ESAIM: PROCEEDINGS AND SURVEYS, October 2024, Vol. 77, p. 79-99
Emmanuel Franck, Hélène Hivert, Guillaume Latu, Hélène Leman, Bertrand Maury, Michel Mehrenberger, Laurent Navoret

REDUCING THE MEMORY USAGE OF STRUCTURED GRID SOLVERS WITH WAVELET COMPRESSION

Clément Flint[1] and Philippe Helluy[2]

Abstract. This paper presents a new solution to address the challenge of increasing memory usage in high-performance computing simulations on structured grids. Our approach utilizes a lossy compression scheme based on the Discrete Wavelet Transform (DWT) to achieve high compression ratios while preserving the accuracy of the simulation. Our evaluation on a finite volume scheme demonstrates that the approach can reduce memory usage by several orders of magnitude.

Résumé. Ce papier présente une nouvelle solution pour faire face à l'augmentation de l'utilisation de la mémoire dans les simulations haute performance sur des grilles structurées. Notre approche utilise un schéma de compression avec perte basé sur la transformée par ondelettes discrète pour obtenir des taux de compression élevés tout en préservant la précision de la simulation. Notre évaluation sur un schéma de volumes finis démontre que l'approche peut réduire l'utilisation de la mémoire de plusieurs ordres de grandeur.

Contents

Introduction

Many numerical methods exist for solving partial differential equations. In several of these methods (e.g. Finite Difference, Finite Volume, Lattice-Boltzmann), the unknowns are defined at the nodes of a 2D or 3D structured stencil. These methods are typically used for their ability to accurately simulate fluid flows, for

[1] clement.flint@inria.fr
[2] philippe.helluy@unistra.fr

Article published online by EDP Sciences and available at https://www.esaim-proc.org or https://doi.org/10.1051/proc/202477079

instance. We refer the reader to [14] for an introduction to the Finite Difference (FD) method, to [10, 17] for general descriptions of the Finite Volume (FV) method and to [24] for a general description of the Lattice-Boltzmann Method (LBM) and. Because the space is discretized on a regular grid, FD/FV/LBM schemes fall into the class of what computer scientists call stencil algorithms. A stencil algorithm is a type of algorithm where the next state of each cell is a function φ of the values of its N relative neighbors. The arrangement of the N neighbors can be arbitrary. This arrangement is referred to as the stencil of the algorithm.

Our primary concern is to improve FD/FV/LBM simulations (and more generally stencil algorithms) that run on a single-GPU machine. This assumption allows us to have a case where memory usage is by far the most limiting factor of the algorithm, making it a good candidate for the use of data compression techniques.

The use of lossless data compression in the field of high-performance computing has already been explored in several works [2, 4]. Besides memory saving, the use of data compression can also be motivated by the need to transfer data between the CPU and the GPU efficiently. It is possible to find situations where the cost of the data compression/decompression is largely compensated by the acceleration of the data transfers. In practice, the compression is achieved through the use of general data compression libraries such as nvCOMP [22] or zfp [18].

However, the data handled by LBM (or FD/FV) schemes are subjects to numerical errors generated by the underlying approximation. Thus, small additional errors caused by a lossy compression method are acceptable in the whole algorithm. The data also have special structures that can be exploited to design a superior compression algorithm. In many physical applications, the computed fields present large regions with smooth variations, separated by sharp discontinuities. This is the case, for instance, in the simulation of fluid flows. In this paper, we present a lossy compression algorithm for FD/FV/LBM schemes that is based on the Discrete Wavelet Transform (DWT), which is well adapted to this type of data.

In the FV framework, multiresolution schemes based on wavelet analysis have a long history. We can mention, among many others, [1, 8, 11]. In these works, the general strategy is to represent the data in the multiscale (wavelet) basis and compute directly the evolution of the solution in this basis, i.e. on the locally refined grid. This leads to very complicated software implementations, with highly non-uniform data access. The programming is even more complex when parallel and hybrid CPU/GPU computing is addressed [5, 6]. In the LBM framework, similar works and strategies have been proposed. See, for instance, [3, 15, 27].

In addition to the high complexity, in many practical cases, these implementations are not very efficient. Finally, they often require a complete rewriting of the FV (or FD/LBM) scheme they are based on.

In this work, we propose and evaluate a much simpler strategy, which avoids algorithms that are difficult to parallelise. The simple idea is to perform a patch decomposition of the computational domain and apply wavelet compression/decompression in each patch, in turn.

In this way, most of the patches are stored in memory in a compressed form, while the patch that is currently being processed is stored in memory in an uncompressed form. This allows us to reduce the memory usage of the algorithm.

There are two main difficulties in this approach. The first point is that the compression is lossy. One has to verify that this loss is small and can be controlled by the user through a threshold value. Secondly, when shock waves are present, it is important to build a globally conservative scheme, in order to ensure the convergence of the scheme under mesh refinement [12]. In this work, we propose a simple way to ensure the mass conservation of the wavelet compression algorithm at the boundaries between the patches. Then, our approach can be used with any standard FD/FV/LBM scheme that uses a regular grid.

In this work, we propose a preliminary performance analysis of our approach on a single patch. We show that using this algorithm will indeed reduce the memory usage of the numerical scheme while preserving its quality. We also evaluate the cost of the data compression. We show that the compression time can be significantly lower than the computation of the FD/FV/LBM scheme. This means that the method is promising for multi-patch computations because the memory transfers between the main memory and the computing accelerators are often the most time-consuming part of the algorithm.

In Section 1, we present the numerical discretization that we use for approximating systems of conservations. In Section 2, we explain how we build the compression pipeline that we use in our experiments. In section 3, we perform experiments to demonstrate the efficiency of our method.

1. Numerical discretization

In this section, we describe the FV and LBM schemes that we use in our experiments. The objective is to solve numerically the following system of conservation laws:

$$\partial_t W + \nabla \cdot (Q(W)) = 0. \tag{1}$$

here the unknown $W(X,t)$ is a vector of m conservative variables depending of the space variable $X = (x,y)$ and the time variable t. For simplicity, we assume that X is in the square $]0,L[\times]0,L[$, but more complex shapes are possible.

The divergence operator is defined by

$$\nabla \cdot (Q(W)) = \frac{\partial}{\partial x} Q^x(W) + \frac{\partial}{\partial y} Q^y(W),$$

where Q^x and Q^y are two application from $\mathbb{R}^m$ to $\mathbb{R}^m$. For a given two-dimensional vector $N = (N_x, N_y)$, we define the flux of the system of conservation laws by

$$Q(W,N) = N_x Q^x(W) + N_y Q^y(W). \tag{2}$$

Finally, we define the four directions

$$N^0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad N^1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad N^2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad N^3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}. \tag{3}$$

In this way:

$$Q(W,N^0) = Q^x(W), \quad Q(W,N^1) = -Q^x(W),$$
$$Q(W,N^2) = Q^y(W), \quad Q(W,N^3) = -Q^y(W).$$

Let n_x be a positive integer. For simplicity, we also define $n_y = n_x$. We approximate W on a regular grid of step

$$h = \frac{L}{n_x}.$$

The grid is made of little square cells

$$C_{i,j} =]ih,(i+1)h[\times]jh,(j+1)h[, \quad i = 0,\dots,n_x-1, \quad j = 0,\dots,n_y-1.$$

For simplicity, we can assume a periodicity condition

$$i + n_x \equiv i, \quad j + n_y \equiv j,$$

which allows to extending the grid to any couple $(i,j) \in \mathbb{Z}\times\mathbb{Z}$. We could also apply boundary conditions on the boundary cells. The centers of the cells are the points

$$X_{i,j} = \begin{pmatrix} x_i \\ y_j \end{pmatrix} = \begin{pmatrix} ih + h/2 \\ jh + h/2 \end{pmatrix}.$$

The conservative data are approximated at times $t = n\tau$, at the center of the cells

$$W_{i,j}^n \simeq W(X_{i,j}, p\tau). \tag{4}$$

The system of conservation laws (1) is then approximated by the following FV scheme, which allows computing the value at time step $n+1$ from that of the time step n

$$W_{i,j}^{n+1} = W_{i,j}^n - \frac{\tau}{h}\sum_{k=0}^{3} Q(W_{i,j}, W_{i',j'}, N^k), \quad \text{with} \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} + N^k. \tag{5}$$

In this formula, we have introduced the numerical flux $Q(W, W', N)$, whose purpose is to approximate the flux $Q(W, N)$ at the interface between two cells. The numerical flux has to satisfy some mathematical property in order to ensure a stable and accurate approximation. It is out of the scope of this work to discuss this aspect. We refer (for instance) to [10, 17].

