 1.000 & 1.125 & 1.250 & 1.375 & 1.500 & 1.625 \\ 0.000 & 1.000 & 1.125 & 1.250 & 1.375 & 1.500 & 1.625 & 1.750 \end{pmatrix}.$$

Note that in $\mathbf{A}'_N$, for the sake of clarity, the values are rounded to the nearest after three decimal digits. Obviously, these values are computed in IEEE754 double precision in our implementation. ■

c) Prediction: This third stage of our compression scheme consists of replacing the slopes of the block $\mathbf{\Delta}'$ by a prediction. Let s_{max} be the maximal value in $\mathbf{\Delta}'$ in absolute value, $s_{max} = \max\{|\mathbf{\Delta}'_{ij}|,\ 0 \leq i,j \leq 7\}$, we take a set $S = \{s_0, s_1, \ldots, s_{255}\}$ of 256 equidistant points in the interval $[s - s_{max}, s + s_{max}]$ as our slopes of reference (see Figure 2). Then each value in $\mathbf{\Delta}'$ is approximated by its closest element in S. Assuming that, for all $0 \leq k < 255$, $s_k \leq s_{k+1}$, we store in a new matrix $\mathbf{\Psi}$ the indexes k of the slopes s_k which are the best approximations of the values in $\mathbf{\Delta}'$. Formally, for $0 \leq i,j < 8$, we have

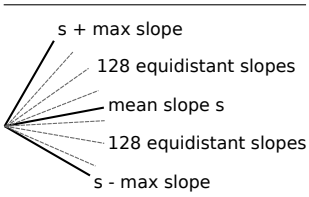

Fig. 2. Slopes of reference used in our prediction.

$$\mathbf{\Psi}_{ij} = k \text{ such that } |\mathbf{\Delta}_{ij} - s_k| \leq |\mathbf{\Delta}_{ij} - s_\ell|,\ \forall 0 \leq \ell \leq 255. \quad (5)$$

Note that $\mathbf{\Psi}$ is a 8×8 matrix of integers encoded on 8 bits. In addition to the mean slope s, we need to store s_{max} into a `binary64` floating-point number.

Example II.4 *Using the matrix $\mathbf{A}'_N$ of Example II.3, we compute the prediction matrix*

$$\mathbf{P} = \begin{pmatrix} 125 & 125 & 125 & 125 & 125 & 125 & 125 & 125 \\ 125 & 143 & 153 & 162 & 171 & 180 & 189 & 198 \\ 125 & 153 & 162 & 171 & 180 & 189 & 198 & 207 \\ 125 & 162 & 171 & 180 & 189 & 198 & 207 & 217 \\ 125 & 171 & 180 & 189 & 198 & 207 & 217 & 226 \\ 125 & 180 & 189 & 198 & 207 & 217 & 226 & 235 \\ 125 & 189 & 198 & 207 & 217 & 226 & 235 & 244 \\ 125 & 198 & 207 & 217 & 226 & 235 & 244 & 253 \end{pmatrix}.$$

The `binary64` values $A_{00} = 0.0$, and $s = 0.04$ are also stored. ■

d) Block Transform and Quantization: As in many other compressors (e.g. JPEG-2000 [10]), we use a Type II two-dimensional discrete cosine transform (DCT, [22]) of the block resulting from the normalization stage. DCTs are used to aggregate large coefficients in the first lines and columns of a matrix, small values occurring in the other elements after transformation. The quantization then consists of considering that these small coefficients are equal to 0 and of avoiding to store them in the compressed matrix. Our compressor only keeps the elements of the first two lines and columns of the matrix, i.e. 28 values. Note that the values returned by the DCT may be larger than 127 in absolute value. Let m be the greatest value in absolute value of the matrix $\mathbf{D}$ resulting from the DCT. We re-scale $\mathbf{D}$ by multiplying all its elements by

$$\varphi = \frac{127}{m}. \quad (6)$$

In order to recover the original matrix during the decompression phase, the value φ must also be stored into a 8 bits integer.

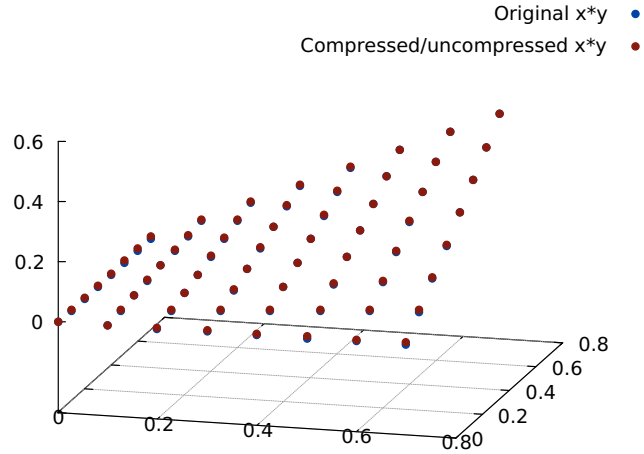

Fig. 3. Comparison between the original block of Example II.1 and its compressed/uncompressed version.

Example II.5 *After re-scaling, the DCT of matrix $\mathbf{P}$ of Example II.4 gives the new matrix*

$$\mathbf{P}' = \begin{pmatrix} \mathbf{122} & \mathbf{-51} & \mathbf{-11} & \mathbf{-14} & \mathbf{-8} & \mathbf{-8} & \mathbf{-4} & \mathbf{-2} \\ \mathbf{-51} & \mathbf{15} & \mathbf{7} & \mathbf{7} & \mathbf{5} & \mathbf{4} & \mathbf{3} & \mathbf{1} \\ \mathbf{-11} & \mathbf{7} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-14} & \mathbf{7} & 0 & 1 & 0 & 0 & 0 & 0 \\ \mathbf{-8} & \mathbf{5} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-8} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-4} & \mathbf{3} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{-2} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

For this example, all the non-zero coefficient but one in position $(3,3)$ already occur in the first two lines and columns of the matrix $\mathbf{P}'$ (in bold in the equation above) and the quantization only sets $\mathbf{P}'_{33}$ to 0 introducing a slight information loss. Recall that, in addition to $\mathbf{P}'$, the values $A_{00} = 0.0$, $s = 0.04$ and the scale factor $\varphi = 20$ are also stored. ■

Example II.6 *The difference between the original block of Example II.1 and the result of its compression and decompression by our method is displayed in Figure 3.* ■

Following our scheme, a compressed block is stored using 29 8-bits integers and 2 `binary64` floating-point numbers.

