Using the Discrete Wavelet Transform for Lossy On-the-Fly Compression of GPU Fluid Simulations

Clément Flint, Atoli Huppé, Philippe Helluy, Bérenger Bramas, Stéphane Genaud

Using the Discrete Wavelet Transform for Lossy On-the-Fly Compression of GPU Fluid Simulations

Clément Flint[1,2,3], Atoli Huppé[1,2], Philippe Helluy[3,4], Bérenger Bramas[2,3], and Stéphane Genaud[1,2]

[1]ICPS team, Université de Strasbourg, Grand Est, France
[2]CAMUS team, Inria, Grand Est, France
[3]TONUS team, Inria, Grand Est, France
[4]IRMA, CNRS, Université de Strasbourg, Grand Est, France

Abstract

[Abstract] High-performance computing in fluid dynamics frequently confronts substantial memory demands, especially in large-scale applications. Data compression techniques can alleviate these memory constraints, but introduce new challenges. This paper introduces an innovative on-the-fly low-overhead lossy compression technique tailored for GPU-based fluid simulations, utilizing the Discrete Wavelet Transform (DWT). Our approach significantly diminishes memory requirements, achieving up to a 10-fold long-term reduction on a D3Q27 simulation, while minimally impacting the simulation accuracy. The methodology is built around careful design choices to achieve a satisfactory compression ratio/speed trade-off. It effectively maintains mass conservation and accurately preserves essential discontinuities in simulations. Extensive testing with a D3Q27 Lattice-Boltzmann Method (LBM) simulation on a single GPU has shown that large-scale grids can be processed with minimal impact on the simulation accuracy and acceptable compression times. This compression technique demonstrates a robust capability to handle memory limitations in fluid simulations, opening the door to more complex and larger-scale simulations.

1 Introduction

Fluid simulations are essential tools in many scientific and engineering fields, aiding in the understanding of complex fluid dynamics. The Lattice-Boltzmann Method (LBM) is particularly selected for incompressible flows and low Mach aerodynamics, not only due to its inherent suitability for these applications but also because of its parallel nature, which aligns well with modern computing technologies like GPU computing [38]. Several books have been written on the subject, see for instance [51, 26] for a full history. This method typically uses a regular grid to represent space, allowing for necessary refinements to capture flow details accurately. However, such a data structure demands significant memory, particularly in three-dimensional simulations, presenting a notable challenge for large-scale applications [40, 27].

In grid-based CFD simulations, memory constraints are typically addressed using either a multi-level or a stencil-based approach. Multi-level methods partition space into blocks of varying resolutions, focusing on regions of interest (ROIs) for higher detail and using coarser resolutions elsewhere to save memory (or computations). Hence, the memory requirements are reduced without the need for an explicit compression step. Despite their efficacy, these methods can be challenging to implement and may interfere with numerical schemes. This is why our work focuses on stencil-based methods, which are more straightforward to implement and are more compatible with existing numerical schemes. These methods utilize a uniform data structure across the simulation domain, processing it in manageable blocks. The memory requirements of these methods can become a bottleneck, especially in three-dimensional simulations, where the memory requirements grow cubically with the grid size.

A natural idea is to compress the data to fit within the available memory, but this introduces new challenges.

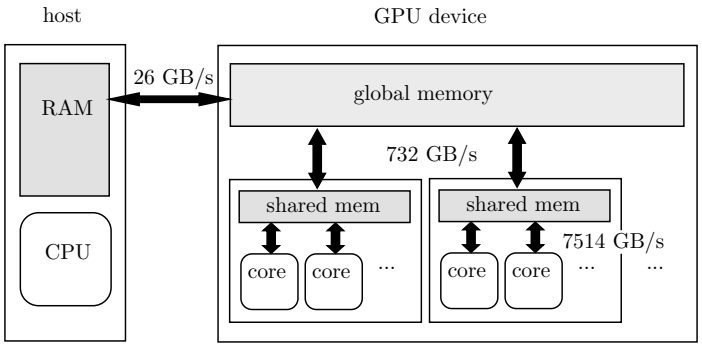


Figure 1: Schematic representation of the memory setup of a GPU. The provided throughputs have been measured with *gpumembench* [25] and *nvbandwidth* [37] on a P100 GPU. They are only indicative and can vary depending on the used hardware/software. The bandwidth of the memory transfers between the host and the GPU device is significantly lower that what can be achieved within the GPU. The main GPU memory (DRAM) is slower than the shared memory (SRAM), but has a much larger capacity and can be accessed by all the GPU cores. The shared memory is a block-level memory and can only be accessed by the cores of the same block.

The adoption of compression in numerical simulations is generally driven by two main needs: overcoming the memory capacity constraints of the hardware and accelerating simulations by utilizing computational resources more efficiently. Figure 1, illustrates the various levels of memories within the context of GPU programming and their typical bandwidths. Visualizing the memory setup in this manner helps to understand the different possible bottleneck scenarios. While some simulations may accommodate slower compression methods if it is the only option to fit within the hardware memory limits, others might focus on reducing the overhead of the compression process.

Our work focuses on striking a balance between compression time and compression ratio. The idea is to integrate compression/decompression steps within the simulation loop, which is known as *in situ* or *on-the-fly* compression. The goal is to simulate grids that would not fit in the GPU memory without compression and to minimize the impact of the compression on the overall simulation time. Our compression scheme is based on the Discrete Wavelet Transform (DWT), which is known for its high compression ratios and its ability to handle discontinuities in data. We ensure that the overall compression scheme remains exactly mass conservative, but allows for some loss in the compression process. The compression process is performed on-the-fly and tuned for GPU architecture.

In this paper, we explain our compression method, discuss why certain design choices were made, and assess its performance. Our tests demonstrate that our method allows for the simulation of grids that would not fit in the GPU memory without compression. We also provide insights into the impact of the compression on the overall simulation time. Depending on the configuration, we can expect the overall scheme to be between 2 and 3 times slower (although this measure can be disputed in cases where the scheme could not be run without compression). Given the significant reduction in memory requirements, this trade-off is acceptable in many scenarios, especially in cases where CPU-GPU transfers of the whole data are required to fit the simulation in the GPU memory. Overall, our method is a promising solution to improve the memory efficiency of large-scale CFD simulations.

This paper is organized as follows. In Section 2, we present the related work on data compression in fluid simulations. In Section 3, we describe the compression scheme, focusing on the Discrete Wavelet Transform. In Section 4, we present different results obtained with our compression scheme on a 3D 27-velocity Lattice-Boltzmann Method (D3Q27) simulation. Finally, in Section 5, we conclude and discuss future work.

2 Related Work

The use of data compression in scientific computing has been a topic of interest for several years. Cappello *et al.* conducted a thorough study to assess the relevance of using lossy

compression in scientific computing [9]. They identify use cases where lossy compression is advantageous, particularly when it comes to addressing memory bandwidth and storage limitations of computational hardware. In scenarios where memory bottlenecks significantly constrain the simulation, employing compression to alleviate these bottlenecks can be more advantageous than the overhead introduced by the compression process.

The idea of incorporating data compression in CFD simulations has been explored in several studies that we will survey in the following sections. We will refer to the concept of entropy, which is a measure of the information content of a message, as defined in *information theory* [49]. Because CFD data are not random, the entropy of the signal can be reduced by making assumptions about the data structure. The scientific literature on this topic differs from that of general data compression in that it aims to find the best assumptions on the signal for a given class of simulations to achieve the best compression ratio and/or compression speed. In Section 2.1, we present a general overview of data compression in fluid simulations. Then, in Section 2.2, we present the current state of the art in the use of the Discrete Wavelet Transform in CFD data compression.

2.1 Data compression in fluid simulations

Fluid simulations often generate massive datasets, necessitating advanced data compression techniques to address memory and computational challenges. Compression techniques can be classified by several criteria, such as the nature of compression (lossy/lossless), processing mode (streaming/offline), the employment of predictive models, reliance on floating-point representations, and specific spatial structures considerations. Such categorizations facilitate a better understanding and comparison of different techniques within fluid simulations.

Lossless compression methods, being reversible, preserve the integrity of data, ensuring no impact on the physics simulation results. Conversely, lossy compression leads to some information loss. While these techniques can achieve higher compression ratios, the design of lossy compression must be meticulous to maintain result accuracy. One strategy for managing data loss in simulations is to identify regions of interest (ROIs) and permit data loss primarily in areas outside these specified regions [35]. Machine learning is an option for achieving lossy compression but offers limited guarantees on the accuracy of the results. It has, however, demonstrated potential for in situ visualization of CFD data [32].

Streaming compression methods, designed for real-time data processing, are preferred in large-scale simulations due to their reduced memory footprint. A rudimentary example of compression is the mixed-precision representation. This method provides limited compression ratios, while considerably impacting accuracy, rendering it less effective compared to other techniques [27]. A more effective approach is to use a generic compression algorithm that is known to be efficient on the used hardware (here, GPUs) [46, 30, 50, 24, 56]. This approach benefits from incorporating domain-specific knowledge, which reduces the entropy of the data [28]. Typically, the streaming compression approach encompasses:

- **Predictor**: Estimates data from previously encoded points.
- **Difference Operator**: Computes the difference between the predicted and actual value, producing the *residual*.
- **Residual Coder**: Encodes residuals for compression, often using entropy coding.