The implementation of this algorithm is done in the following way:

- The data at time n and $n+1$ are stored into two buffers of floats `wn` and `wnp1`.
- The initial data at time $t = 0$ is known and allows to construct the buffer `wn` thanks to (4), which reads here
$$W_{i,j}^0 = W(X_{i,j}, 0).$$
- At each time step n we compute `wnp1` from `wn` on the full grid, thanks to (5). These computations make many calls to the numerical flux function $Q(W, W', N)$, which is the heaviest part of the algorithm. It is important to take care of the data arrangement in the buffers `wn` and `wnp1` in order to ensure optimal memory access.
- At the end of the time step `wnp1` is copied into `wn`, before the next step.
- When the final time T, is reached, i.e. when $n\tau \geq T$, the results are displayed.

2. COMPRESSION

In this section, we explain how we build the compression pipeline that we use in our experiments. The compression pipeline is comparable to that of the JPEG2000 standard [21]. The goal of this pipeline is to apply a pre-processing to the data before applying a lossless compression algorithm. Applying a pre-processing allows us to take advantage of the spatial structure of the data to make some symbols (in this case: zeros) more frequent.

The compression pipeline is composed of three steps. In the first step, a Discrete Wavelet Transform (DWT) is applied to the data (Subsection 2.1). The purpose of this step is to create numerous near-zeros without compromising the information content. In the second step, a threshold is applied to the resulting DWT coefficients (Subsection 2.2). The coefficients whose absolute value is less than the threshold are set to zero. This step introduces the loss of information in the compression pipeline. The loss increases as the threshold increases. The final step is the application of a lossless compression algorithm to the thresholded DWT coefficients (Subsection 2.3). This step achieves effective memory compression. It is anticipated that the thresholding step will increase the frequency of the "zero" symbol, thus enabling the lossless compression algorithm to achieve a better compression ratio.

In the following Subsection 2.1, we describe the used discrete wavelet transform. In Subsection 2.2, we present the thresholding step. In Subsection 2.3, we describe two different lossless compression algorithms that we use in our experiments.

2.1. **Wavelet Transform**

The Discrete Wavelet Transform (DWT) is a mathematical tool that is used to decompose a signal into a set of coefficients. It is a discretization of the Continuous Wavelet Transform (CWT). The idea is to decompose a

sampled signal into a sum of wavelets. For a very general introduction to wavelet theory and applications, we refer to the book of Mallat [19].

In this work, we refer to Le Gall-Tabatabai (LGT) 5/3 wavelets [16]. The LGT5/3 wavelets, also known as the 5/3 biorthogonal wavelets are known for their ability to achieve first-order compression, which means that they filter out linear polynomials. This makes them well suited for a wide range of applications, including image and audio compression. They are particularly effective at preserving the important structure of a signal while achieving high compression ratios. They are also efficient for compressing discontinuous data, because of their short filters that concentrate the compression analysis on small regions of the signal.

In this work, we use a variation of the LGT5/3 wavelets for the non-periodic case and a discrete signal of length of $2^k + 1$, $k \in \mathbb{N}$. This variation is based on a folding of the wavelet (see [19] p. 321). This allows for improving the compression ratio in the non-periodic case. In addition, we show that this construction satisfies a fundamental conservation property, which is mandatory for ensuring the convergence of the numerical scheme under grid refinement [12].

In practice, our implementation is based on the wavelet lifting approach introduced by Sweldens [25, 26].

2.1.1. *Discrete wavelet transform in the periodic case*

Let us first introduce the wavelet compression scheme for a 1-periodic function f on the real line. Its definition on the interval $[0, 1]$ is thus sufficient.

We define the following sampling points in $[0, 1]$:

$$x_{j,k} = k2^{-j}, \quad 0 \leq k < 2^j, \quad j \geq 0.$$

The j index is the scale index. The lower scales correspond to the *coarser* grids and the higher scales to the finer grids. In some sense, the scale index j gives the highest frequency that can be represented by the grid $\{x_{j,k}\}_{0 \leq k < 2^j}$. Let us also remark that the grid at scale j has 2^j points. The first point is always 0 and the last point is always $1 - 2^{-j}$:

$$x_{j,0} = 0, \quad x_{j,2^j-1} = 1 - 2^{-j}.$$

For a a given scale $j = j_0 \geq 1$ the function is sampled at the grid points

$$s_{j,k} \simeq f(x_{j,k}).$$

The idea of the wavelet transform is to transform the set of samples $(s_{j,k})_{0 \leq k < 2^j}$ at scale j into a set of samples $(s_{j-1,k})_{0 \leq k < 2^{j-1}}$ at the coarser scale $j - 1$ and details $(d_{j,k})_{0 \leq k < 2^{j-1}}$. The precise formula for computing the samples and details are given in the next subsection. The role of the details is to allow the reconstruction of the $s_{j,k}$'s from the $s_{j-1,k}$'s because there is a loss of information in the coarsening process. Let us define

$$u = \begin{pmatrix} s_{j,0} \\ s_{j,1} \\ \vdots \\ s_{j,2^j-1} \end{pmatrix}$$

and

$$v = \begin{pmatrix} s_{j-1,0} \\ d_{j-1,0} \\ s_{j-1,1} \\ d_{j-1,1} \\ \vdots \\ s_{j-1,2^{j-1}-1} \\ d_{j-1,2^{j-1}-1} \end{pmatrix}.$$

A step of the wavelet transform can be defined in matrix form

$$v = Au. \tag{6}$$

The even rows of A correspond to the $s_{j-1,k}$'s while the odd rows correspond to the $d_{j-1,k}$'s. The resulting v vector gives us the grid at scale index $j-1$ as well as the details. The details allow us to reconstruct the exact $s_{j,k}$'s from the $s_{j-1,k}$'s. If the DWT scheme is properly designed, the transformation is bijective and there is an A^{-1} matrix such that

$$u = A^{-1}v.$$

2.1.2. *Wavelet transform in the non-periodic case*

We now adapt the previous scheme to the non-periodic case. The difference with the periodic case is that we have to take into account the boundaries of the interval. We propose a wavelet construction where the wavelets filter out linear polynomials.

We consider a function on the interval $[0,1]$. We define the following sampling points

$$x_{j,k} = k2^{-j}, \quad 0 \leq k \leq 2^j, \quad j \geq 0.$$

At scale j, the signal is represented by the grid $\{x_{j,k}\}_{0\leq k\leq 2^j}$ and has 2^j+1 points. The first point is always 0 and the last point is always 1:

$$x_{j,0} = 0, \quad x_{j,2^j} = 1.$$

They both correspond to even index points. The usual wavelet constructions are on the whole real line or on a periodic domain. The separation between even or odd indices k is very important. In the usual wavelet constructions, there are as many odd points as even points. Because we are on an interval, at scale $j \geq 1$ we have $2^{j-1}+1$ even indices and 2^{j-1} odd indices.

For a given scale $j = j_0 \geq 1$ the function is sampled at the grid points

$$s_{j,k} \simeq f(x_{j,k}).$$

At coarser scales, we decide to always keep the boundary values unmodified

$$s_{j-1,0} = s_{j,0} = f(0), \quad s_{j-1,2^{j-1}} = s_{j,2^j} = f(1). \tag{7}$$

The idea of having different resolutions through the j scales is called the multiresolution analysis. This technique is aimed to represent the samples more efficiently by keeping only the relevant information at scale $j < j_0$. Another aspect of the multiresolution analysis is that it gives a way to extrapolate f at finer scales $j > j_0$ on the dyadic points $x_{j,k}$.