The 29 8-bits integers correspond to the 28 values that we keep after quantization plus one 8-bits integer corresponding to the re-scaling factor φ. A block is then stored into 360 bits instead of the original 4096 bits needed to store 64 `binary64` numbers, yielding a compression rate of 11.37.

III. Basic Linear Algebra Algorithms

In this section, we introduce the basic linear algebra algorithms used to compute among matrices following the scheme of Section II. More precisely, in what follows, we are going to introduce successively the algorithms for addition, multiplication by a constant, dot product and matrix multiplication. These operations are detailed for elementary 8×8 blocks. Obviously, for larger matrices, they are repeated for each elementary block.

All along this section, we use the following notations. Let $\mathbf{B}$ denote a 8×8 block of a compressed matrix, f and s denote respectively the values of the first element $\mathbf{B}_{00}$ and of the mean slope s of $\mathbf{B}$ as defined in Equation (2). Both f and s are `binary64` numbers. Next, φ denotes the scale factor φ introduced in Equation (6) to normalize the result of the DCT. Recall that φ is a 8-bits integer. Finally, `C` is an array of 28 8-bits values containing the coefficients quantized after the DCT.

a) Addition: The algorithm for the addition $\mathbf{B} = \mathbf{B}_1 + \mathbf{B}_2$ of two compressed blocks is given in Algorithm 1. First of all, at Line 1, the first element f of $\mathbf{B}$ is the addition $f_1 + f_2$ of the first elements of $\mathbf{B}_1$ and $\mathbf{B}_2$. Similarly, the mean slope of $\mathbf{B}$ is defined by $s = s_1 + s_2$. Next, we want $\varphi = \frac{127}{m}$ with $m = m_1 + m_2$. Then we have

$$\varphi = \frac{127}{m_1 + m_2} = \frac{127}{\frac{127}{\varphi_1} + \frac{127}{\varphi_2}} = \frac{1}{\frac{1}{\varphi_1} + \frac{1}{\varphi_2}} = \frac{\varphi_1 \varphi_2}{\varphi_1 + \varphi_2}. \quad (7)$$

Equation (7) motivates the computation carried out at Line 2 of Algorithm 1. Let $\mathbf{D}_1$ and $\mathbf{D}_2$ denote the blocks obtained after the DCT by passing $\mathbf{B}_1$ and $\mathbf{B}_2$ through the scheme of Figure 1. Intuitively, if no re-scaling were done in our scheme, we could simply add $\mathbf{D}_1$ and $\mathbf{D}_2$ to obtain the block $\mathbf{D}$ corresponding to $\mathbf{B}$. But two re-scalings are done in our scheme, during the normalization stage and among the values resulting from the DCT. The coefficients α_1 and α_2 are used to transform blocks made for scale factors φ_1 and φ_2 to the new scale factor φ, accordingly to Equation (4). Similarly, β_1 and β_2 are used to adapt the scale of the blocks resulting from the DCT to the scale factor φ. It is then possible to re-scale and add the coefficients of the DCT contained in $\mathtt{C}_1$ and $\mathtt{C}_2$ as done at Line 6, in the for loop of Algorithm 1. Note that adding the coefficients is possible since the DCT defines a linear map among our blocks. Formally, for a 8×8 block $\mathbf{\Psi}$ as defined in Equation (5), the two-dimensional DCT of $\mathbf{\Psi}$, denoted $\mathrm{DCT}(\mathbf{\Psi}) = \mathbf{D}$ is defined, for $0 \leq i, j < 8$, by

$$\mathbf{D}_{ij} = \alpha_i \alpha_j \sum_{u=0}^{7} \sum_{v=0}^{7} \mathbf{\Psi}_{ij} \cos\left[\frac{(2u+1)i\pi}{16}\right] \cos\left[\frac{(2v+1)j\pi}{16}\right] \quad (8)$$

and it follows that for two blocks $\mathbf{\Psi}'' = \mathbf{\Psi} + \mathbf{\Psi}'$,

$$\begin{aligned} & \mathbf{D}_{ij} + \mathbf{D'}_{ij} \\ = \;& \alpha_i \alpha_j \sum_{u=0}^{7} \sum_{v=0}^{7} \mathbf{\Psi}_{ij} \cos\left[\tfrac{(2u+1)i\pi}{16}\right] \cos\left[\tfrac{(2v+1)j\pi}{16}\right] \\ + \;& \alpha_i \alpha_j \sum_{u=0}^{7} \sum_{v=0}^{7} \mathbf{\Psi}'_{ij} \cos\left[\tfrac{(2u+1)i\pi}{16}\right] \cos\left[\tfrac{(2v+1)j\pi}{16}\right] \quad (9) \\ = \;& \alpha_i \alpha_j \sum_{u=0}^{7} \sum_{v=0}^{7} (\mathbf{\Psi}_{ij} + \mathbf{\Psi}'_{ij}) \cos\left[\tfrac{(2u+1)i\pi}{16}\right] \cos\left[\tfrac{(2v+1)j\pi}{16}\right] \\ = \;& \mathbf{D}''_{ij},\ 0 \leq i, j, < 8\ , \end{aligned}$$

where the coefficients α_i are the usual DCT coefficients defined by

$$\alpha_0 = \sqrt{\frac{1}{8}} \quad \text{and} \quad \alpha_i = \sqrt{\frac{1}{4}},\ 1 \leq i < 8\ . \quad (10)$$