In the context of fluid simulations, innovations in streaming data compression often originate from enhancements in these components [14, 43, 29]. Using prediction/difference pretreatments aims to reduce data entropy, making it more compressible. By allowing one of these components to be lossy, it becomes possible to further reduce entropy, often by setting a threshold and encoding residuals that exceed it.

Transitioning from streaming to offline compression, we explore techniques that process data in chunks or entirely. These techniques often achieve superior compression ratios but can demand more memory and sometimes entail higher computational costs. The Lorenzo predictor, which uses neighbors across N dimensions for data prediction, serves as a notable example of offline compression for CFD data [21, 31]. Though it is categorized as offline due to its multidimensional data access, with the right implementation, its memory usage can remain low. Other techniques that employ multidimensional prediction exist as well [16]. While these approaches primarily focus on compression speed, other methods

aim to achieve higher compression ratios. The Discrete Wavelet Transform (DWT), known for its high compression ratios with CFD data, exemplifies this approach. The subsequent section discusses the Discrete Wavelet Transform, its role in CFD data compression, and reviews relevant studies on the topic.

2.2 Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) is widely recognized across various fields, especially in CFD data compression [34, 11, 1, 2]. In this context, wavelets with compact support, biorthogonality, and a design tailored for a multi-resolution approach are favored. These attributes enable the DWT to process both frequency and spatial data efficiently. DWT decomposes an input signal into two main components: the approximation, which captures low-frequency information, and the detail, representing high-frequency nuances. This decomposition is performed iteratively, with each level of the DWT capturing a different frequency band. At the end of this process, the low details can be discarded, hence introducing loss, and fed to an entropy coder to achieve high compression ratios.

Researchers have extensively explored the use of DWT in CFD data compression [55, 23, 47]. These investigations establish the foundational knowledge supporting the use of DWT in CFD data compression. Because of its significant memory demand, DWT is not typically utilized as a standalone compression method. Instead, it serves as a tool to manage different resolution levels in data, optimizing computational resource allocation. Prominent multi-level methods, such as AMR [3] and multigrid methods [6], share this concept but do not incorporate the DWT directly. Other methods explicitly integrate different DWT levels within their framework [18, 44, 45].

One foundational study by Cohen *et al.* [11] conducts the CFD computations directly on the adaptive wavelet structure. They perform a careful mathematical analysis showing that this approach is almost optimal in terms of memory occupation and algorithm complexity. This optimality is obtained thanks to the excellent compression and approximation properties of the wavelet transform and also to its suitability to hyperbolic conservation laws, where local perturbations propagate at finite speed. However, this mathematical analysis is rather theoretical and does not take into account the very irregular structure of the sparse wavelet representation. In practice, handling this structure generates an unacceptable overhead on modern GPUs, because of non-coalescent memory access. In this paper, we propose a more pragmatic approach which harnesses the compression rate of the DWT, but keeps as much as possible the very efficient memory pattern of the LBM stencil pattern.

The use of wavelets in explicit data compression is also very common. However, traditionally, the considerable overhead of DWT has made these methods more suitable for storing or visualizing results, rather than for direct compression of simulation data [5]. Many of these techniques are inspired by or based on the JPEG2000 standard [17]. The advent of modern GPUs, with their architectural improvements, marks a shift in this domain. These advancements enable more efficient use of DWT, making it a viable option for in situ compression in performance-critical simulations [41]. Our research aligns with this trend, aiming to leverage the DWT for in situ compression of CFD data on GPUs. In a previous work, we have designed a DWT-based compression scheme for CFD data, with a focus on the numerical aspects of the method [15]. In this work, the compression algorithm is improved to lower its computational overhead and to ensure that it indeed leads to effective memory savings and acceptable compression times.

3 Description of the Compression Scheme

This section provides details on our compression method. We first briefly describe the Discrete Wavelet Transform (DWT) as a general tool and then present the wavelet scheme in Section 3.1. Then we explain how we use the DWT representation to design the compression scheme in Section 3.2.

3.1 Discrete Wavelet Transform

3.1.1 Basics of the Wavelet Transform

The Discrete Wavelet Transform (DWT) is a variant of the Continuous Wavelet Transform (CWT) tailored for sampled signals. The main concept is to represent a signal using a combination of wavelets. These wavelets are functions that capture variations in the signal.

For a comprehensive understanding of wavelet theory and its applications, the book by Mallat [33] is a valuable resource. In our research, we primarily use the Battle and Lemarié (BL) wavelets [12]. These wavelets, often referred to as the 5/3 biorthogonal wavelets, excel at first-order compression. This means they can effectively filter out local linear polynomial behavior, making them suitable for diverse applications like image and audio compression.

3.1.2 Building Wavelets for Non-Periodic Signals

Many standard wavelets are constructed for signals that span the entire real line or are periodic. However, we often deal with non-periodic signals, such as in realistic fluid simulations. Such signals require specialized wavelets, in particular at their boundaries. Our choice of wavelets is designed to filter out linear polynomials. This choice is based on empirical evidence that linear filtering usually offers satisfactory balance between compression ratio and compression speed. The linear filtering property is referred to as the *vanishing moment of order 1* in wavelet theory. In this section, we explain how to construct the wavelet scheme for a 1-dimensional non-periodic signal, but the construction can be extended to higher dimensions.

Let f, defined on $[0, 1]$, be the non-periodic signal on which we want to apply the DWT and J be the sampling scale. We begin with $2^J + 1$ sampling points:

$$x_{J,k} = k2^{-J}, \quad 0 \leq k \leq 2^J, \quad J \geq 0. \tag{1}$$

For a given scale J, the signal is represented using $2^J + 1$ points. Let us point out that at the coarsest scale $J = 0$ the signal is represented by its values at the two boundary points $x = 0$ and $x = 1$. These values will never be changed by the compression algorithm. This is slightly different from what is done usually for wavelet compression of periodic data. We now define the wavelet coefficients $s_{j,k}$ and $d_{j,k}$ to refer to the approximation and detail coefficients at scale j, respectively. At the finest scale $j = J$, the approximation coefficients are the samples of the signal f:

$$s_{J,k} = f(x_{J,k}), \quad 0 \leq k \leq 2^J. \tag{2}$$

Given approximation and detail coefficients at scale $j + 1$, the approximation and detail coefficients at scale j are found by applying the DWT transform:

$$s_{j,k} = T_{\mathrm{j,k,approx}}(s_{j+1,0}, s_{j+1,1}, \ldots, s_{j+1,2^{j+1}}), \quad 0 \leq k \leq 2^j \tag{3}$$

$$d_{j,k} = T_{\mathrm{j,k,detail}}(s_{j+1,0}, s_{j+1,1}, \ldots, s_{j+1,2^{j+1}}), \quad 0 \leq k \leq 2^j - 1 \tag{4}$$

for $0 \leq k \leq 2^j$, where T_{approx} and T_{detail} are transforms corresponding to the approximation and detail parts of the DWT, respectively. This lets us represent the signal f at different scales j by applying successive DWT transforms on the approximation coefficients $s_{s,k}$. The goal is now to define a reversible transform that satisfies a set of constraints.

3.1.3 Description of the lifting scheme

For constructing our DWT, we follow the so called approach of the "lifting scheme" introduced by Sweldens in [52, 53]. At any scale, the first and last samples are fixed as $x_{j,0} = 0$ and $x_{j,2^j} = 1$. These points are intentionally chosen to have even k indices and will always correspond to approximation coefficients. In our setting, at scale $j \geq 1$, there are $2^{j-1} + 1$ even indices and 2^{j-1} odd indices. This differs from usual wavelet constructions and is designed to handle the borders of the interval more effectively.

Given a scale $j = j_0 \geq 1$, we expect the approximation coefficients $s_{j_0,k}$ to be close to the signal f at the sampling points $x_{j_0,k}$:

$$s_{j,k} \approx f(x_{j,k}). \tag{5}$$

We impose them to be exactly equal at the boundaries:

$$s_{j,0} = s_{j+1,0} = f(0), \quad s_{j,2^j} = s_{j+1,2^{j+1}} = f(1). \tag{6}$$

If the signal f is locally almost linear (at the scale j), it is also expected that the odd samples satisfy:

$$s_{j,2k+1} \approx \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1} - 1. \tag{7}$$

As we want to filter out linear polynomials, we set the detail coefficients at scale $j-1$ accordingly:

$$d_{j-1,k} = s_{j,2k+1} - \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1} - 1, \tag{8}$$

If our assumptions are correct, the detail coefficients $d_{j-1,k}$ should be close to zero. One can notice that this operation is reversible, as it allows for the exact reconstruction of the original odd values at scale j:

$$s_{j,2k+1} = d_{j-1,k} + \frac{s_{j,2k} + s_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1} - 1. \tag{9}$$

We now need to define the approximation coefficients $s_{j-1,k}$ at scale $j-1$. We want these coefficients to satisfy multiple properties. First, we want them to be close to the signal f at the sampling points $x_{j-1,k}$, to match the expectation we made in equation (7). Second, to achieve minimal memory intensity, we want to use as few other coefficients as possible. This is related to the concept of *compact support* in wavelet theory. One way to achieve this is to only use the approximation coefficients at scale j and the detail coefficients at scale $j-1$. For example, we can introduce coefficients $\alpha_{j,k}$, and define:

$$s_{j-1,k} = s_{j,2k} + \alpha_{j-1,k-1} d_{j-1,k-1} + \alpha_{j-1,k} d_{j-1,k}, \quad 0 < k < 2^{j-1}. \tag{10}$$

Since we have a degree of freedom for the coefficients $\alpha_{j,k}$, we add a mass conservation requirement:

$$\frac{f(0) + f(1)}{2} + \sum_{k=1}^{2^{j-1}-1} s_{j-1,k} = \frac{f(0) + f(1)}{4} + \frac{1}{2} \sum_{k=1}^{2^j - 1} s_{j,k}. \tag{11}$$

This equation is essentially a quadrature formula that equates the mass of the samples at scale j to the mass of the samples at scale $j-1$ using the trapezoidal quadrature formula. Notably, the weights of the first and last samples are halved because half of their weight is located outside the interval $[0, 1]$ if we use symmetric wavelets. However, we can notice that the first and last samples will not have any impact on the overall mass, as they are fixed by equation (6). In the framework of numerical simulations, global mass conservation is generally necessary to ensure proper convergence [20]. As we state it (equation (11)), the mass conservation property ensures that the mass of the original sampled signal $s_{J,k}$ is concentrated only in the samples $s_{j,k}$ at all scales j. This implies that the detail coefficients $d_{j,k}$ have no mass and, hence, that any modification of the detail coefficients $d_{j,k}$ will not affect the mass of the reconstructed signal.