If the signal represented by f has smooth variations, it seems reasonable to keep only the even samples $s_{j,2k}, \quad 0 \leq k \leq 2^{j-1}$.

By linear interpolation, we expect that the odd samples satisfy

$$s_{j,2k+1} \simeq \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1}-1.$$

Because it is not exact, we keep track of the details at the scale $j-1$

$$d_{j-1,k} = s_{j,2k+1} - \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1}-1,$$

which allows us to exactly reconstruct the initial odd values at the scale j:

$$s_{j,2k+1} = d_{j-1,k} + \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1}-1.$$

Now, we introduce a mass conservation requirement:

$$\frac{f(0)+f(1)}{2}+\sum_{k=1}^{2^{j-1}-1} s_{j-1,k}=\frac{f(0)+f(1)}{4}+\frac{1}{2}\sum_{k=1}^{2^{j}-1} s_{j,k}. \tag{8}$$

Essentially, this is a quadrature formula that states that the mass of the samples at the scale j equals the mass of the samples at the scale $j-1$. We use the trapezoidal quadrature formula. This explains why the weights of the first and last samples are halved. The mass conservation property (8) is an essential property for ensuring the convergence of the whole numerical scheme. It is proved to be necessary for instance in [12]. However, it is not solely sufficient. To prove the convergence of the entire approximation, it is also crucial to control the error induced by the compression, as pointed out in [8].

Condition (8) ensures that the mass of the signal $s_{j,k}$ is concentrated in the samples $s_{j-1,k}$. This implies that the details $d_{j-1,k}$ wear no mass. Therefore, the detail coefficients $d_{j-1,k}$ can be modified without affecting the mass of the signal.

The last equation that needs to be set is the relation between $s_{j-1,k}$ and $s_{j,2k}$. The naive relation $s_{j-1,k}=s_{j,2k}$ does not always verify the mass conservation property (8). To satisfy this property, we introduce $\alpha_{j,k}$ coefficients and we set

$$s_{j-1,k}=s_{j,2k}+\alpha_{j-1,k-1}d_{j-1,k-1}+\alpha_{j-1,k}d_{j-1,k},\quad 0<k<2^{j-1}.$$

This equation corresponds to the lifting part of the wavelet construction introduced by Sweldens [25, 26]. This formula has no meaning if $k=0$ or $k=2^{j-1}$, but then we use (7) to find that $s_{j-1,0}=s_{j,0}$ and $s_{j-1,2^{j-1}}=s_{j,2^j}$. With the constraints we set, one can verify that the only choice of coefficients that verifies the mass conservation property (8) is

$$\alpha_{j-1,k}=\begin{cases}\frac{1}{4} & \text{if } 0<k<2^{j-1}-1\\ \frac{1}{2} & \text{if } k=0 \text{ or } k=2^{j-1}-1\end{cases} \tag{9}$$

The presented formulae let us perform one DWT step. Multiple steps can be performed by applying the same scheme to the scales $j-1$, $j-2$, ... At the end of the process, we have $2^{j-L}+1$ samples and 2^j-2^{j-L} details, where L is the number of performed DWTs, also known as the compression level. If most details are near-zeros, we can expect the compression ratio to increase as the compression level L increases.

This scheme can be expressed in a matrix form with (6). Because of our construction, the v vector has $2^{j-1}+1$ samples and 2^{j-1} details while the one presented in 2.1.1 has 2^{j-1} samples. For $j=3$, we obtain

$$A=\frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ -4 & 8 & -4 & 0 & 0 & 0 & 0 & 0 & 0\\ -2 & 4 & 5 & 2 & -1 & 0 & 0 & 0 & 0\\ 0 & 0 & -4 & 8 & -4 & 0 & 0 & 0 & 0\\ 0 & 0 & -1 & 2 & 6 & 2 & -1 & 0 & 0\\ 0 & 0 & 0 & 0 & -4 & 8 & -4 & 0 & 0\\ 0 & 0 & 0 & 0 & -1 & 2 & 5 & 4 & -2\\ 0 & 0 & 0 & 0 & 0 & 0 & -4 & 8 & -4\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

and

$$A^{-1} = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 6 & 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 8 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 4 & 6 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & 8 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 4 & 6 & 4 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 8 & -4 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 & 6 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}.$$

In the even rows of A, we can read the wavelet low-pass filter coefficients. In the odd rows, we read the high-pass filters. Only the filters at the beginning and at the end of the interval have small variations. In the middle they are constant. Away from the boundaries, we recognise the filters of the first order 5/3 biorthogonal wavelet transform presented in [7]. Those wavelets are attributed to Le Gall and Tabatabai [16].

The presented wavelet transform achieves first order compression, meaning that it filters out linear polynomials. It is possible to achieve order two compression by using the Cohen-Daubechies-Feauveau 9/7 wavelets [7]. However, increasing the order of the filtered polynomials is not necessarily significantly better in terms of compression ratio. In this work, we will only use the first order compression wavelets (LGT5/3).

(A) Original grayscale image. (B) Resulting grayscale image.

FIGURE 1. Transformation of a 2-dimensional image (left) after the application of 3 steps of the wavelet scheme (right). In the result (right), the samples are located in the upper-right corner of the image. The remaining coefficients represent details at different levels. Details colors are obtained with $\mathrm{mod}(x, 256)$, where x is the detail value. The resulting color ranges from 0 (black) to 255 (white).

This section presents a 1-dimensional wavelet transform. To extend it to N dimensions, we simply apply this scheme successively to each dimension. For example, in 2 dimensions (a matrix), we first apply the wavelet transform to all the rows and then to all the columns. Figure 1b shows the result of the application of 3 steps of the wavelet scheme to a 2-dimensional image (Figure 1a). We can recognise the original image in the samples. We can also recognise a sketchy version of the image in the different detail levels. In particular, we can see that the outline of the cat is well distinguishable in the details. We can see that the sought property is reached: the

parts of the image that are near-linear result in small coefficients (very dark or very bright colors), while the non-linear parts (outline) result in large coefficients.

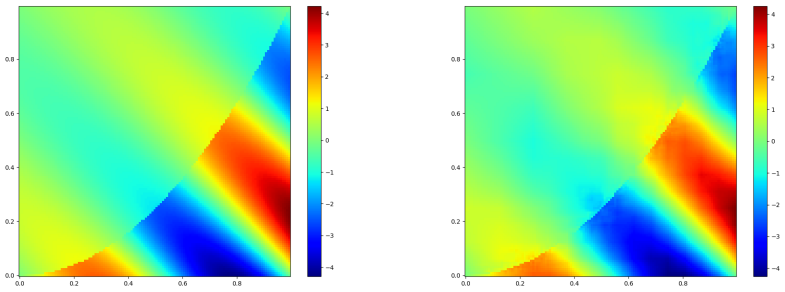
To demonstrate the ability of the scheme to preserve discontinuities, we present the result of the application of 6 steps of the wavelet scheme to a 2-dimensional regular function that has a discontinuity. The function is defined by:

$$f(x,y) = e^{x-y}\sin(2\pi(x+y)) \times \text{step}(y-x^2),$$

where

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 2 & \text{if } x < 0 \end{cases}.$$

Figure 2 shows the plots before and after the application of the wavelet scheme. All the details that are between -0.2 and 0.2 are nullified to create a loss. The reconstructed image (Figure 2b) is close to the original function. In particular, the discontinuity is well preserved. This is due to the fact that large details (induced by the discontinuity) are fully kept and let us perfectly reconstruct the original signal. We can also notice that artefacts appear near the discontinuity. These artefacts are typically due to the thresholding of a low-level detail. We verify experimentally that the mass conservation property is reached. The masses of the signal before and after the transformation are equal (up to the machine precision level), even when all the details are nullified. Incidentally, the number of non-null coefficients in the compressed form is 481 (out of 16641 in total), meaning that we can reasonably expect a compression ratio of the order of $\frac{481}{16641} \approx 34.596674$.



(A) Original function. (B) Reconstructed function.