Let us remark that the subtraction of blocks works similarly to the addition. We omit to detail it in the present article.

b) Multiplication by a Constant: In our framework, the multiplication by a constant c is the simplest operation to implement. The first element of the block and the slope are multiplied by c. The other elements remain unchanged. This is summarized in Algorithm 2. This algorithm is straightforward and we do not provide more details about it.

c) Dot product: Let $\mathbf{B}_1$ and $\mathbf{B}_2$ be two 8×8 matrix blocks and let $\langle \mathbf{B}_1, \mathbf{B}_2 \rangle_{ij}$ denote the dot product between the i^{th} line of $\mathbf{B}_1$ and the j^{th} column of $\mathbf{B}_2$, $0 \leq i, j < 8$. Our dot product requires a partial decompression of the blocks. More precisely, we have to recover the values resulting from the prediction stage of our scheme (third step of Figure 1 and Section II-c) and this implies to compute the inverse discrete cosine transform (IDCT) of the values contained in $\mathtt{C}_1$ and $\mathtt{C}_2$ (obviously, the values discarded by the quantization are replaced by zeros.)

Algorithm 1 Addition of two compressed matrix blocks $\mathbf{B}_1$ and $\mathbf{B}_2$.

1: $f \leftarrow f_1 + f_2$; $s \leftarrow s_1 + s_2$
2: $\varphi \leftarrow \mathrm{int}_8\big((\varphi_1 \times \varphi_2) \div (\varphi_1 + \varphi_2)\big)$
3: $\alpha_1 \leftarrow s_1 \div (s_1 + s_2)$; $\alpha_2 \leftarrow s_2 \div (s_1 + s_2)$
4: $\beta_1 \leftarrow \alpha_1 \div \varphi_1 \times \varphi$; $\beta_2 \leftarrow \alpha_2 \div \varphi_2 \times \varphi$
5: **for** $i \leftarrow 0$ to 27 **do**
6: $\quad \mathtt{C}[i] \leftarrow \mathrm{int}_8(\beta_1 \times \mathtt{C}_1[i] + \beta_2 \times \mathtt{C}_2[i])$
7: **end for**
8: **return** f, s, φ, `C`

Algorithm 2 Multiplication of a compressed matrix block $\mathbf{B}_1$ by a constant c.

1: $f \leftarrow f_1 \times c$; $s \leftarrow s_1 \times c$
2: $\varphi \leftarrow \varphi_1$
3: **for** $i \leftarrow 0$ to 27 **do**
4: $\quad \mathtt{C}[i] \leftarrow \mathtt{C}_1[i]$
5: **end for**
6: **return** f, s, φ, `C`

Algorithm 3 Dot product between Line i and Column j of the compressed matrix blocks $\mathbf{B}_1$ and $\mathbf{B}_2$.

$r \leftarrow 0$
$\mathbf{P}_1 \leftarrow \texttt{IDCT}(\texttt{C}_1)$; $\mathbf{P}_2 \leftarrow \texttt{IDCT}(\texttt{C}_2)$
$\mathbf{B}_1 \leftarrow \texttt{IPREDICT\_LINES}(\texttt{P}_1, i)$
$\mathbf{B}_2 \leftarrow \texttt{IPREDICT\_COLS}(\texttt{P}_2, j)$
for $k \leftarrow 0$ to 7 **do**
    $r \leftarrow r + \mathbf{B}_1[i,k] \times \mathbf{B}_2[k,j]$
end for
return r

Let $\mathbf{P}_1 = \texttt{IDCT}(\texttt{C}_1)$ and $\mathbf{P}_2 = \texttt{IDCT}(\texttt{C}_2)$ be the blocks obtained by applying the inverse discrete cosine transform to $\texttt{C}_1$ and $\texttt{C}_2$. We have to compute

$$\langle \mathbf{B}_1, \mathbf{B}_2 \rangle_{ij} = \sum_{k=0}^{7} \mathbf{B}_{1_{ik}} \mathbf{B}_{2_{kj}}. \tag{11}$$

where the coefficients of $\mathbf{B}_\ell$, $\ell \in \{1, 2\}$, are defined by

$$\mathbf{B}_{\ell_{ij}} = \begin{cases} f_\ell \text{ if } i = j = 0 \ , \\ \mathbf{B}_{\ell_{0,j-1}} + s_\ell \cdot \mathbf{P}_{\ell_{0,j-1}}, \text{ if } i = 0, \ 1 \leq j \leq 7 \ , \\ \mathbf{B}_{\ell_{i-1,0}} + s_\ell \cdot \mathbf{P}_{\ell_{i-1,0}}, \text{ if } 1 \leq i \leq 7, \ j = 0 \ , \\ \frac{\mathbf{B}_{\ell_{i-1,j}} + \mathbf{B}_{\ell_{i,j-1}}}{2} + s_\ell \cdot \mathbf{P}_{\ell_{i-1,j-1}} \text{ otherwise} \ . \end{cases} \tag{12}$$

Note that we do not need to compute the entire blocks $\mathbf{B}_1$ and $\mathbf{B}_2$. It is sufficient to compute the first i lines of $\mathbf{B}_1$ and the first j columns of $\mathbf{B}_2$ in order to compute the dot product $\langle \mathbf{B}_1, \mathbf{B}_2 \rangle_{ij}$. Algorithm 3 summarizes how we perform the dot product following equations (11) and (12). In this algorithm, the function $\texttt{IPREDICT\_LINES}(\texttt{C}, i)$ and $\texttt{IPREDICT\_COLS}(\texttt{C}, j)$ use Equation (12) to compute respectively the first i lines and j columns of $\mathbf{B}_1$ and $\mathbf{B}_2$.

d) Matrix multiplication: Our algorithm for the multiplication $\mathbf{B} = \mathbf{B}_1 \times \mathbf{B}_2$ of two blocks uses the same ideas than for the dot product and we do not detail it hereafter. The difference comes from the fact that the blocks resulting from the IDCT and from Equation (12) are needed many times since, for each $0 \leq i, j < 8$, the element $\mathbf{B}_{ij}$ corresponds to the dot product $\mathbf{B}_{ij} = \langle \mathbf{B}_1, \mathbf{B}_2 \rangle_{ij}$. To avoid redundant computations, these blocks are computed only once.