We can check that there is only one choice of coefficients $\alpha_{j,k}$ that satisfies the mass conservation property (11):

$$\alpha_{j-1,k} = \begin{cases} \frac{1}{4} & \text{if } 0 < k < 2^{j-1} - 1 \\ \frac{1}{2} & \text{if } k = 0 \text{ or } k = 2^{j-1} - 1 \end{cases} \tag{12}$$

It is now possible to compute all the coefficients $s_{j,k}$ and $d_{j,k}$ knowing the approximation coefficients at scale $j+1$ thanks to equations (6), (8), (10), and (12). These equations align with the wavelet lifting scheme, introduced by Sweldens, which is known to be optimal in terms of memory accesses [52, 53].

The presented scheme lets us perform a reversible transform on the samples of a 1-dimensional non-periodic signal. It can be directly extended to higher dimensions by successively applying the same scheme along each axis. The DWT steps are commutative across dimensions and scales, meaning that the order of the performed transforms can be leveraged for pratical purposes.

After applying the DWT, the number of coefficients in the data remains identical. At the finest scale J, we have 2^J+1 (approximation) coefficients, while we have 2^j+1 approximation coefficients and $\sum_{i=j}^{J-1} 2^i$ detail coefficients at scale $j < J$ (hence, a total of 2^J+1 coefficients as well). To achieve effective compression, this transform must be combined with a true (potentially lossy) compression method. The idea is that the DWT produces a near-sparse representation of the signal thanks to the vanishing moment property of the wavelets. This property can be leveraged by the compression scheme to achieve higher compression ratios. The next section describes how we design an efficient compression scheme that relies on the data near-sparsity produced by the DWT.

3.2 Compression Scheme

While the DWT offers a near-sparse representation of a signal, this representation alone does not guarantee data compression. Achieving high compression ratios, especially for CFD data, necessitates a well-tailored methodology that makes the most of this near-sparsity. In this section, we initially provide an overview of a general methodology suitable for compressing such data, as detailed in Section 3.2.1. Subsequently, we introduce our novel modification, optimized for GPU hardware, in Section 3.2.2.

3.2.1 General Compression Methodology with DWT

To leverage the Discrete Wavelet Transform (DWT) for data compression, the process typically involves two main steps: thresholding and entropy coding. When applying the DWT to a signal f that is predominantly linear at the scales of application, the outcome is an expected reduction in data entropy. This reduction is due to the majority of the detail coefficients $d_{j,k}$ being close to zero, which enables classical entropy coders to achieve better compression ratios. Introducing a controlled amount of loss into the compression scheme can further enhance these ratios significantly. Central to this approach is the implementation of a threshold τ, with hard thresholding and soft thresholding being the two commonly employed strategies. Hard thresholding zeroes out all coefficients $d_{j,k}$ with absolute values below τ, while soft thresholding extends this by adjusting coefficients with absolute values above τ, bringing them nearer to zero. This is based on the understanding that small $d_{j,k}$ coefficients relates to near-accurate predictions, meaning their removal should minimally affect the signal reconstruction. Importantly, these operations maintain the overall mass of the reconstructed signal, as indicated by equation (11). To compensate for the increased sensitivity at coarser scales, an effective threshold $\tau_j = \tau 2^{J-j}$ is applied for each scale j, to account for the greater impact of removing a coefficient $d_{j,k}$ at coarser scales. Mathematically, this choice is governed by the norm in which we measure the approximation. On this technical subject, we refer for instance to [10] and included references.

Finding the best threshold τ is not straightforward, because it affects multiple aspects of the scheme, such as the compression ratio and the accuracy of the reconstructed signal. In practice, it is often observed that there is a range of values in which any τ value yields acceptable results. Values outside this range either lead to poor compression ratios or to a significant loss of accuracy. Hence, the choice of τ is let to the expertise of the user, but does not require extensive tuning.

Finally, any lossless compression method can be applied to the remaining coefficients $d_{j,k}$. Since the details are spare, sparse data storage methods can be used to achieve lossless compressions, by discarding the zeros from the data representation. One such format is the COO (Coordinate) sparse matrix format, which stores only the non-zero coefficients along with their coordinates. To implement this methodology on the GPU, several challenges arise, which we address in the next section.

3.2.2 GPU-Optimized Compression Methodology

In this section, we present challenges and solutions for implementing the DWT-based compression scheme on GPUs. We assume that the data are stored in a 3-dimensional array $f_{i,j,k}$, where i, j, and k are the indices along the x, y, and z axes, respectively. Performing the DWT on the y and z axes is straightforward, as the lifting scheme can be performed directly on vectors of lines of the x axis. On the x axis, however, a coalesced access leads to the values being stored on different threads because the x dimension is contiguous in memory. A solution is to store each line (of the x axis) in the shared memory of the block and then perform the DWT within the shared memory. With this method, one read and one write need to be performed per coefficient in the global memory for the x axis, which is optimal. Then, the result of the DWT can directly be processed by the lossless compression method. In this work, we use the `dense_to_CSR` and `CSR_to_dense` algorithms from the `cuSPARSE` library to convert the data to the CSR format, which is a sparse matrix format.

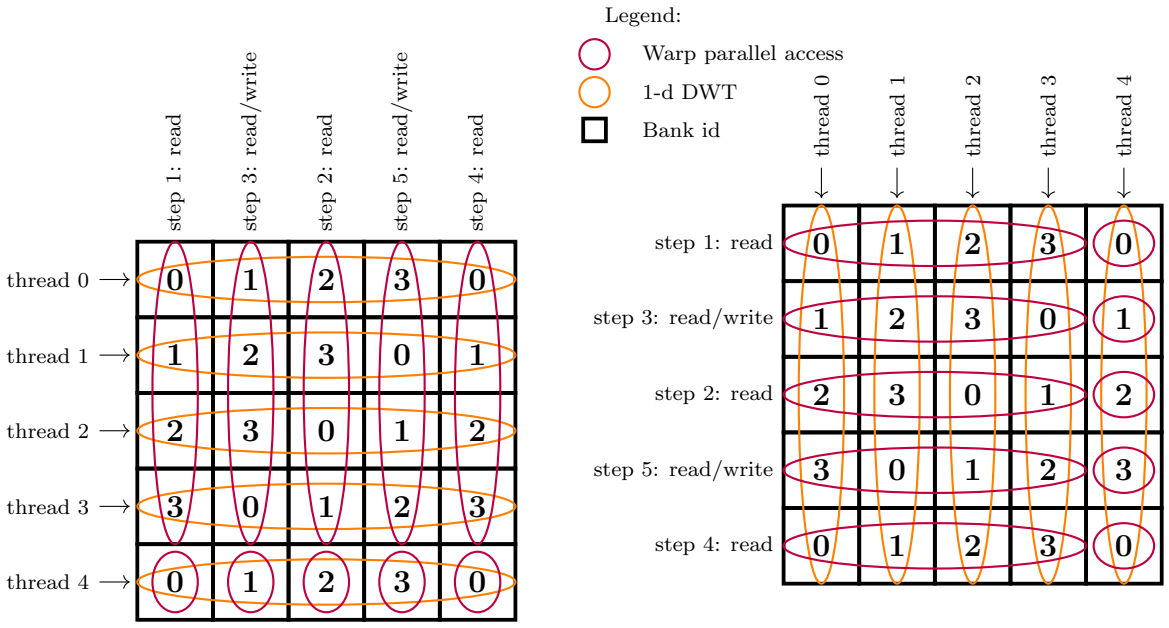
This naive, but reasonably efficient approach is slowed down by several factors and should not be used in practice. While the achieved compression ratio can be impressive (up to 10000x can be achieved at the beginning of the later described D3Q27 scheme), the throughput is limited by the memory intensity of the DWT. It is also limited by the reliance on separate kernels for the different steps of the compression scheme, which leads to an overhead.