FIGURE 2. Transformation of a 2-dimensional regular function with a discontinuity on a 129x129 grid (left) after the application of 6 steps of the wavelet scheme (right). The details that are between -0.2 and 0.2 are nullified to create a loss. The reconstructed image (right) is noticeably degraded. In particular, artefacts appear near the discontinuity.

2.2. **Thresholding**

In the thresholding step, the goal is to transform the near-zeros values into zeros depending on a threshold. This threshold depends on the scale j and is only applied to the details. In the case of an N-dimensional wavelet transform, the threshold depends on the N-tuple $(j_1, \ldots, j_N)$ formed by the scales of each dimension. We use a constant c to have a global control on the threshold. This constant represents the threshold value for the lowest scale (coarsest) details. We define 3 thresholding functions:

- **Constant thresholding**: the threshold is constant along all the scales. $(j_1, \ldots, j_N) \mapsto c$.
- **Accumulation thresholding**: the threshold is defined by $(j_1, \ldots, j_N) \mapsto c \times \alpha^{\sum -j_i}$ where α is a constant that we set to 2 and j_i are the scales for the N dimensions (in 2 dimensions, j_x and j_y for example). With this method, the coarser scales have a higher threshold and are, therefore, less likely to be nullified.

- **Capped thresholding**: the threshold is defined by $(j_1, \dots, j_N) \mapsto c \times \alpha^{max(j_i)}$. Again, we set α to 2. This method is similar to the accumulation thresholding, but it only takes into account the highest scale between each dimension.

In essence, these different methods aim at setting a fair threshold. The constant thresholding is the simplest one and assumes that nullifying an equally-valued detail on two different scales has the same impact. The accumulation thresholding and the capped thresholding try to take into account the fact that the coarser scales may have a larger impact on the final result. These different methods correspond to the type of loss that we want to minimize. To justify the use of one method over the other, we test them empirically. For this, we set an arbitrary maximum accepted error for a simulation and manually tune c to achieve this error. Then, the most efficient method is the one that achieves the best compression ratio, i.e. the one that nullifies the most details (for the same error). According to this test, the capped thresholding is the most efficient and will be the one used in the following experiments. The c constant will be referred to as the *threshold value* and can be viewed as the used threshold for the first level of details.

After the thresholding step, the small details are set to zero but still stand in the memory. Thus, the data size is not changed. An additional lossless compression step is necessary to effectively compress the data.

2.3. **Lossless compression**

In the context of GPU computations, the use of lossless data compression techniques is often explored. These are particularly relevant when the data need to be transferred between the CPU and the GPU.

For instance, Patel *et al.* propose a fast compression algorithm for GPU computations [20]. Their algorithm is based on the bzip2 compression algorithm. E. Sitaridi *et al.* develop a fast GPU decompression method based on the LZ4 method [9, 23]. F. Knorr *et al.* propose a fast lossless GPU compression method for floating-point scientific data known as ndzip-gpu [13]. This compression method typically achieves excellent compression throughput but worse compression rate.

Currently, many GPU compression algorithms are based on the use of the nvCOMP library [22]. This library has become widely used in the HPC community and provides a lossless compression/decompression framework that aims at being runtime-efficient. It is based on the use of the CUDA API and is compatible with the CUDA programming model. Multiple highly efficient compression algorithms are implemented in this library, such as LZ4, zStandard, or Bitcomp.

In the case of FD/FV/LBM simulations, the use of these generic lossless compressions would typically not be the most adapted as they would not take into account the spatial structure of the data. Moreover, it can be acceptable to use a lossy compression method, as long as we can verify that the induced loss does not impact the validity of the simulation. This is why we combine a Wavelet Lifting Scheme with a lossless compression algorithm.

In this work, we compare two different lossless compression schemes: a CSR (Compressed Sparse Row) method provided by cuSPARSE and an LZ4 method provided by the nvCOMP library.

These lossless compressions are performed after the previous wavelet transform step to achieve an effective compression of the data.

2.3.1. *Sparse Matrix Representation*

The Compressed Sparse Row (CSR) format, also known as the Yale format, is a widely used representation for sparse matrices. In this format, the non-zero elements of the matrix are stored in 3 arrays. The first array, V, contains the non-zero elements of a $m \times n$ matrix. The second array, COL, of the same size as V, contains the column indices of the non-zero elements. The third array, ROW, of size $m + 1$ contains the offset of the first non-zero element of each row. This representation can be used as a lossless compression technique since we anticipate a large number of zero values in the matrix. While the CSR format was not designed specifically for compression purposes, it is both well-known and efficient. Additionally, its compressed size is proportional to the number of non-zero elements, making it a consistent representation.

2.3.2. *LZ4*

LZ4 is a widely used lossless compression algorithm known for its high compression ratios and fast processing speeds on GPU. It has gained popularity due to its efficient CUDA implementation in the nvCOMP library, which was developed by Nvidia [22]. LZ4 is a byte-oriented algorithm that is designed to be fast and parallelizable. It was initially developed to perform well on CPUs and has since been optimized for use on GPUs [23]. The algorithm uses a block-based approach and compresses each block independently, with a configurable chunk size. The chunk size determines the size of the input data that is processed at once by the LZ4 algorithm. Usually, a larger chunk size implies a better compression ratio, but a slower compression speed. LZ4 is commonly used in data-intensive applications such as scientific simulations and big data analytics.

3. Practical GPU implemention

3.1. **Protocol**

We propose a numerical experiment to test the performance of our algorithm. The idea is to perform a two-dimensional LBM/FV simulation while introducing compression/decompression cycles between each time step.

3.1.1. *Structure of the data*

The simulation grid is divided into 8 subgrids: 2 subgrids in each of the 3 dimensions. This division in subgrids does not serve any computational purpose but forces the wavelet transform to be applied on a non-periodic space. Each subgrid has a logical size and a true size. The logical space represents the simulation space, meaning that each value corresponds to a fixed space in the simulation. The true size is the whole space of the subgrid. It contains the logical space and the ghost cells (also known as halo or overlap). The ghost cells are useful in a simulation because they duplicate the values of the neighbouring cells and avoid the need to explicitly communicate with the neighbouring subgrids. Keeping the ghost cells up to date, however, requires a synchronization phase between time steps. The synchronization process is illustrated in Figure 3.

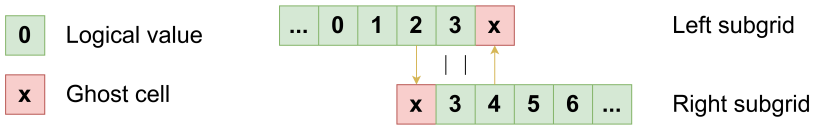


Figure 3. This schema shows the synchronization process in 1 dimension. The ghost cells are updated by copying the corresponding values of the neighbouring subgrids. One value is shared between both subgrids and does not need to be updated. The yellow arrows show which values are copied. In a real-case scenario, the data consist of floating-point values, rather than integers. In multiple dimensions, the synchronization process is performed successively in each dimension.

We chose a simulation grid of size 128x128x128. Each subgrid has a logical size of 65x65x65 and a true size of 67x67x67. The bordering values of the logical space are shared between multiple subgrids. In Figure 3, the shared value is the cell "3". As these values are not synchronized, the same FD/FV/LBM computation must be performed once for each subgrid. The reason for introducing this overlap is to keep the mass-conservation property of the wavelet transform. Indeed, one can verify that the mass conservation property equation (8) implies that:

$$\sum (s_{j,k} - T(s_{j,k})) \times \begin{cases} \frac{1}{2} & \text{if } k = 0 \text{ or } k = 2^j - 1 \\ 1 & \text{otherwise} \end{cases} = 0, \tag{10}$$

where $s_{j,k}$ is the k-th coefficient, j is the scale and T is the application of the compression pipeline: wavelet transform, thresholding, and inverse wavelet transform. This means that the total mass of the compressed interval is conserved regardless of the amount of information lost during the thresholding phase. By sharing a bordering value, we ensure that this exact mass conservation property is achieved for the entire grid.