For large matrices made of many 8×8 blocks, the algorithms introduced in this section need to generalized. Briefly speaking, this corresponds to apply block-wise the former algorithms.

IV. Experimental Evaluation

Our compression scheme as well as the operations among compressed matrices have been implemented into an open-source library named `blaz`[1], written in `C`, and, in this section, we present two kinds of experiments, concerning the time performances (Section IV-A) and the accuracy (IV-B) of `blaz`. Altogether these experiments show that our technique makes it possible to save simultaneously storage and computation time. In both cases, comparisons with the `zfp` [19] library are reported. All these experiments have been carried out on a quad core Intel Core i7-1165G7 processor with 16 gigabytes of RAM and running Ubuntu 20.04.3 LTS. For the sake of simplicity, all our experiments are carried out on square matrices even though our techniques works for matrices of any dimensions. For the relevance of the comparison, the compression rate of `zfp` is set to a ratio equivalent to `blaz`. These experiments are completed by a evaluation of the precision of `blaz` when compressing climate simulation data coming from the Scientific Data Reduction Benchmarks [25].

A. Performance

In this section, the performances of the basic linear algebra operations performed by `blaz` are compared to the same operations done without any compression and to the case where matrices are compressed by `zfp` [19] (`C++` version).

First, we compare the execution time of operations among matrices compressed with `blaz` to the execution time of the same operations performed among uncompressed matrices. We use square matrices of sizes ranging from 80 to 2000 and the measures are done for addition, multiplication by a constant and for the dot product. Our results are displayed in Figure 4. Let us mention that the time in the y-axis of Figure 4 is given in logarithmic scale and that the execution time for the uncompressed dot product is almost null and is not represented. The first noticeable point is that the addition and multiplication by a constant are significantly faster with our compressed matrices than without any compression. Indeed, for square matrices of size 2000, the execution time is divided by 62 for addition (0.15ms vs 9.35ms) and by 58 for the multiplication by a constant (0.14ms vs 8.14ms.) Our technique makes it possible to compute faster than without any compression when small losses of accuracy are acceptable (see Section IV-B.) Concerning the dot product, the compressed

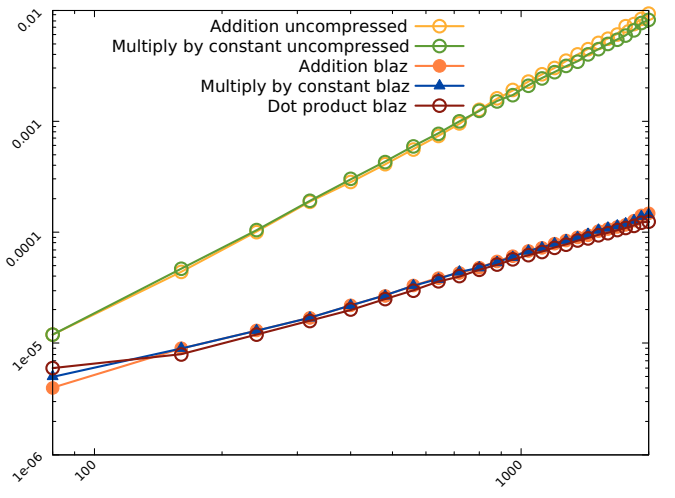

Fig. 4. Time measurement of operations in function of the size of the matrices for addition, multiplication by constant and dot product.

[1] https://github.com/mmartel66/blaz

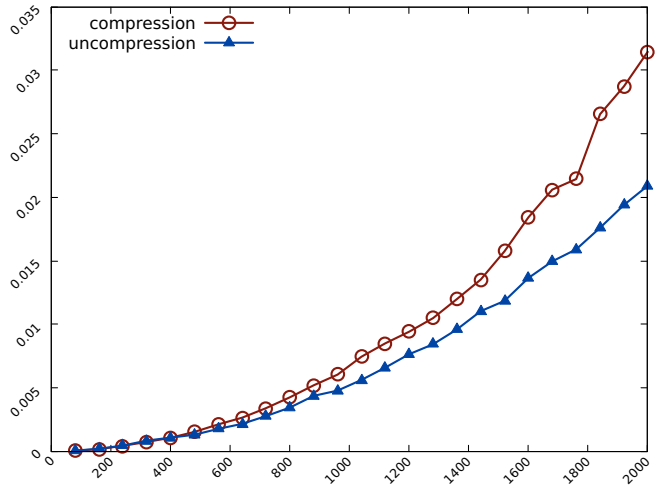

Fig. 5. Time measurement of `blaz` compression and decompression operations in function of the size of the matrices.

operation is slower with a compressed matrix than without compression (0.12ms vs 0.004ms which is 30 times slower.) This is due to the fact that a partial decompression of the matrix is done in this case (see Algorithm 3).

To complete our study, we have also measured the time needed by `blaz` to compress and decompress matrices. These times are given in Figure 5.

Second, we compare `blaz` to the well-known compressor `zfp`. For both tools, we start with compressed matrices. In the case of `zfp`, when a matrix element is read, the corresponding block is uncompressed and the value is returned. Conversely, when some value is assigned to a certain element, `zfp` directly modifies and re-compress the corresponding block. In the case of `blaz`, no decompression/re-compression is needed, the operations are carried out directly on the compressed matrices. The execution times are given in Figure 6 for our basic linear algebra operations. As in Figure 4, the execution time represented on the y-axis is displayed in logarithmic scale.