In our novel approach, we propose to perform smaller local DWTs on the data and leverage the shared memory to store the data. As the shared memory is significantly faster than the global memory, we expect this approach to yield substantial performance improvements. The shared memory differs from global memory in different aspects. It is block-level memory with limited capacity (usually in the order of tens of kilobytes per block) and is accessible only by cores within the same block. Shared memory is also divided into several regions (banks) that can be accessed simultaneously by threads of a same warp (set of 32 threads). While exact coalesced access patterns are unnecessary, shared memory is prone to bank conflicts when multiple threads access the same bank simultaneously, leading to potential slowdowns due to serialized access. To leverage shared memory effectively, our parallelization strategy is designed to circumvent bank conflicts and optimize memory throughput.

Our algorithm divides the global grid into smaller blocks, each sized to fit within the capacity limitations of shared memory. Each CUDA block transfers data from global to shared memory, executes the DWT in-place within shared memory, and writes the compressed results back to global memory. The chosen block size is $33 \times 17 \times 17$, consuming approximately 37.25 KB of shared memory for single-precision data, which is within the capacity of most modern GPUs. The block sizes are intentionally set to powers of 2 plus 1 to accommodate the DWT scheme utilized. The 1-d DWT is applied consecutively along each axis of the 3-d block within the shared memory, with each thread processing a different line of the block. Threads employ the lifting scheme on their respective lines and synchronize between axes using the `__syncthreads()` function, hence requiring at least two thread synchronizations overall.

Figure 2 provides a schematic of the shared memory layout in a 2-d slice of the 3-d block fetched from global memory, demonstrating the execution of memory accesses for the DWT along the x and y axes. The "step x" labels indicate the sequence of memory accesses in the lifting scheme implementation. Assuming a bank number and a warp size of 4 for illustration, the principle remains applicable for any power of 2. Given the block sizes are powers of 2 plus 1, the bank numbers are the same across each axis, with bank numbers incrementing by 1 (modulo the total bank number) when transitioning to adjacent cells in the same row or column. This layout ensures an even distribution of memory accesses across banks, crucial for minimizing bank conflicts. Moreover, the indivual warp accesses (depicted by the red oval shapes) always access to different banks, which guarantees no bank conflicts. The same principles apply to the z axis, which is not shown for brevity.

Finally, the result of the DWT can be compressed using any lossless compression method. We choose to use a COO (Coordinate) format to achieve lossless compression. This choice offers both high compression ratios thanks to the sparsity of the data and relatively fast compression/decompression times. However, the GPU code for performing the *dense-to-COO*



(a) Wavelets on the x axis.

(b) Wavelets on the y axis.

Figure 2: Schematic representation of the shared memory layout in a 2-d slice of the 3-d block loaded from the global memory. For the sake of visibility, a bank number and a warp size of 4 are assumed. Part 2a shows how the wavelets on the x axis are performed, while part 2b shows how the wavelets on the y axis are performed. The parts where the 1-d DWT is performed are hightlighted with orange oval shapes. The individual warp accesses are represented by the red oval shapes. Each cell in the shared memory is represented by a square and its corresponding bank is written in the cell.

(compression) and *COO-to-dense* (decompression) operations is not trivial, as it involves irregular memory accesses and inter-thread communication. The *dense-to-COO* is close in spirit to a parallel *reduction* or *scan* [4], while the *COO-to-dense* is close to a parallel scatter operation. Our implementation of *dense-to-COO* is close to the idea for GPU parallel reduction provided by Harris [19], with the notable change that the warp-level reduction primitives are now directly available in the CUDA programming model. The warps start by scanning the non-near-zero coefficients across the block and counting them by performing a warp-level reduction. Then, a block-level reduction is performed to compute the offset of each warp for the final write to the COO format. The *COO-to-dense* operation is more direct, as it is a simple scatter operation. The thresholding is performed during the *dense-to-COO* by integrating only the coefficients above the threshold into the COO format.

This new compression scheme, which works with local wavelets, is expected to be faster than the first version with global wavelets, as it minimizes the number of global memory accesses. The compression kernel performs a single read from global memory on the decompressed data, followed by a single write on the compressed data (and conversely for the decompression kernel). Data reuse is maximized by performing the required memory accesses on the shared memory, which moves the bottleneck from DRAM (global) accesses to faster SRAM (shared) accesses. The downside of this approach is that the compression is performed on local blocks, which hurts the compression ratio. However, we will see in Section 4 that the compression ratio is still acceptable in practice.

3.3 Methodology for CFD Data Compression

Our approach strategically partitions the computational grid into smaller, manageable subgrids, which lets us process the data in a more flexible manner. This partitioning is an important aspect of our compression scheme, as it is required to achieve actual memory savings. As depicted in Figure 3, the entire grid is divided into subgrids, which are further segmented into blocks. These blocks are only used in the DWT step, where they are loaded in the shared memory of a CUDA block to perform the DWT locally.

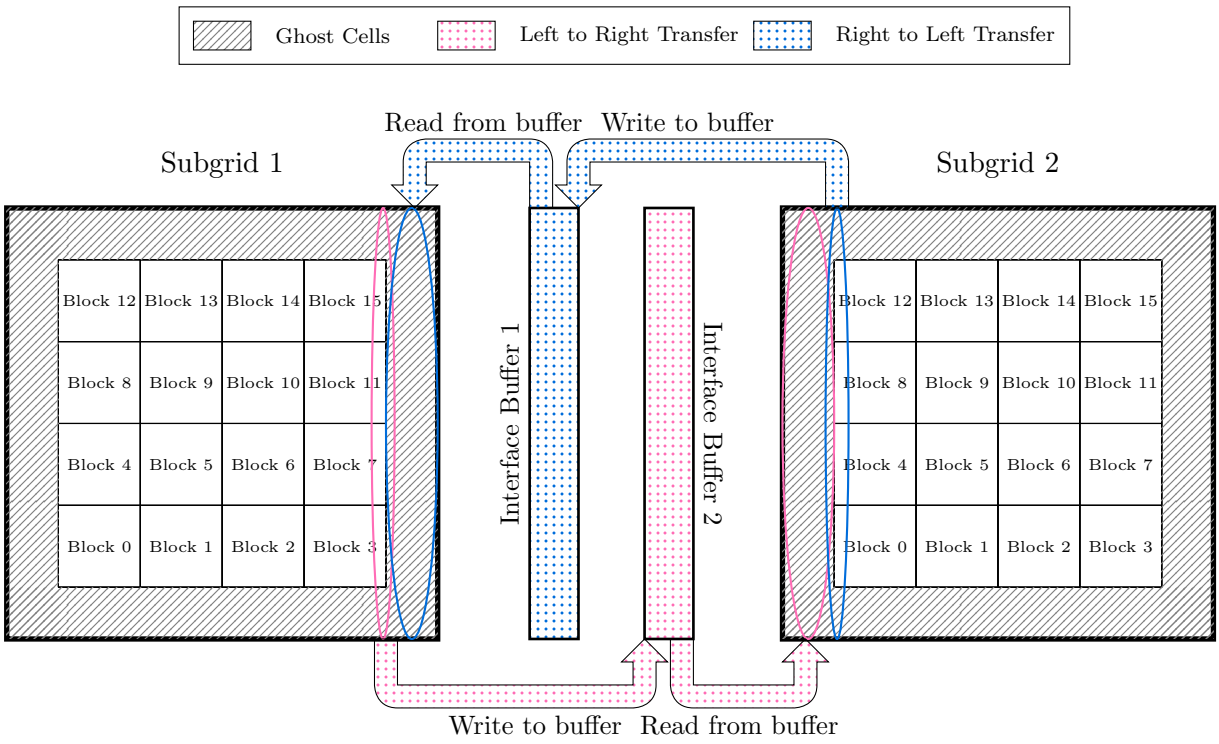


Figure 3: Illustration of the hierarchical grid subdivision in 2D. The grid is subdivided into subgrids, which are further segmented into blocks. Each block is processed by a single CUDA block for the DWT and is sized to fit within the shared memory of the GPU blocks. Subgrids represent contiguous memory segments in the global memory (when decompressed), while blocks include offsets between rows. Synchronization between subgrids is facilitated through interface buffers along each dimension/direction. Ghost cells are updated by reading from these buffers and values are written back to the buffers after each LBM iteration. Blocks do not need synchronization, as the LBM computations are performed directly on the global decompressed subgrid. This hierarchical model is extendable to multiple dimensions.

When decompressed, the subgrids use a classical row-major storage format, with each row stored contiguously in memory. This implies that the blocks are not contiguous in the global memory, as they are separated by the offsets between rows. Each partitioning serves a different purpose: subgrids allow for partial decompression of of the grid, while blocks facilitate the compression/decompression (DWT and COO) operations.

LBM computations are performed directly on the decompressed subgrids, which are stored in the global memory. These necessitate the values of the neighboring subgrids to be available. To achieve this, we use a classical ghost cell approach, where the ghost cells duplicate the values of the neighboring subgrids. To account for the fact that the neighboring subgrids are not necessarily directly available (due to being compressed), we use interface buffers. These interface buffers let us have an uncompressed version of the relevant edge values of all the grid. This lets us divide the subgrid synchronization into two phases: reading from the interface buffers to update the ghost cells and writing to the interface buffers the results of the LBM computations.