3.1.2. *Transport simulation*

The first tested simulation is a 2D simplistic computation of the displacement of an arbitrary structure given by the following rules:

$$\begin{cases} f\_init(x,y) = 1 + e^{-30(x^2+y^2)} \\ f(x,y,t) = f\_init(x - \alpha t, y - \beta t). \end{cases} \tag{11}$$

In other words, $W = f \in \mathbb{R}^m$, $m = 1$, is solution of the transport equation

$$\partial_t W + \nabla \cdot \left(W \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right) = 0.$$

With the definition (2), this amounts to considering the conservation law with the flux

$$Q(W, N) = W(\alpha N^x + \beta N^y).$$

The f_init is the initial state and f is the state at time t. The initial state is a 2D Gaussian function centered in the middle of the grid (Figure 4a). This structure is displaced at a constant speed (α, β). The transport equation is solved using the scheme (5) with the standard upwind flux

$$Q(W_L, W_R, N) = W_L \max(\alpha N^x + \beta N^y, 0) + W_R \min(\alpha N^x + \beta N^y, 0). \tag{12}$$

where Δx is the size of the cells, CFL is the Courant-Friedrichs-Lewy number, $v_{max} = max(\alpha, \beta)$ is the maximum speed of the structure and $dt = \frac{\text{CFL} \cdot \Delta x}{v_{max}}$ is the duration of a time step. This scheme can be implemented as follows:

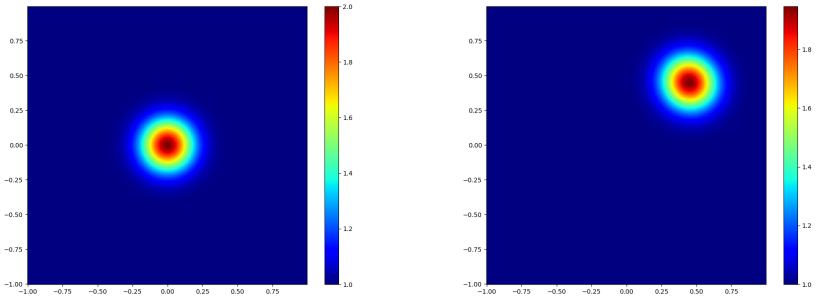
```
for i = 1 to n_x -2 do
    for j = 1 to n_y -2 do
        W_next[i][j] = W_now[i][j]
        for dir = 0 to 3 do
            W_next[i][j] -= dt/h * fluxnum(W_now[i][j],
                                           W_now[i+N[dir][0]][j+N[dir][1]],
                                           N[dir])
        end for
    end for
end for
```

Listing 1. Pseudo-code of the transport kernel

Where `fluxnum` is a function implementing the numerical flux and `N` is the array of the normal vectors of the cells defined in (3).

Note that indices i=0, j=0, i=n_x-1, and j=n_y-1 can be omitted as they are part of the ghost cells and will be overwritten by the neighbouring subgrids during the synchronization phase. It is a stencil computation, meaning that the next state of each cell (excluding the borders) is a function φ of its neighbors. Here, the stencil targets 5 cells: the current cell and its 4 neighbors.

Figure 4a shows f_init at t=0s. Figure 4b shows the exact solution at t=0.5s for $\alpha = 0.9$ and $\beta = 0.9$.



(A) t=0s (B) t=0.5s

FIGURE 4. This figure shows the initial state of the simulation (left) and the exact solution of the simulation at t=0.5s for $\alpha = 0.9$ and $\beta = 0.9$ (right). The original structure is displaced with a speed of (α, β). The color represents the value of the function.

3.1.3. *Compression*

To test the compression algorithm, we perform a compression/decompression cycle between each time step. The compression cycle is described in Section 2. We recall that the compression consists of (1) applying the wavelet transforms on the subgrids, (2) thresholding near-zeros, and (3) applying a lossless compression on the resulting data. The decompression process consists of performing the opposite operations in reverse order. A various number of successive wavelet transforms (1) can be applied, the limit being that the number of samples $2^j + 1$ must remain valid. We exclude the ghost cells from the wavelet transform because they are not part of the logical space, meaning that the information they contain can be rediscovered thanks to the neighbouring subgrids. The number of performed wavelet transforms is a parameter of the algorithm that we will refer to as the *compression level*. The threshold value (2) is manually set and corresponds to the c constant introduced in Subsection 2.2 A threshold of 0 implies that no data is nullified.

We test two lossless compressions (3): the *CSR* matrix format and the *LZ4* compression. The CSR conversion is performed using the *cuSPARSE* library. The LZ4 compression is performed using the *nvCOMP* library.

3.1.4. *Methodology*

The benchmark program has been written in CUDA and compiled with the nvcc compiler and the `-O3 -use_fast_math` flags. We run the program on an NVIDIA Tesla V100 GPU with 12GB of memory.

We refer to the following program parameters:

- the number of performed wavelet transforms (i.e. the compression level);
- the threshold from which the details are nullified;
- the used lossless compression (LZ4 or CSR);
- in the case of LZ4, the chosen chunk size. A lower chunk size leads to a lower compression rate, but a higher compression speed;
- the measured simulation time (related to the number of time steps and the grid size).

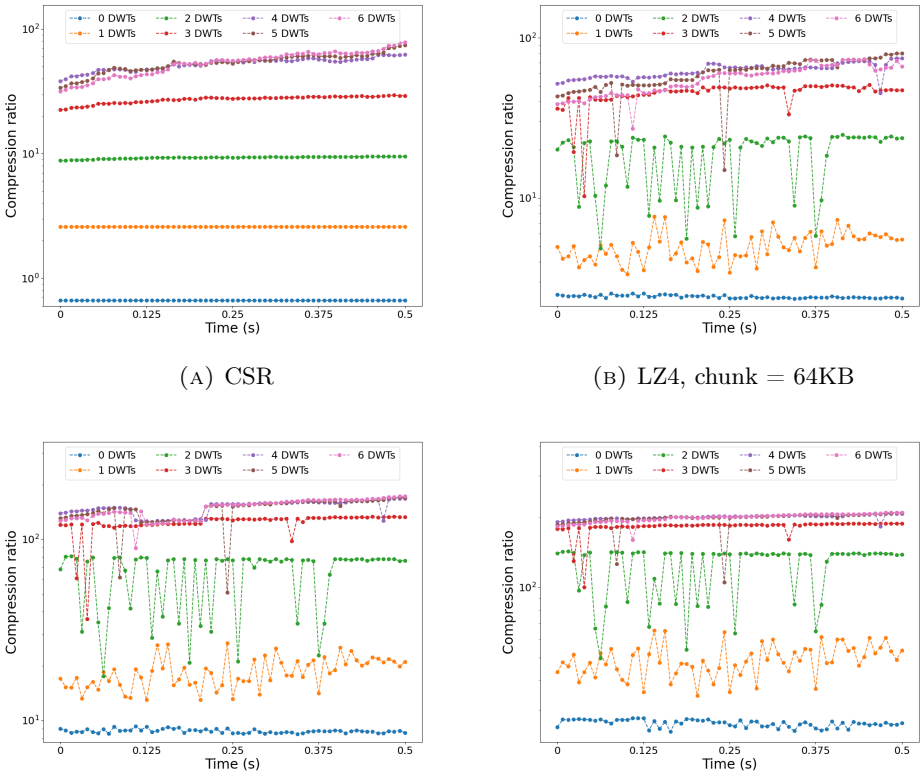
We set two measures of interest: the effective compression ratio and the quality of the simulation. The effective compression ratio is the data size after the compression divided by the initial data size. The quality of the simulation is measured by comparing the results of the simulation with the exact solution. It is measured with the L2 error against the exact solution at a given time step with the formula:

$$L2\_error = \frac{SizeX \times SizeY \times SizeZ}{NX \times NY \times NZ} \times \sum_{i=0}^{NX-1} \sum_{j=0}^{NY-1} \sum_{k=0}^{NZ-1} (f\_sim(i,j,k) - f\_exact(i,j,k))^2. \quad (13)$$

The primary objective of this study is to determine the effect of different parameters on the compression ratio and the quality of the simulation, which is discussed in detail in Sections 3.2.1, 3.2.2, and 3.2.3. In Section 3.2.4, the computational cost of the compression/decompression cycles is analysed, serving as a preliminary reference for the performance of our implementation.