Again, we take square matrices of size ranging from 80 to 2000 and we show on a same graph the execution time taken by `blaz`, `zfp` and without compression. The top two graphs of Figure 6 are for addition and multiplication by a constant. The main observation is that `zfp` introduces a huge overhead due to the decompression and re-compression of matrices. For addition, taking as reference the execution time without any compression, `blaz` and `zfp` introduce respectively a speedup of 3 and a speed down of 121 for matrices of size 80 and a speedup of 62 and a speed down 115 for matrices of size 2000. In other terms, the addition is more than 50 times faster with `blaz` and more than 100 times slower with `zfp` compared to the uncompressed operation. Note that similar results are observed for the multiplication by a constant.

The execution times for the dot product are displayed in the left bottom corner of Figure 6. As already mentioned, this operation carried out with `blaz` matrices is slower than without any compression. Nevertheless, it remains significantly faster than with `zfp` matrices. Again, taking as reference the execution time without compression, `blaz` is 31 times slower for matrices of size 2000 while `zfp` is 62 times slower. In this case, `blaz` remains 2 times faster than `zfp`.

Finally, the graph in the bottom right corner of Figure 6 shows the performances of matrix multiplication. A first observation is that `blaz` is not much slower than the uncompressed operation. This is due to the fact that time spent in the matrix multiplication dominates the partial compression/decompression time. The second observation is that `zfp` is much slower than `blaz` (approximately 10 times.)

	M_1	M_2	M_3	M_4	M_5	M_6
Compression/Decompression of Matrices						
`blaz`	0.43%	0.39%	0.53%	0.44%	1.95%	1.17%
`zfp`	0.0006%	0.0009%	0.02%	0.002%	0.13%	0.15%
Additions of Compressed Matrices (`blaz` & `zfp`)						
M_1	–	0.98%	0.67%	0.91%	0.72%	0.78%
M_2	0.001%	–	0.62%	1.07%	1.76%	1.61%
M_3	0.82%	0.12%	–	2.27%	0.71%	0.65%
M_4	0.03%	0.42%	0.27%	–	1.68%	1.70%
M_5	0.68%	1.62%	1.25%	0.89%	–	0.94%
M_6	0.31%	1.27%	0.25%	2.32%	0.16%	–
Multiplications of Compressed Matrices by Constants						
`blaz`	0.43%	0.44%	0.53%	0.44%	1.95%	1.17%
`zfp`	0.001%	0.001%	0.03%	0.005%	0.12%	0.23%
Multiplications of Compressed Matrices (`blaz`)						
M_1	$2.0e^{-4}$	$2.0e^{-4}$	$1.0e^{-4}$	$5.1e^{-2}$	$3.6e^{-2}$	$9.2e^{-2}$
M_2	$2.0e^{-4}$	$2.0e^{-4}$	$1.0e^{-4}$	$5.3e^{-2}$	$3.6e^{-2}$	$1.0e^{-1}$
M_3	$3.4e^{-2}$	$3.5e^{-2}$	$7.9e^{-3}$	$9.0e^{-4}$	$5.0e^{-4}$	$5.0e^{-4}$
M_4	$3.7e^{-2}$	$3.4e^{-2}$	$2.1e^{-3}$	$8.0e^{-4}$	$6.0e^{-4}$	$5.0e^{-4}$
M_5	$4.4e^{-2}$	$4.5e^{-2}$	$1.7e^{-3}$	$8.0e^{-4}$	$1.1e^{-3}$	$1.7e^{-3}$
M_6	$1.8e^{-1}$	$3.7e^{-1}$	$2.0e^{-2}$	$8.0e^{-4}$	$7.0e^{-4}$	$2.9e^{-3}$

TABLE I
RELATIVE ERRORS ON THE RESULTS OF OPERATIONS AMONG THE MATRICES M_1 TO M_6. FOR ADDITION: UPPER RIGHT TRIANGLE: `BLAZ`. LOWER LEFT TRIANGLE: `ZFP`. FOR MULTIPLICATION: `BLAZ` ONLY.

B. Accuracy Measurements

In this section, we introduce a second set of experiments concerning the accuracy of `blaz`. We start by comparing our tool to `zfp`. Next, an evaluation of `blaz` accuracy on climate simulation data coming from the Scientific Data Reduction Benchmarks [25] is presented.

For our comparison with `zfp` and contrarily to the execution time measurements introduced in Section IV-A, the values of the matrix elements now matter. We use matrices M_k corresponding to the discretisation of the following six non-linear functions $f_k : \mathbb{R}^2 \to \mathbb{R},\ 1 \leq k \leq 6$,

$$f_1(x,y) = x \times y \qquad f_4(x,y) = x^2 \times y^2$$
$$f_2(x,y) = \frac{xy}{1+x^2+y^2} \qquad f_5(x,y) = \cos(\sqrt{x^2+y^2})$$
$$f_3(x,y) = x^2 - y \qquad f_6(x,y) = \cos(x^2+y^2) \cdot \mathrm{e}^{-0.1\cdot(x^2+y^2)}$$