To achieve effective memory gains, we only reserve a fixed amount of memory for two subgrids and all the interface buffers. The idea is to process each subgrid consecutively, hence only allowing to have at most two uncompressed subgrids in the global memory at a time. The same idea would work with a single subgrid if we assumed in-place computation of the LBM step, but we do not make this assumption for the sake of generality. The interface buffers are used in a duplicated, alternating fashion to ensure coherent data accesses between different time steps. The rest of the memory is used for the compressed version of the subgrids and is stored in a circular buffer. This buffer can be viewed as an infinite succession of compressed subgrids $s_{0,\mathrm{it}}, s_{1,\mathrm{it}}, \ldots, s_{N-1,\mathrm{it}}, s_{0,\mathrm{it}+1}, s_{1,\mathrm{it}+1}, \ldots$, where $s_{i,\mathrm{it}}$ is the compressed subgrid i at iteration it and N is the number of subgrids.

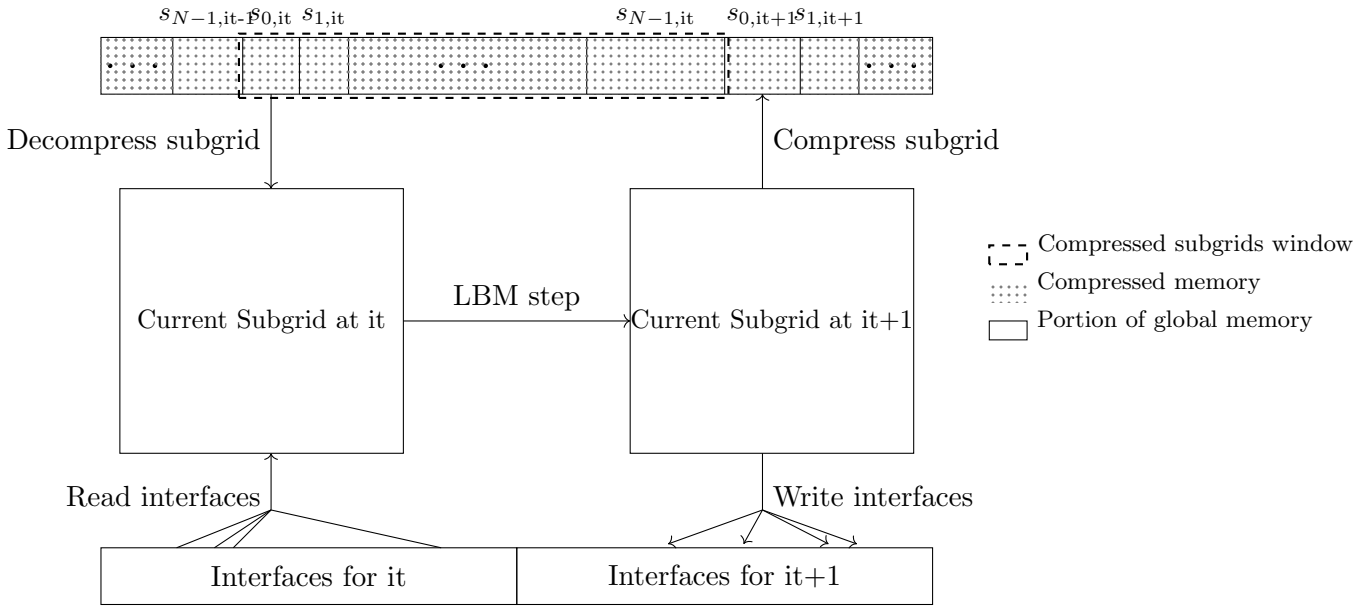


Figure 4: The figure illustrates the workflow for executing a Lattice-Boltzmann step on subgrid 0 at iteration *it*. All the data shown in the figure are stored at all time in the GPU global memory (DRAM). Initially, the subgrid is retrieved from the circular buffer in its compressed form and decompressed into a subgrid buffer. Prior to the LBM step, ghost cells are updated (indicated by "Read interfaces" arrows). Post-LBM step, the processed data are stored in a new buffer. Interface data for subsequent iterations are stored ("Write interfaces" arrows), and the resultant subgrid is re-compressed and written to the circular buffer. The GPU memory is strategically partitioned into three segments: the circular buffer for compressed subgrids, subgrid buffers for decompressed data, and interface buffers. These subgrid and interface segments are of fixed sizes, ensuring efficient reuse in each iteration. The design incorporates two sets of interface buffers that alternate between iterations to represent current and subsequent iteration data. It is assumed that $s_{0,\text{it}}, s_{1,\text{it}}, \ldots, s_{N-1,\text{it}}$ does not overlap with $s_{0,\text{it+1}}, s_{1,\text{it+1}}, \ldots, s_{N-1,\text{it+1}}$ (i.e., the compressed size fits within the circular buffer).

Figure 4 illustrates how these segments are used in the execution of a LBM step on subgrid 0 at iteration *it*. Let us first notice that the circular buffer contains a window in which the compressed subgrids that are still needed for the simulation are stored. The goal of this whole process is to advance this window by one subgrid. The process begins with reading the compressed subgrid from the circular buffer and decompressing it into a buffer. Prior to the execution of the LBM step, the ghost cells are updated using data from the corresponding interface buffers. Post LBM step, the results are stored in the other interface buffer and the processed subgrid is re-compressed and written to the current cursor of the circular buffer. This process is then repeated for the next subgrid (or the next time step if there are no more subgrids), with the compressed subgrids window slided to the right. This cyclical process of reading, updating, processing, and writing back to the circular buffer ensures that the required memory remains below the GPU capacity (assuming a given compression ratio). Overall, this process flow allows to reach effective memory savings, as only partial decompression of the data is performed at a time.

The integration of our compression methodology with an LBM simulation lets us bypass the need for CPU-GPU data transfers, as long as the compressed data and the decompressed data (the subgrid and interface buffers) fit in the GPU memory. By addressing these memory constraints, our approach enables the execution of larger-scale simulations for a given hardware. In the next section, we present results to demonstrate the effectiveness of this approach.

4 Results

4.1 Experimental Setup

To evaluate how our compression scheme influences the execution of an LBM simulation, we performed a set of experiments. Our selection for this assessment is a D3Q27 LBM flow simulation featuring a sphere as an obstacle. This scenario is a well-established challenge in fluid simulations, providing insights into various flow regimes. Our D3Q27 LBM is a variant of the initial scheme of d'Humières [13].

Additionally, it is known to present local complex fluctuation mixed with large quiet regions, especially in the presence of unsteady flows. These features are well adapted to wavelet compression. Additionally, our investigation confirmed that this approach exhibits a memory-bound characteristic on GPUs, achieving a memory bandwidth of approximately 80% of the theoretical peak across all tested GPUs (excluding the bounce-back condition). This observation underscores rapid execution of the scheme on GPUs, a feature that poses challenges for employing compression, as it accentuates the overhead associated with compression techniques.

The conservative variables of this LBM scheme are the density and the density-weighted velocity: $W = (\rho, \rho u_x, \rho u_y, \rho u_z)$. We have thus 4 physical data by grid point. The physical data are represented at each grid point by a larger set of so called kinetic data f_i for $0 \leq i < 27$, hence the "D3Q27" terminology. The mapping between f_i and W_j is performed as described below. We refer to the (excellent) book of Krüger *et al.* [26] for an introduction to the LBM.

The equilibrium distribution function $f_{eq,i}$ provides the lattice velocity i at equilibrium, and is computed using the following equation:

$$f_{eq,i}(W) = C_i \rho \left(1 + \frac{3}{c^2}\vec{e}_i \cdot \vec{u} + \frac{9}{2c^4}(\vec{e}_i \cdot \vec{u})^2 - \frac{3}{2c^2}\vec{u} \cdot \vec{u}\right), \tag{13}$$

where C_i is the weight of the velocity $\vec{e}_i$ and c is the speed of sound. The implementation is based on the commonly used 27-velocity set: $(0,0,0)$, $(\pm 1,0,0)$, $(0,\pm 1,0)$, $(0,0,\pm 1)$, $(\pm 1,\pm 1,0)$, $(\pm 1,0,\pm 1)$, $(0,\pm 1,\pm 1)$, and $(\pm 1,\pm 1,\pm 1)$, with corresponding weights: $\frac{8}{27}$ (for the center), $\frac{2}{27}$ (for the 6 "faces"), $\frac{1}{54}$ (for the 12 "edges"), and $\frac{1}{216}$ (for the 8 "corners") [57]. This allows us to compute W from the distribution function f and vice versa using the following equations:

$$\begin{aligned} \rho &= \sum_{i=0}^{26} f_i &&= \sum_{i=0}^{26} f_{eq,i}, \\ \rho\vec{u} &= \sum_{i=0}^{26} c\vec{e}_i f_i &&= \sum_{i=0}^{26} c\vec{e}_i f_{eq,i}. \end{aligned} \tag{14}$$

Each step of the LBM algorithm starts by shifting the 27 kinetic data f_i in the directions of the corresponding 27 lattice velocities $\vec{e}_i$. This step induces memory transfer between the grid points. In the second step, the kinetic data are updated according to:

$$f = \omega f_{eq} + (1 - \omega) f, \tag{15}$$

where ω is the relaxation parameter. This step is done locally at each grid point and is completely parallelizable on the GPU.