3.2. **Results**

3.2.1. *Compression ratio during the simulation*



(A) CSR

(B) LZ4, chunk = 64KB

(C) LZ4, chunk = 256KB

(D) LZ4, chunk = 1MB

FIGURE 5. This figure shows the compression ratios at each time step of the simulation with different lossless compression methods. The different curves (compression levels) represent the number of performed wavelet transforms.

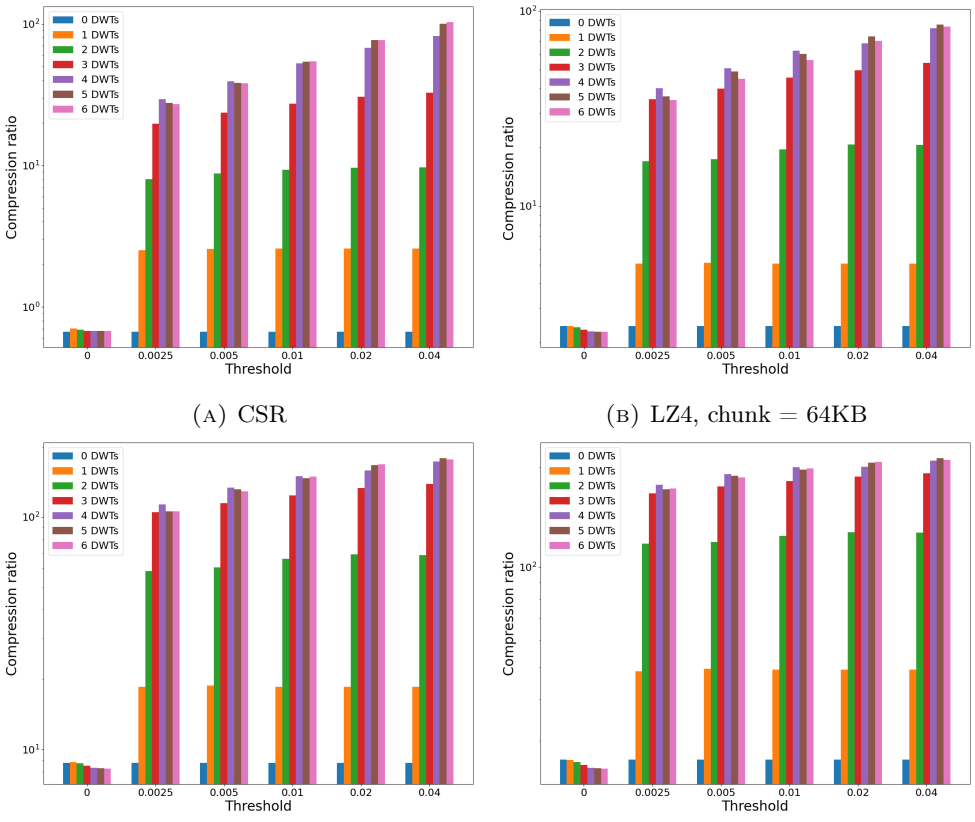
In this section, we show the compression ratios during the simulation with different lossless compression methods. The tested compression methods include CSR (Figure 5a), LZ4 with a chunk size of 64 KB (Figure 5b), LZ4 with a chunk size of 256 KB (Figure 5c), and LZ4 with a chunk size of 1 MB (Figure 5d). A threshold value of 0.01 is used, and $\alpha = 0.9$, $\beta = 0.9$, $CFL = 0.45$ are chosen for the numerical simulation. The compression ratio, which is defined as the size of the uncompressed data divided by the size of the compressed data, is not constant during the simulation and tends to increase as the simulation progresses. This trend becomes more pronounced as the compression level increases, indicating that the data become more easily compressible as the simulation progresses. This is likely because each time step allows for removing more and more details from the simulation. An expected observation is that the compression ratio tends to increase with

the compression level. Compression levels 5 and 6 are an exception to this trend but we can reasonably exclude them from the analysis, as they have too few sampling points to perform a reasonable DWT.

There are substantial differences between the CSR and the LZ4 compression methods. The CSR method produces smooth compression ratios over the simulation, while the LZ4 method produces more erratic compression ratios. For the CSR method, this is explained by the fact that the compression ratio is directly dependent on the number of non-zero values which have no reason to brutally change from one time step to another. For the LZ4 method, this is likely due to an inner mechanism of the LZ4 algorithm that makes it underperform in some time steps. The obtained compression ratios are comparable for the CSR method and the LZ4 method with a chunk size of 64 KB. However, the LZ4 method with chunk sizes of 256 KB and 1 MB both outperform the CSR method.

In terms of compression ratio, LZ4 with a chunk size of 1 MB is the best method. The CSR format is not designed to be a compression method. It is, therefore, not surprising that it can be outperformed. On the other hand, having a larger chunk size typically leads to increased compression ratios but at the cost of increased computation time. Finding the right balance between compression ratio and computation time is a challenge that we will address in the future.

3.2.2. *Impact of the threshold value on the compression ratio*



(A) CSR

(B) LZ4, chunk = 64KB

(C) LZ4, chunk = 256KB

(D) LZ4, chunk = 1MB

FIGURE 6. This figure shows the average compression ratio for different threshold values and lossless compression methods. The compression level is the number of performed wavelet transforms.

In this section, we aim to determine the impact of the threshold value on the compression ratio. We keep the previous simulation parameters: $\alpha = 0.9$, $\beta = 0.9$, and $CFL = 0.45$. We test the following threshold values: 0, 0.0025, 0.005, 0.01, 0.02, and 0.04. We show the results for 4 lossless compression methods: CSR (Figure 6a), LZ4 with a chunk size of 64 KB (Figure 6b), LZ4 with a chunk size of 256 KB (Figure 6c), and LZ4 with a chunk size of 1 MB (Figure 6d).

We observe that as the threshold value increases, the compression ratio generally increases as well. This trend is more pronounced at higher compression levels. At compression level 1, the compression ratio remains constant across non-zero threshold values, whereas at compression level 4, the compression ratio always increases with increasing threshold value. Additionally, we note that, up to compression level 4, the compression ratio tends to increase as the compression level increases. The four compression methods exhibit similar trends, although they differ in scale. CSR and LZ4 with a chunk size of 64 KB show similar performance, while LZ4 with a chunk size of 256 KB and 1 MB outperform the other two methods. With the 1 MB chunk size, the average effective compression ratio always reaches more than 100x if the threshold value and the compression level are greater than or equal to 0.0025 and 2, respectively.

We find that setting a non-null threshold value is crucial for achieving a high compression ratio. Furthermore, the first three compression levels have the most significant impact on the compression ratio in our case. We also note that while LZ4 compression without thresholding can already achieve a high compression ratio, the compression ratio significantly improves as the threshold value increases from 0 to 0.0025. This outcome is expected for CSR compression, where the compression ratio is directly related to the number of non-zero values. For LZ4 compression, thresholding the data increases the frequency of the "zero" symbol, resulting in a higher compression ratio. Nonetheless, LZ4 can still achieve a high compression ratio even without thresholding the data, with the compression ratio being greater than 2x for all chunk sizes. The 256 KB chunk size achieves an average compression rate of nearly 9x, while the 1 MB chunk size nearly achieves a compression rate of more than 25x, both with no pre-processing (i.e., threshold value and compression level of 0).