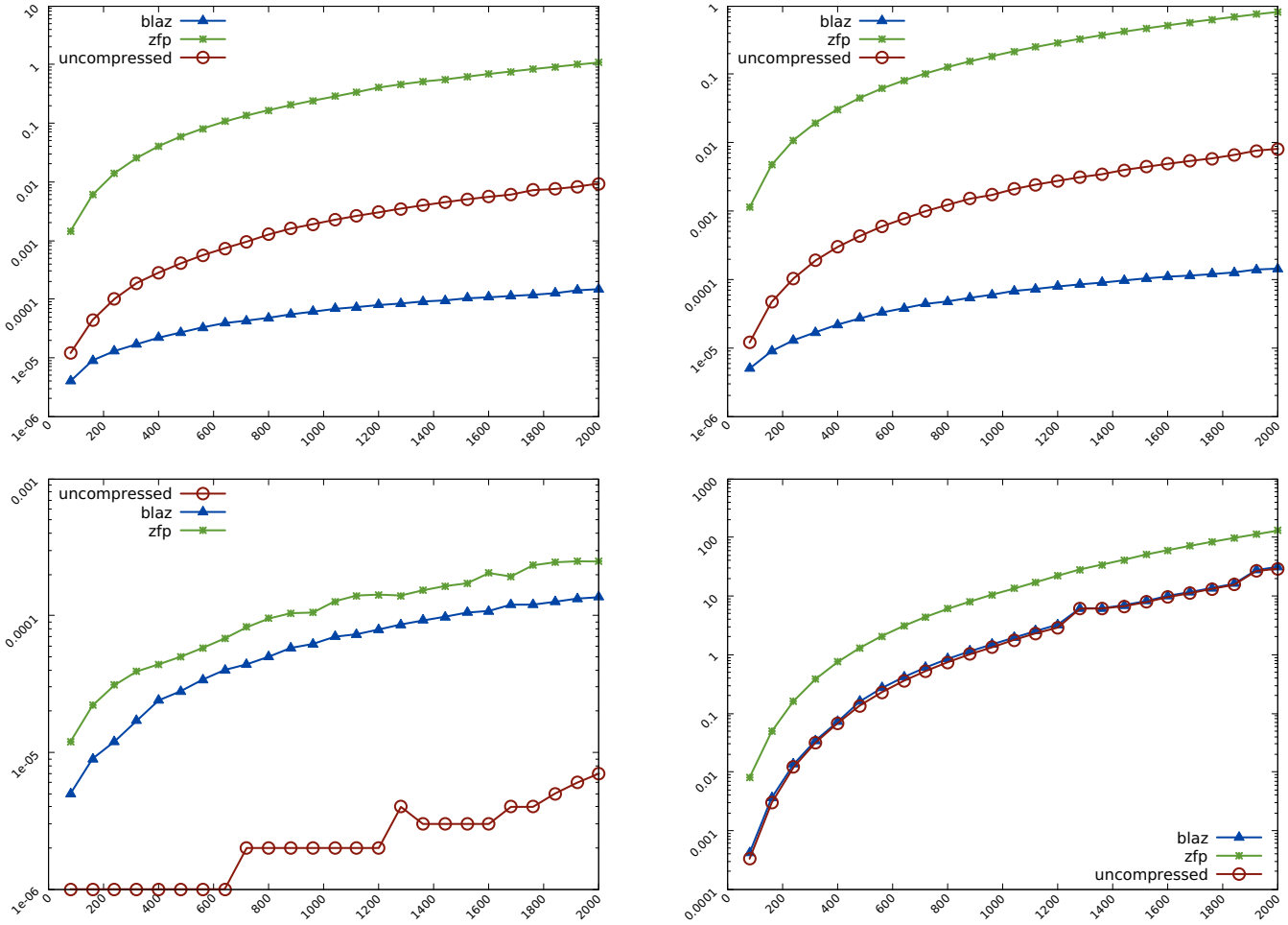

Fig. 6. Time measurement of operations in function of the size of the matrices (time given in logarithmic scale). Top left: Addition. Top right: Multiplication by a constant. Bottom left: Dot product. Bottom right: Matrix multiplication.

We set $M_{k_{ij}} = f_k(x_j, y_i)$ where the points (x_j, y_i) are taken in the range $[-2, 2] \times [-2, 2]$ with a constant step depending on the size of M_k.

Our experimental results are displayed in Table I. They correspond to mean relative errors computed as follows. Let M, M_1 and M_2 denote square matrices of size n whose elements are floating-point numbers. Let $\mathcal{C}$ and $\mathcal{U}$ denote the compression and decompression functions and let $M' = \mathcal{U}(\mathcal{C}(M))$, the mean relative error e on M' is

$$e = \frac{1}{n^2} \sum_{i,j=1}^{n} \left| \frac{M'_{ij} - M_{ij}}{M_{ij}} \right| . \tag{13}$$

We proceed similarly for matrices resulting from basic linear algebra operations. For instance, we apply Equation (13) to $M = M_1 + M_2$ and $M' = \mathcal{U}(\mathcal{C}(M_1) + \mathcal{C}(M_2))$.

The first two lines of Table I indicate the mean relative error on the compression/decompression of the matrices M_1 to M_6 using `blaz` and `zfp`. While `zfp` is more accurate than `blaz`, this latter remains accurate with relative errors of less than 0.5% in half of the cases and never greater than 2%.

The second part of Table I is dedicated to addition. We display the mean relative errors for all the additions $M_i + M_j$, $1 \leq i, j \leq 6$, $i \neq j$ performed on `blaz` matrices (upper right triangle) and on `zfp` matrices (lower left triangle). For example, the mean relative errors on $M_1 + M_6$ with `blaz` and `zfp` respectively are 0.78% and 0.31%. The errors introduced by `blaz` and `zfp` are comparable (same magnitude in general) even if `zfp` is better in most cases (at the price of the huge overhead in terms of execution time shown in Section IV-A.) In some cases, `blaz` is more accurate than `zfp` (e.g. for $M_4 + M_6$ `blaz` error is 1.70% while `zfp` error is 2.32%.) We may conclude that, for addition, there is not a clear advantage to use `zfp` instead of `blaz`.

The third part of Table I deals with the multiplication by a constant and, in our experiments, the matrices M_1 to M_6 are multiplied by 2. While being less accurate than `zfp`, the accuracy of the computations remain accurate, with relative errors around 1%. Note that changing the constant does not change the relative error.