Thus, a time step consists of the following operations:

1. read the kinetic data f_i from the neighbors (shift phase). If the neighbor is in an obstacle, we consider reading $f_{i'}$ instead, where $e_{i'}$ is the opposite velocity to e_i in the lattice (bounce-back condition). This boundary correction can break the coalescent memory access, as we shall see later;
2. compute the conservative data W from equation (14);
3. compute the equilibrium distribution function f_{eq} from W with equation (13);
4. write the new distribution function f according to (15)

We divide the domain into $4 \times 16 \times 4$ subgrids, each of which being a grid of cells of size $\Delta x \times \Delta y \times \Delta z$, with $\Delta x = \Delta y = \Delta z$. The initial condition is $\rho = 1$ everywhere, $\vec{u} = (0.0001, 0.03, -0.0001)$ outside the obstacle, and $\vec{u} = (0, 0, 0)$ inside the obstacle. The time step Δt is deduced from Δx and the speed of sound c (set to a dimensionless value of 1): $\Delta t = \text{CFL}\frac{\Delta x}{c} = \Delta x$ in our case. The CFL number is set to one as is always the case in the LBM. We set the relaxation parameter ω depending on Δt so that the corresponding Reynolds number is 300. The ω parameter is computed from the following equations:

$$\nu = c^2 \left(\frac{1}{\omega} - \frac{1}{2} \right) \Delta t, \tag{16}$$

$$\Leftrightarrow \omega = \frac{2}{1 + 2\frac{\nu}{c^2 \Delta t}}, \tag{17}$$

while the viscosity ν is set such that the Reynolds number is 300:

$$\text{Re} = \frac{cL}{\nu} = 300, \tag{18}$$

where L is the characteristic length of the obstacle; in our case, the diameter of the sphere. We use periodic boundary conditions in all directions except when we show the vortices passing the obstacle. In this case, we use a fixed boundary condition for the low y values, so that the incoming flow is constant.

We run tests with different NVIDIA GPUs: P100 (16GB), V100 (16GB), and A100 (40GB). The code is written in CUDA and runs mostly on the GPU. We use custom CUDA events to measure the time spent and verify that our measures are coherent with the output of `nvprof`. Our implementation allows for different scenarios, such as with or without compression, with or without subgrids, different threshold values, and different obstacles. The results can be saved thanks to a custom compressed file format and visualized thanks to a custom python script based on the mayavi library [42].

4.2 Setting the Threshold

The objective of this experiment is to assess the impact of the threshold value on simulation execution. Flow simulations were conducted at various threshold levels, with results recorded at $t_{\max} = 1.0s$. The grid size for these simulations was $231 \times 952 \times 238$, with $\Delta x \approx 0.01732$.

Figure 5 displays the results of the experiment, underscoring two main observations. The graph shows the average compression ratio, depicted by blue crosses, which compares the size of compressed data to uncompressed data at a specific timestep. It also presents the Normalized Mean Squared Error (NMSE) between the reference density and the density after lossy compression, defined as:

$$\text{NMSE} = \frac{\sqrt{\sum_{i,j,k} \Delta x \Delta y \Delta z \left(f_{i,j,k} - \hat{f}_{i,j,k} \right)^2}}{\sqrt{\sum_{i,j,k} \Delta x \Delta y \Delta z \left(f_{i,j,k} \right)^2}}, \tag{19}$$

where $f_{i,j,k}$ is the reference density and $\hat{f}_{i,j,k}$ is the density after lossy compression.

This visual analysis permits a detailed examination of the effects of threshold variation on the simulation. Distinct behavioral regimes are identifiable:

- In the range of $[10^{-8}, 2 \cdot 10^{-8}]$, increasing the threshold significantly improves the compression ratio without altering the error, suggesting coefficients removed in this range likely correspond to noise.
- Between $]2 \cdot 10^{-8}, 10^{-6}]$, there is a nearly linear increase in error, with a less pronounced rise in the compression ratio. This indicates the beginning of an impact on the simulation by the removal of the coefficients, yet without major disruption.
- Beyond $]10^{-6}, 2 \cdot 10^{-4}]$, a sharp increase in compression ratio is observed alongside error stabilization, implying that artifacts at this stage severely compromise simulation integrity, rendering the simulation impractical.

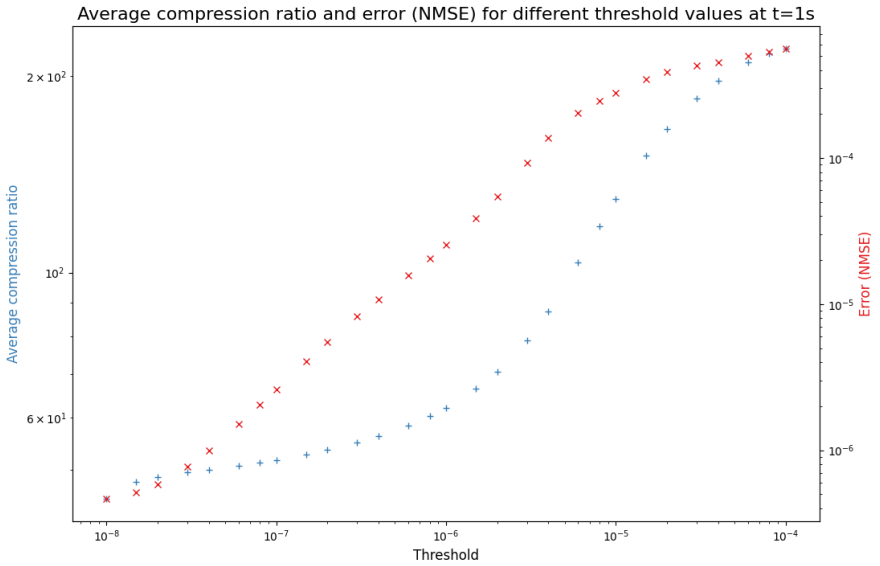


Figure 5: Impact of the threshold on compression ratio and error in a D3Q27 LBM simulation. The figure shows average compression ratios and errors across the domain for different threshold settings. Blue crosses indicate the compression ratio, and red crosses denote error, measured as the Normalized Mean Squared Error (NMSE) between reference density and density after lossy compression.

The first two regimes are considered potentially beneficial for meaningful simulations. The initial regime offers an optimal scenario, enhancing compression without affecting accuracy and eliminating superfluous noise. The subsequent regime, though riskier, allows for increased compression at the risk of introducing disruptive artifacts, necessitating thorough result analysis. The final regime, marked by excessive error, is deemed unsuitable for productive simulation efforts.

Notably, threshold determination is influenced by specific problem parameters and poses a challenge due to interactions between numerical and compression schemes. However, chosing a threshold value in the first regime is likely to yield acceptable results, as it offers a balanced compromise between compression efficiency and error minimization without evident artifacts. Hence, the threshold value is set to $2 \cdot 10^{-8}$ for the subsequent experiments, as it falls within the optimal range.

For simulations using different grid sizes than $231 \times 952 \times 238$, the threshold value adjusts to ensure analyzed frequencies align with identical physical scales:

$$\tau = \tau_0 \frac{\Delta x}{\Delta x_0}, \tag{20}$$

where Δx_0 is the baseline spatial step (≈ 0.01732) and τ_0 is the predetermined threshold value ($2 \cdot 10^{-8}$).

4.3 Validation of the Scheme

We perform various tests to validate the scheme. We first verify that the scheme preserves the mass up to machine precision. This property holds true as long as the domain is periodic and no fixed boundary condition is used. It works, among other:

- with the direct implementation of the LBM scheme on the GPU (without compression);
- with the implementation with subgrids but no compression;
- with the implementation with subgrids and compression.

We, hence, have strong evidence that both the subgrid synchronization mechanism and the compression scheme are correctly implemented.

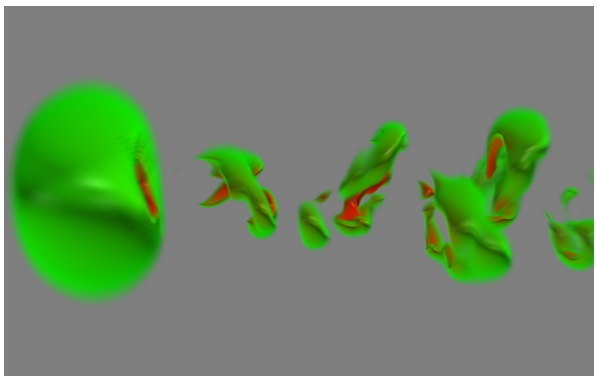


Figure 6: D3Q27 LBM simulation at $t_{\max} = 600s$ (24777 iterations) with a $165 \times 680 \times 170$ grid ($\approx$ 1.92GB) and no compression.

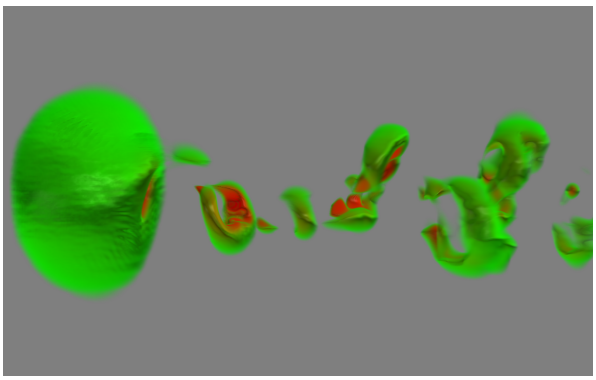


Figure 7: D3Q27 LBM simulation at $t_{\max} = 600s$ (24777 iterations) with a $165 \times 680 \times 170$ grid ($\approx$ 1.92GB) and lossy compression.