3.2.3. *Quality of the simulation*

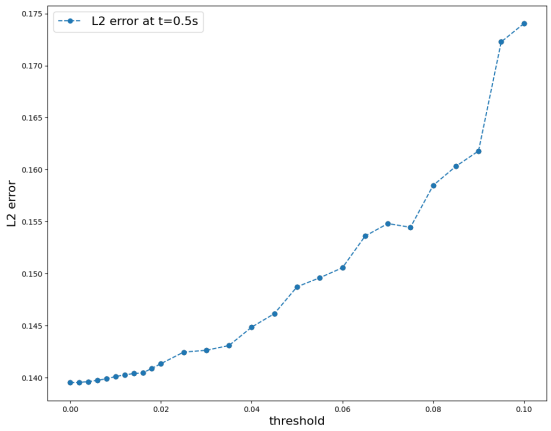
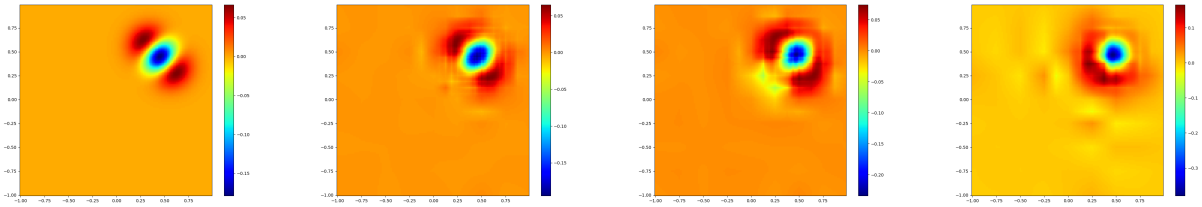


FIGURE 7. This figure shows the L^2 error (formula 13) for different threshold values at compression level 4 and t=0.5s, with $\alpha = 0.9$, $\beta = 0.9$, and $CFL = 0.45$.

The accuracy of the simulation is evaluated by comparing it with the exact solution. To measure the error, we use the L^2 norm, which is calculated at the end of the simulation when the time is 0.5 seconds. We plot the global L^2 error depending on different threshold values (Figure 7), and the results show that the error tends to



(A) threshold=0 (B) threshold=0.01 (C) threshold=0.02 (D) threshold=0.05

FIGURE 8. These plots show how the error is distributed over the domain at $t = 0.5s$ and with a compression level of 4 for different threshold values.

increase as the threshold value increases. This is expected, as the thresholding phase introduces an error in the signal reconstruction. This increase is slow for threshold values below 0.02. For higher values the L^2 increases faster. This indicates that the compression error is negligible, compared to the numerical scheme errors, for threshold values below 0.02.

We also plot the difference between the simulation result and the exact solution for each point in the domain in (Figure 8). This plot confirms the above interpretation: the compression error starts to dominate the scheme error for a threshold greater than 0.02.

It is clear that the adequate threshold value will vary depending on the specific application. This is a real difficulty of the approach. However, one of the advantages is that the induced error can be controlled by the user through the threshold value. This study shows that our lossy compression approach can be tuned to have little impact on the simulation results.

3.2.4. *Computational cost*

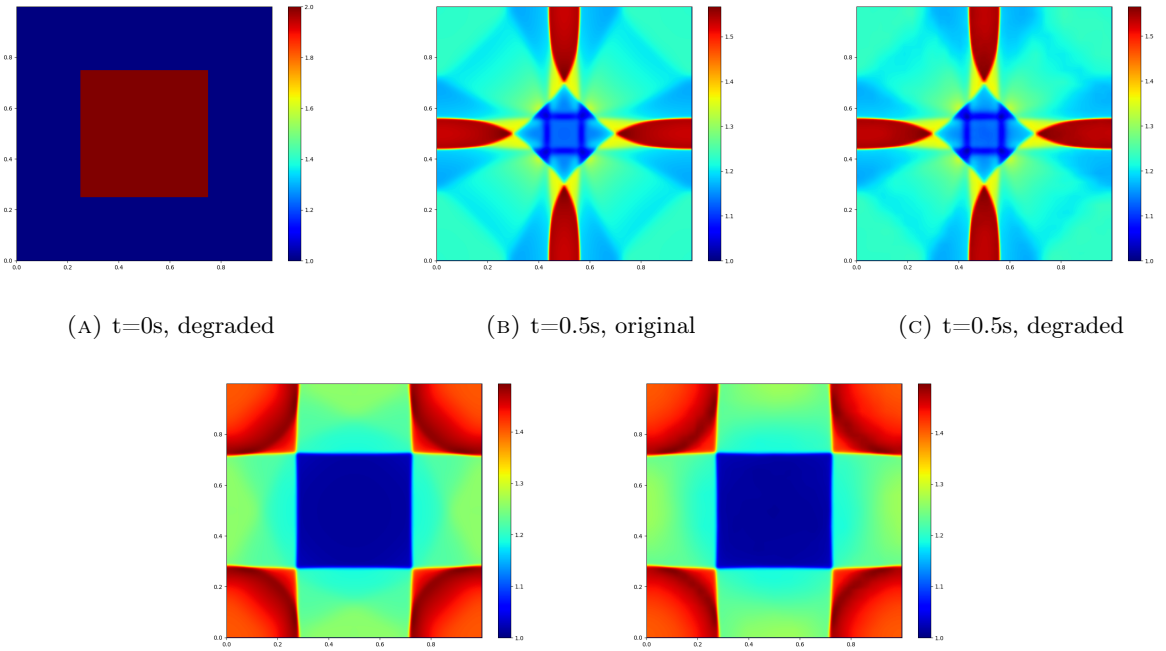
It is important to note that the compression and decompression kernels presented in this study are not intended to reach the highest level of optimization. Rather, the purpose of evaluating the computational cost is to provide a reference point for the reader and to give an estimate of the potential impact of our method on a real simulation. As such, the results should be considered as an indicative measure of the performance and not as an absolute representation of the optimization level achievable with further fine-tuning.

To evaluate the computational cost of the compression pipeline on a more computationally intensive simulation, we use a Godunov scheme to solve a shallow water model. The shallow water model is defined by

$$W = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \quad Q(W, N) = \begin{pmatrix} h(uN^x + vN^y) \\ hu(uN^x + vN^y) + \frac{1}{2}gh^2N^x \\ hv(uN^x + vN^y) + \frac{1}{2}gh^2N^y \end{pmatrix}, \tag{14}$$

where h is the water level, (u, v) the horizontal velocity vector and $g = 9.81\text{m/s}^2$ the gravitational acceleration. In the following simulations, we use the Godunov numerical flux, based on exact Riemann solvers (we refer, for instance, to [17] for the details). At $t = 0$, there is a 0.5-meter square where the water level is $h = 2$ meters high and the rest of the domain is at $h = 1$ meter (Figure 9a). We run the simulation on a 1025x1025 grid and perform 30390 time steps, which correspond to a simulation time of 10 seconds. We use a compression level (number of DWTs) of 6 and a threshold value of 10^{-5}. We only show the results for the CSR lossless compression because the LZ4 compression provided poor compression ratios due to the method we used to implement fast wavelets (more details on this in the following). The lossless CSR compression is implemented using the `cuSPARSE` library.

Table 1 displays execution times of various kernels used for the Godunov simulation. Rows in the table indicate the total time spent in a kernel. The `total GPU time` row displays the total GPU computation time. The `time_step` row represents the execution time of a time step. The `wav_...` kernels correspond to the DWT



(A) t=0s, degraded (B) t=0.5s, original (C) t=0.5s, degraded

(D) t=1s, original (E) t=1s, degraded

FIGURE 9. These plots show the results of the simulation with no compression (original) and with the DWT (degraded, double precision). The threshold value is 10^{-5}.

Kernel	Execution time (s)					
	no compression		only wavelets		wavelets + CSR	
	single	double	single	double	single	double
total GPU time	19.6138	125.281	51.5696	168.010	64.5597	188.879
time_step	19.6138	125.281	19.6739	126.217	19.6581	126.439
wav_x_compress	∅	∅	4.08561	5.39602	4.09886	5.39384
wav_step_y_compress_samples	∅	∅	5.24588	7.50239	5.25510	7.50315
wav_step_y_compress_details	∅	∅	6.66140	8.09396	6.67206	8.09803
wav_x_decompress	∅	∅	3.47536	4.85359	3.48760	4.84650
wav_step_y_decompress_samples	∅	∅	7.23108	8.42857	7.37205	8.55370
wav_step_y_decompress_details	∅	∅	5.19632	7.51815	5.21071	7.55852
dense_to_csr	∅	∅	∅	∅	10.9194	11.8537
csr_to_dense	∅	∅	∅	∅	0.51037	0.52552
cusparseParseDenseByRows_kernel	∅	∅	∅	∅	0.12544	6.32183
other_cusparse_kernels	∅	∅	∅	∅	1.25000	1.785477
overhead	0%	0%	+162.12%	+33.11%	+228.41%	+49.38%

TABLE 1. Execution times of the different kernels for the Godunov simulation in single and double precision.

kernels which include wavelet compression and decompression operations along the X and Y axes. The wav_y kernels are split into two parts and perform only one step of the DWT. The wav_x kernels perform the entire DWT along the X-axis in one kernel. The `dense_to_csr` and `csr_to_dense` kernels refer to the conversion between dense and compressed sparse row representations. The `cusparseParseDenseByRows_kernel` is an internal cuSPARSE kernel, and the remaining internal cuSPARSE kernels are grouped in the `other_cusparse_kernels` row. Finally, the `overhead` row shows the overhead percentage introduced by the compression method.