The last part of Table I is for matrix multiplication. Let

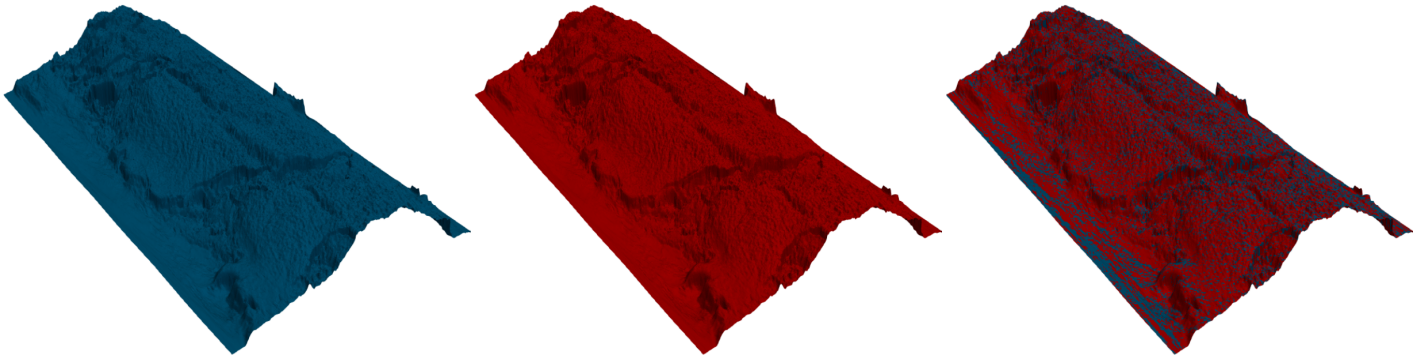

Fig. 7. Compression and decompression of the SRFRAD data of the CESM-ATM dataset of SDRBench. Left: Original data. Center: Result of the compression/decompression of the original data. Right: Differences between the two former sets of values.

us mention that, in `blaz`, the value and accuracy of the dot product $\langle M, M' \rangle_{ij}$ is the same than these of the element (i, j) of the matrix $M \times M'$. Then we only discuss the accuracy of the matrix multiplication which encompasses the one of the dot product. Matrix multiplication not being symmetric in general, we display in Table I the mean accuracies of all the products $M_i \times M_j$, $1 \leq i, j \leq 6$, computed following Equation (13). These accuracies are all for `blaz`. The errors obtained with `zfp` being almost zero in any case, we do not display them. We can see that the accuracy of `blaz` for matrix multiplication is very good (relative errors of order 10^{-3} or less in more than half of the cases and never greater than 10^{-1} in magnitude), even if they remain worse than `zfp`. We can also remark that `blaz` multiplication is more accurate than `blaz` addition. This is due to the fact that a partial decompression is done for the multiplication, contrarily to the case of addition. This better accuracy then results from more computations and, in future work, we would like to design even more frugal algorithms (yet less accurate) for the dot product and multiplication.

Name	Mean Error	Max Error	Range	Entropy	Variance
FLDS	0.000564	0.030513	325.53	6.76	8408.92
FLDSC	0.000449	0.037887	325.39	6.851068	8874.69
FLNT	0.000453	0.025463	180.10	6.704787	1500.09
FLNTC	0.000283	0.034623	175.35	6.668748	1866.13
FLUT	0.000451	0.025458	180.78	6.703362	1507.86
FLUTC	0.000281	0.034482	176.03	6.668475	1879.43
PCONVB	0.000221	0.024115	51533.16	6.381181	$445.5 \cdot 10^8$
PCONVT	0.000589	0.050704	64182.17	6.562340	$539.0 \cdot 10^8$
PS	0.000214	0.024079	51535.69	6.372430	$451.8 \cdot 10^8$
PSL	0.000015	0.000685	7084.63	6.165232	$180.8 \cdot 10^6$
SRFRAD	0.000771	0.061515	600.76	6.976098	$327.7 \cdot 10^5$
TREFHT	0.000088	0.006900	74.32	6.505165	472.23
TREFMNAV	0.000105	0.008555	73.90	6.519205	534.82
TREFMXAV	0.000084	0.005932	80.46	6.504760	421.66
TROP_P	0.000359	0.014251	20954.13	6.741598	$473.7 \cdot 10^7$
TROP_T	0.000042	0.001487	32.54	6.432132	92.30
TROP_Z	0.000307	0.016175	12109.46	6.795903	$120.3 \cdot 10^7$
TS	0.000121	0.013840	80.51	6.515242	515.90
TSMN	0.000244	0.034819	92.32	6.553231	862.96
TSMX	0.000135	0.027411	95.01	6.476401	355.07

TABLE II
MEAN AND MAXIMAL RELATIVE ERRORS INTRODUCED BY `BLAZ` ON CLIMATE SIMULATION DATA FROM THE SDR BENCHMARK.

We end this section by introducing an evaluation of `blaz` accuracy when compressing and decompressing real scientific data. We use climate simulation data coming from the Scientific Data Reduction Benchmarks (SDRBench[2]) [25] (Dataset 1 of the CESM-ATM dataset.) This dataset contains arrays of size 3600×1800. An example showing the original values of one of these arrays (SRFRAD), the corresponding compressed/uncompressed arrays and the difference between the former two arrays is displayed in Figure 7.

We show in Table II the mean and maximal relative errors obtained when compressing and decompressing 20 arrays taken from the CESM-ATM dataset of SDRbench. The relative errors are computed between the original and compressed/uncompressed values. The mean error is the mean of the relative errors on each element of the array and the maximal error is the maximal relative error obtained among all the elements of the array. The columns Range, Entropy and Variance of Table II correspond respectively to the range of values among the elements of the array, to their entropy and to their zero-mean variance (these values are coming from SDRbench and are simply reproduced here in order to provide a better characterization of the data.)

We can observe that for all the considered arrays of Table II, the mean and maximal relative errors are respectively less than 0.1% and 3.5%. This assess the ability of `blaz` to compress realistic scientific data with acceptable accuracy while offering the ability to compute directly with these compressed data.