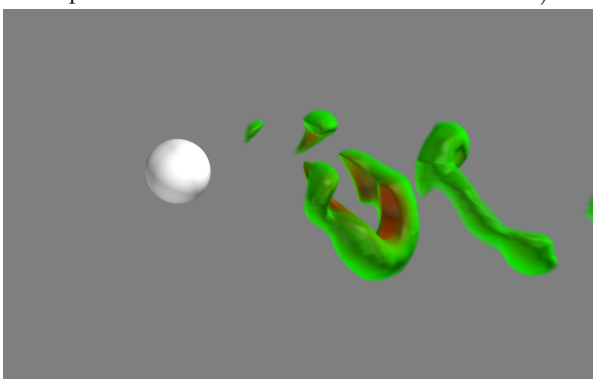


Figure 8: D3Q27 LBM simulation at $t_{\max} = 1000s$ (148685 iterations) with a $594 \times 2448 \times 612$ grid ($\approx$ 89.51GB).

To further validate the scheme, we show the visual results of the simulation for various configurations. We set ω so that the corresponding Reynolds number is 300. Figures 6 and 7 show the results of the simulation at $t_{\max} = 600s$ with a $165 \times 680 \times 170$ grid. The first figure is the result of the simulation without compression, while the second figure is the result of the simulation with compression. In both cases, we observe the highly periodic flow that we would expect for a Reynolds number of 300 [22, 54]. Figure 8 shows the result of the simulation at $t_{\max} = 1000s$ with a $594 \times 2448 \times 612$ grid. This simulation is costly in terms of computations and took approximately 3 days to run on an A100. Only the subspace $x = [-1.25, 0.75]$, $y = [1.9, 5]$, $z = [-0.55, 1.45]$ is saved, corresponding to approximately 9GB of floating point data in our case, which is close to the maximum size that can be visualized on a regular laptop. The saved space captures two vortex rings, which consist of a similar flow pattern to the smaller simulation. These visual results indicate that the compression scheme does not introduce significant errors and demonstrate the potential to simulate higher-precision simulations on GPUs. Let us note that this D3Q27 scheme is a worst case scenario for the compression scheme, as the turbulences are known to originate from slight variations in the flow, which can be disrupted by the compression scheme. To obtain these results, the threshold must be particularly low, which severely impacts the compression ratio, as we will see in the next section.

4.4 Performance Evaluation

This section presents the results of our performance evaluation for the D3Q27 simulation, considering five configurations: direct implementation with two fully decompressed grids (with and without bounceback condition), implementation with subgrids and no compression, and implementation with subgrids and compression (block-level wavelets or global wavelets). The block-level compression is the novel compression scheme where the wavelets

are performed locally in the shared memory, while the global compression is the previous compression scheme where the wavelets are performed on the whole subgrids. In the following experiments, the block-level performs 3 DWTs on the x axis and 2 DWTs on the y and z axes (to achieve $j = 2$ on all axes for a block size of $33 \times 17 \times 17$), while the global compression performs 2 DWTs on all axes.

For each configuration, we run the simulation with different grid sizes with $t_{\max} = 1.0s$, except for the compression with global wavelets, where $t_{\max}$ is lowered depending on the grid size because of how slow the execution is. We verified that the number of performed iterations does not significantly impact processing speeds.

		Proportion of time spent in the kernels (in percentage)				Compression ratio (first time step)
GPU	Configuration	Numerical scheme	Compression	Decompression	Synchronization	Ratio
A100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
A100	Subgrids, no compression	91.12%	0.00%	0.00%	8.88%	1.00
A100	Subgrids, block compression	51.04%	28.59%	17.78%	2.59%	206.57
A100	Subgrids, global compression	3.18%	81.20%	15.49%	0.13%	1206.12
V100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
V100	Subgrids, no compression	87.96%	0.00%	0.00%	12.04%	1.00
V100	Subgrids, block compression	46.57%	25.92%	22.65%	4.86%	200.93
V100	Subgrids, global compression	4.83%	73.12%	21.68%	0.36%	539.41
P100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
P100	Subgrids, no compression	87.70%	0.00%	0.00%	12.30%	1.00
P100	Subgrids, block compression	36.25%	37.75%	22.27%	3.73%	200.93
P100	Subgrids, global compression	6.18%	68.60%	24.73%	0.49%	539.41

Table 1: Average percentage of time spent in the different kernels on the different configurations.

Table 1 shows the average percentage of time spent in the different kernels for the different configurations. It provides insights into the performance of the different configurations. We can see that the global compression, where the wavelets are performed on the whole subgrids, is significantly slower than the block-level compression, where the wavelets are performed at the block level. With global compression, between 3% and 7% of the time is spent on average in the LBM computations. For the block-level wavelets, both the A100 and the V100 spend approximately 50% of the time in the LBM computations, while the P100 spends approximately 35% of the time. We hence, see that the global wavelets are drastically slower than the block-level wavelets. The table also shows the compression ratio achieved on the first time step for the different compression kernels. This metric highlights the fact that the global compression yields better compression ratios than the block-level compression, with one less DWT level on the x axis. Both algorithms, hence, provide a different trade-off between compression ratio and execution time. If the required compression ratio becomes the bottleneck, the global compression kernels can be used, and the amount of performed DWTs can be adjusted to reach the desired compression ratio.

To normalize the performance between different configurations, we propose to compare the processing speed of the different configurations in Figure 9. The x-axis represents the total grid size if decompressed, while the y-axis indicates processing speed if decompressed. The processing speed (S) is calculated using Equation 21, considering grid size, number of iterations, and total time, with a factor of 2 for read-write cycles per iteration.

$$\mathrm{S} = 2 \times \frac{\text{grid\_size} \times \text{num\_iterations}}{\text{total\_time}} \text{ GB/s} \tag{21}$$

This figure helps to evaluate the impact of the method on simulation performance. Firstly, the executions with no bounceback condition (dashes) achieve high processing speeds, typically about 80% of the theoretical peak of the GPU. On the other hand, the same executions with the bounceback condition (dots) are significantly slower, between 2 and 5 times slower, depending on the grid size and the GPU. This is explained by the fact that the bounceback condition breaks the coalescent memory access, which is a well-known issue when implementing the LBM on the GPU [39]. The executions with subgrids, but no compression (triangles), use the same workflow we use for compression (see Section 3.3). We can see that this method is able to run larger simulations. This is due to the used workflow, which does not require to store two fully decompressed grids at the same time. We can also see that the processing speed is not systematically lower than the version with no subgrids. This can be explained by the better granularity of the version with the subgrids, where the subgrids that have no bounceback condition (due to not including the obstacle) operate at the near-perfect

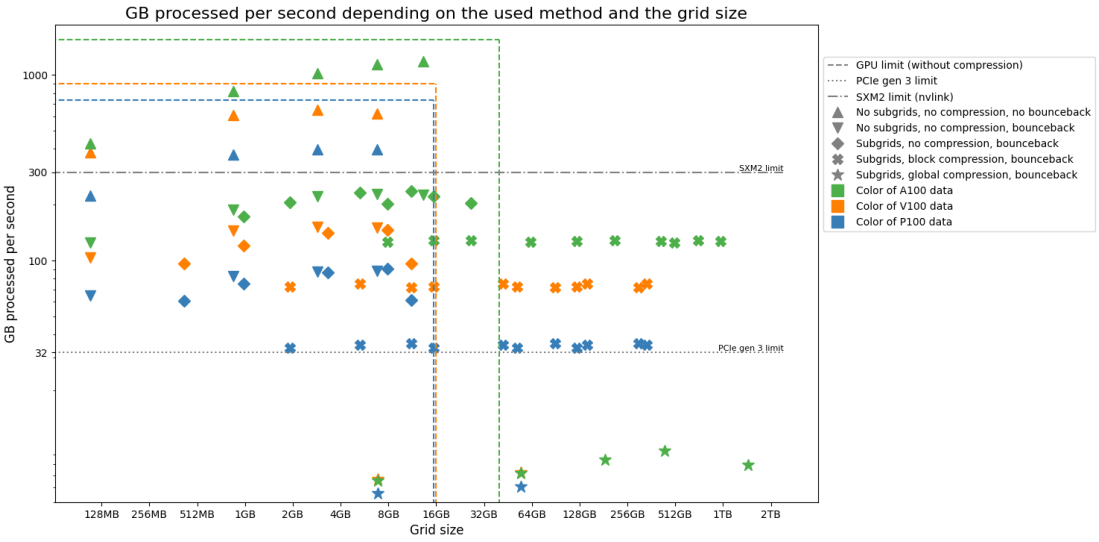


Figure 9: Performance evaluation of the D3Q27 LBM simulation with different configurations. The x-axis shows the total grid size (in GB) if decompressed, while the y-axis displays the processing speed (in GB/s). Hardware (P100, V100, or A100) is represented by color, while the marker denotes the method used. The markers represent the result for a single execution. The dashed colored lines represent the theoretical bounds of each tested GPU based on their specifications, assuming no compression.

speeds that we oberve for the version with no bounceback condition (dashes). This gain can overcome the overhead of the subgrid synchronization.

Finally, the executions with subgrids and compression (crosses) show the impact of the compression on performance. We can expect the simulations that integrate compression to be approximately 2 to 3 times slower than the best case scenario without compression. This slowdown allows the simulation of grids of size up to 13 times the capacity of the 40GB A100 and 8 times the capacity of the 16GB P100/V100 (up to $t_{\max} = 1.0s$). It is important to note that unless in-place computation is used, the grid size would normally be at most half the capacity of the GPU, as the grid needs to be stored twice (once for the input and once for the output). The implication of this observation is that for a given hardware, the effective grid size that can be simulated is significantly increased by the use of compression.