LZ4 compression was not tested on this simulation because it provides poor compression ratios. This is because the optimized DWT kernels used in this study are different from the ones used in the previous sections, where samples are grouped in a corner of the matrix and the details are stored in the remaining cells. This is convenient for the LZ4 compression because the details tend to be contiguous in memory, leading to a higher chance of finding matches in this area. In the optimized DWT kernels, samples are distributed evenly in the matrix and the details are stored in the remaining cells, which makes it harder for the LZ4 compression to find matches.

We measured 4.652 TFLOPS/s for the single precision simulation and 728.227 GFLOPS/s for the double precision simulation by accessing the performance counters. The peak performance of the V100 GPU according to the NVIDIA specification is 14 TFLOPS/s for single precision and 7 TFLOPS/s for double precision. The observed gap between our measured values and the peak performance specified by NVIDIA can be explained by the fact that some optimizations, such as the flush-to-zero, do not appear in the double precision version. This could be due to the compute capability of the V100 GPU not supporting these types of optimizations in double precision. We verified that the single precision version without the math optimizations has a ratio of approximately 2:1 compared to the double precision version.

The compression pipeline demonstrates impressive compression rates, with an average compression rate of **x31.8** for the single precision simulation and **x42.6** for the double precision simulation. The overhead introduced by the compression pipeline is **228.41%** for the single precision simulation and **49.38%** for the double precision simulation. The overhead introduced by the compression kernels is currently a bottleneck, particularly for the single precision simulation. However, the double precision simulation shows that the compression pipeline can be relevant, with a 50% overhead that is acceptable given the extremely high compression ratio of more than x40. It is noteworthy that the overhead could be reduced by a factor of at least 2 by investing efforts in optimizing the compression kernels. Our model suggests that the compression ratio can be even higher in a 3D case, with an expected ratio of around 160 for the single precision simulation and 250 for the double precision simulation.

The quality of the result can be evaluated by comparing the original uncompressed results to those obtained through DWT compression/decompression at each time step, as illustrated in Figure 9. While minor details may be lost in the compressed version, the general structure remains intact. The level of compression can be fine-tuned by adjusting the threshold value, but finding the optimal value can be challenging due to the artifacts introduced during detail thresholding and subsequent compression.

Despite the current overhead, our preliminary results are highly encouraging, as they showcase the potential of our compression pipeline for large-scale simulations. Future work will focus on further optimizing the compression kernels and exploring methods for decompressing only a limited number of subgrids at once, allowing for the surpassing of GPU memory limitations.

Conclusions

In this work, we have adapted a compression scheme that combines a discrete wavelet transform with a lossless compression algorithm for optimizing the memory management of numerical simulations on regular grids. This algorithm is designed to allow a controlled loss of information while ensuring conservation of the global mass throughout the simulation. We have shown that in a 1025x1025 grid, 2-dimensional compression can achieve a ratio of approximately 200x on a simple transport equation. On a more complex simulation based on the shallow water equations, we have shown that the compression ratio can reach more than 40x. This is substantial and can help to fit large simulations in the GPU memory. With this approach, simulation schemes

that are traditionally thought to be unadapted to GPU computing because of their high memory requirements could become feasible without changing the FD/FV/LBM kernels. The downside of this approach is that the compression is lossy. However, we have shown that only a small loss of information is necessary to achieve a high compression ratio. In addition, this loss can be controlled by the user, with the threshold value. We have shown that on a double precision shallow-water simulation, the compression/decompression pipeline only slows down the simulation by less than 50%. This slowdown is the only additional cost of our method and it can reduce the memory requirements by multiple orders of magnitude. To be able to run an execution where the simulation grid actually exceeds the GPU memory, only a few subgrids should be decompressed in global memory at a time. In our current implementation, this is not possible because the synchronization between the subgrids assumes that the neighbors are also decompressed. This is why our next work will consist of running a large-scale simulation on a GPU with a memory that is not large enough to fit the whole simulation grid. Additionally, we plan to improve the efficiency of our compression pipeline to take full advantage of the GPU architecture.

References

[1] R. Abgrall. Multiresolution representation in unstructured meshes. *SIAM journal on numerical analysis*, 35(6):2128–2146, 1998.

[2] H. Astsatryan, A. Kocharyan, D. Hagimont, and A. Lalayan. Performance optimization system for hadoop and spark frameworks. *Cybernetics and Information Technologies*, 20(6):5–17, 2020.

[3] T. Bellotti, L. Gouarin, B. Graille, and M. Massot. Multiresolution-based mesh adaptation and error control for lattice boltzmann methods with applications to hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 44(4):A2599–A2627, 2022.

[4] T. Beneš, M. Bartík, and P. Kubalík. High throughput and low latency lz4 compressor on fpga. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019.

[5] K. Brix, S. S. Melian, S. Müller, and G. Schieffer. Parallelisation of multiscale-based grid adaptation using space-filling curves. In *ESAIM: proceedings*, volume 29, pages 108–129. EDP Sciences, 2009.

[6] C. Burstedde, D. Calhoun, K. T. Mandli, and A. R. Terrel. Forestclaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253–262. IOS Press, 2014.

[7] A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 45(5):485–560, 1992.

[8] A. Cohen, S. Kaber, S. Müller, and M. Postel. Fully adaptive multiresolution finite volume schemes for conservation laws. *Mathematics of Computation*, 72(241):183–225, 2003.

[9] Y. Collet. Lz4 - extremely fast compression. `https://github.com/lz4/lz4`.

[10] R. Eymard, T. Gallouët, and R. Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[11] A. Harten. Multiresolution representation of data: A general framework. *SIAM Journal on Numerical Analysis*, 33(3):1205–1256, 1996.

[12] T. Y. Hou and P. G. LeFloch. Why nonconservative schemes converge to wrong solutions: error analysis. *Mathematics of computation*, 62(206):497–530, 1994.

[13] F. Knorr, P. Thoman, and T. Fahringer. ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[14] H. P. Langtangen and S. Linge. *Finite difference computing with PDEs: a modern software approach*. Springer Nature, 2017.

[15] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, et al. Palabos: parallel lattice boltzmann solver. *Computers & Mathematics with Applications*, 81:334–350, 2021.

[16] D. Le Gall and A. Tabatabai. Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques. In *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*, pages 761–764. IEEE, 1988.

[17] R. J. LeVeque et al. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[18] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[19] S. Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

[20] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[21] M. Rabbani. JPEG2000: Image Compression Fundamentals, Standards and Practice. *Journal of Electronic Imaging*, 11(2):286, 2002.
[22] N. Sakharnykh, D. LaSalle, and B. Karsin. Optimizing data transfer using lossless compression with nvidia nvcomp, 2020.
[23] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 242–247, 2016.
[24] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 2001.
[25] W. Sweldens. Lifting scheme: a new philosophy in biorthogonal wavelet constructions. In *Wavelet applications in signal and image processing III*, volume 2569, pages 68–79. International Society for Optics and Photonics, 1995.
[26] W. Sweldens and P. Schröder. Building your own wavelets at home. In *Wavelets in the Geosciences*, pages 72–107. Springer, 2000.
[27] Z. Yu and L.-S. Fan. An interaction potential based lattice boltzmann method with adaptive mesh refinement (amr) for two-phase flow simulation. *Journal of Computational Physics*, 228(17):6456–6478, 2009.