V. Related Work

Roughly speaking, compressors for scientific data (i.e. arrays of floating-point numbers) can be classified into two categories: Either the user sets the compression rate and this determines the accuracy of the encoding or the user sets the accuracy and this determines the compression rate. The compressors `zfp` [19] and `fpzip` [20] before it have been initially designed as compressors of the first category. Conversely, `sz` [5], [18] has been designed as a compressor of the second category. Elements of comparison between these two approaches can be found in [23]. Note that the current

[2]https://sdrbench.github.io/

versions of `zfp` and `sz` offer many options which make them encompass the categories mentioned earlier.

Another related topic is about the use of GPUs for scientific data compression [14], [17], [24]. Finally, much attention has also been paid recently to the compression of neural networks [13], [16], [21]. We do believe that the ideas developed here could be useful in the context of neural networks which perform mainly basic linear algebra operations.

VI. Conclusion and Perspectives

The work presented in this article opens a new research direction and many perspectives remain to be explored. First of all, we would like to determine formal error bounds for our compressor (in the spirit of [6]) and, more interestingly, for the operations on compressed matrices. We strongly believe that the error on the result of some operation between compressed matrices can be expressed in function of the errors on the compressed matrices corresponding to the operands.

Second, we aim at making the compression rate tunable, more or less compression being permitted with the corresponding impact on accuracy. This can be achieved by skewing the quantization or by introduction an interpolation stage in our scheme. For example, we could add a Lorenzo predictor compatible with our basic linear algebra operations [11].

Third, we believe that compressors allowing to compute on the compressed matrices could be useful in other contexts. For example a compressor for integers could be useful to perform fast and frugal basic linear algebra operations in the context of image processing. As well, a `blaz`-like compressor designed for embedded systems could be useful for many applications with memory constraints.

Next, we plan to extend our library, with new linear algebra operators (e.g. Hadamard product) but also, our compression scheme processing every 8×8 block of a matrix independently, we would like to develop a parallel implementation of our `blaz` library, typically based on the MPI library [8].

Yet another perspective is to use compressed matrices in precision tuning tools [4]. Precision tuning consists of reducing the precision of the variables of a program up to some accuracy requirement set by the user. It would be interesting to use the precision recommended by the precision tuning tools to compress arrays up to this requirement.

Finally, we aim at testing further the `blaz` library. We aim at evaluating it in the context of large numerical simulations to assess its performances in term of execution-time and accuracy on more realistic use-cases [2], [3].

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, 2008.

[2] A. H. Baker, D. M. Hammerling, S. A. Mickelson, H. Xu, M. B. Stolpe, P. Naveau, B. Sanderson, I. Ebert-Uphoff, S. Samarasinghe, F. De Simone, F. Carbone, C. N. Gencarelli, J. M. Dennis, J. E. Kay, and P. Lindstrom. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development*, 9(12):4381–4403, 2016.

[3] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T. Chong. Use cases of lossy compression for floating-point data in scientific data sets. *J. High Perform. Comput. Appl.*, 33(6), 2019.

[4] Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: A survey. *ACM Comput. Surv.*, 53(2):33:1–33:35, 2020.

[5] Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *IEEE Int. Parallel and Distributed Processing Symposium, IPDPS*, pages 730–739. IEEE Computer Society, 2016.

[6] James Diffenderfer, Alyson Fox, Jeffrey A. F. Hittinger, Geoffrey Sanders, and Peter G. Lindstrom. Error analysis of ZFP compression for floating-point data. *SIAM J. Sci. Comput.*, 41(3):A1867–A1898, 2019.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Fourth Edition*. Johns Hopkins University Press, 2013.

[8] William D. Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface, 3rd Edition*. Scientific and engineering computation. MIT Press, 2014.

[9] Linda C. Harwell, Deborah N. Vivian, Michelle D. McLaughlin, and Stephen F. Hafner. Scientific data management in the age of big data: An approach supporting a resilience index development effort. *Frontiers in Environmental Science*, 7:72, 2019.

[10] Abir Jaafar Hussain, Ali Al-Fayadh, and Naeem Radi. Image compression techniques: A survey in lossless and lossy algorithms. *Neurocomputing*, 300:44–69, 2018.

[11] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22(3):343–348, 2003.

[12] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University - Computer and Information Sciences*, 33(2):119–140, 2021.

[13] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *High-Performance Parallel and Distributed Computing, HPDC*, pages 159–170. ACM, 2019.

[14] Sian Jin, Pascal Grosset, Christopher M. Biwer, Jesus Pulido, Jiannan Tian, Dingwen Tao, and James P. Ahrens. Understanding gpu-based lossy compression for extreme-scale cosmological simulations. In *2020 IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 105–115. IEEE, 2020.

[15] Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561:163–166, 09 2018.

[16] Vinu Joseph, Ganesh Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

[17] Fabian Knorr, Peter Thoman, and Thomas Fahringer. ndzip: A high-throughput parallel lossless compressor for scientific data. In *31st Data Compression Conference, DCC 2021*, pages 103–112. IEEE, 2021.

[18] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *IEEE Int. Conference on Big Data*, pages 438–447. IEEE, 2018.

[19] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2674–2683, 2014.

[20] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1245–1250, 2006.

[21] James O'Neill. An overview of neural network compression. *CoRR*, abs/2006.03669, 2020.

[22] Gilbert Strang. The discrete cosine transform. *SIAM Rev.*, 41(1):135–147, 1999.

[23] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP. *IEEE Trans. Parallel Distributed Syst.*, 30(8):1857–1871, 2019.

[24] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. Optimizing error-bounded lossy compression for scientific data on gpus. In *Cluster Computing*, pages 283–293. IEEE, 2021.

[25] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. SDRBench: Scientific data reduction benchmark for lossy compressors. In *IEEE Int. Conference on Big Data*, pages 2716–2724. IEEE, 2020.