The figure also shows the maximum bandwidth of PCIe gen 3 and SXM2 (NVLink), which are associated with the V100 GPU. We can see that all the processing speeds are greater than the maximum PCIe (gen 3) bandwidth of 32GB/s. The maximum PCIe gen 4 bandwidth (64GB/s) is also surpassed by the A100, which is the only GPU that supports PCIe gen 4. This observation highlights the potential of the compression scheme to reduce the requirements for PCIe data transfers, which is generally a bottleneck in multi-GPU CFD simulations. No processing speed surpasses the maximum corresponding NVLink bandwidth (160GB/s for P100, 300GB/s for V100, and 600GB/s for A100), which are particularly fast.

Figure 10 illustrates how the compression ratio changes over time across different simulation setups. The compression ratio, calculated as the ratio between compressed and uncompressed data sizes, exhibits notable fluctuations during the simulation. If the threshold were set constant across simulations, we would expect setups with larger grid sizes to have higher compression ratios, as discontinuities would form a smaller proportion of the grid as the grid size increases. However, since we normalize the threshold with equation (20), this relationship is not as straightforward.

At the beginning of the simulation, when the grid values remain largely constant, all setups show a higher compression ratio. This ratio then rapidly decreases to its minimum before fluctuating over time without displaying significant abrupt changes, continuing to vary without settling into a stable state. The fluctuation over time is influenced by the changing shapes of vortices throughout the simulation. The rapid decline at the beginning is due to the emergence of shock waves from the obstacle at the start of the simulation. These waves

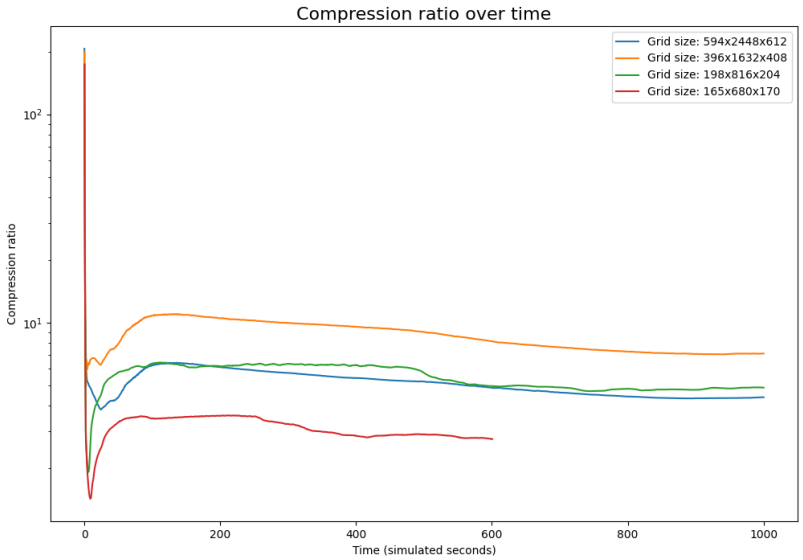


Figure 10: Compression ratio over time for the 3 tested grid sizes. The x-axis represents the time in (simulated) seconds, while the y-axis displays the compression ratio.

propagate through the domain, causing discontinuities that are captured by the wavelet transform, leading to a drop in compression ratio. It is important to understand that these shock waves are numerical artifacts rather than physical phenomena. They are caused by the abrupt change at the initial condition, which is not representative of the physical reality. These artifacts are avoidable through methods such as a gradual initialization process. As these shock waves vanish, the compression ratio stabilizes at a higher level, fluctuating over time but remaining above the minimum.

In our implementation, the lowest compression ratio acts as a bottleneck that may render a simulation unexecutable. This is particularly problematic as simulations often start with shock waves, resulting in temporary drops in compression ratio. However, a more sophisticated implementation could identify such scenarios and employ a slower yet more compressive scheme during the necessary time steps. For example, the global compression scheme can reach extremely high compression ratios, as we have shown in a previous work [15]. Alternatively, compressed subgrids could be stored on a storage device and retrieved when needed.

In conclusion, the results show that the compression scheme is able to run a simulation in a memory constrained environment, introducing an acceptable overhead in comparison to the best possible scenario using the PCIe bus. This compression overhead can even be adjusted downwards by performing multiple LBM steps per compression/decompression cycle, but this would introduce challenges regarding the synchronization of the LBM blocks. The observed compression ratios are high, typically between 5 and 10 for large grids, which validate the usefulness of the method. The trade-off between the compression ratio and the execution time is satisfactory, as the execution throughput is higher than the maximum PCIe bandwidths. Thus, the proposed compression scheme is a viable option in GPU memory-constrained configurations, in particular when the PCIe transfers are a bottleneck.

5 Conclusions

In this paper, we have presented a novel approach to compressing large-scale CFD simulations on GPUs. Wavelet-based compression methods have been widely used in the past, but their application to GPU CFD simulations has been limited due to the memory intensity of the DWT. We have proposed a new method that leverages the SRAM of the GPU to perform local DWT and COO compression. This approach has been tested on a D3Q27

LBM simulation. The results show that it is possible to run simulations that would not be executable with a single GPU, due to the memory requirements. The tested D3Q27 simulation has an overall low execution time on GPUs, which makes it a less favorable candidate for compression. More computationally intensive simulations would presumably yield even better results, as the overhead of the compression would be less significant in comparison to the total execution time. Traditionally, compression is rarely integrated into CFD simulations due to its presumably bad compression ratio/execution time trade-off. However, our results show that it is possible to reach satistactory trade-off, which unlocks significant potential for the execution of larger-scale simulations. Our method can be be improved in different ways. First, we have implemented a pipeline where the LBM step is performed once per compression/decompression cycle. It is, however, possible to perform the LBM step multiple times per compression/decompression cycle. This would lower the overhead of the compression, as the compression would be performed less frequently. This technique is known as temporal blocking and is a common technique in stencil computations. It is normally used to improve the cache usage. Second, the used lossless compression algorithm has been chosen for its simplicity and speed, but it is likely that other algorithms could provide better compression ratios and/or execution times. In particular, for the tested D3Q27 scheme, where the sparsity of the data is not as high as expected due to the low threshold required for accurate results. Further works are being conducted to use better compression methods for near-sparse data. Third, the D3Q27 scheme is performed on the global memory, which is an unnecessary bottleneck. It is likely that the performance of the scheme could be improved by performing the LBM computations on the shared memory, but this introduces challenges regarding the synchronization of the LBM blocks. Finally, integrating our methodology into real-world multi-GPU frameworks is a natural next step. Multi-GPU LBM simulations are often bottlenecked by data accesses and transfers, both of which could be improved by our method.

Acknowledgments

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr). This work of the Interdisciplinary Thematic Institute IRMIA++, as part of the ITI 2021-2028 program of the University of Strasbourg, CNRS and Inserm, was supported by IdEx Unistra (ANR-10-IDEX-0002), and by SFRI-STRAT'US project (ANR-20-SFRI-0012) under the framework of the French Investments for the Future Program. This work was also supported by a Labex IRMIA (11-LABX-0055) grant.

References

[1] Thomas Bellotti, Loïc Gouarin, Benjamin Graille, and Marc Massot. Multidimensional fully adaptive lattice boltzmann methods with error control based on multiresolution analysis. *Journal of Computational Physics*, 471:111670, 2022.

[2] Thomas Bellotti, Loïc Gouarin, Benjamin Graille, and Marc 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.

[3] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[4] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[5] S Boopathiraja and P Kalavathi. A near lossless multispectral image compression using 3d-dwt with application to landsat images. *Int J Comput Sci Eng*, 6(4):332–336, 2018.

[6] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[7] Kolja Brix, Silvia Sorana Melian, Siegfried Müller, and Gero Schieffer. Parallelisation of multiscale-based grid adaptation using space-filling curves. In *ESAIM: proceedings*, volume 29, pages 108–129. EDP Sciences, 2009.

[8] Kolja Brix, Sorana Melian, Siegfried Müller, and Mathieu Bachmann. Adaptive multiresolution methods: Practical issues on data structures, implementation and parallelization. In *ESAIM: Proceedings*, volume 34, pages 151–183. EDP Sciences, 2011.

[9] 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. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[10] Albert Cohen, Wolfgang Dahmen, and Ronald DeVore. Adaptive wavelet techniques in numerical simulation. *Encyclopedia of Computational Mechanics*, 1:157–197, 2004.

[11] Albert Cohen, Sidi Kaber, Siegfried Müller, and Marie Postel. Fully adaptive multiresolution finite volume schemes for conservation laws. *Mathematics of computation*, 72(241):183–225, 2003.

[12] Ingrid Daubechies. *Ten lectures on wavelets*. SIAM, 1992.

[13] Dominique d'Humières. Multiple–relaxation–time lattice boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[14] Vadim Engelson, Peter Fritzson, and Dag Fritzson. *Lossless compression of high-volume numerical data from simulations*. LINKÖPING University Electronic Press, 2000.

[15] Clément Flint and Philippe Helluy. Reducing the memory usage of lattice-boltzmann schemes with a dwt-based compression. *ESAIM: Proceedings*, 2022.

[16] Nathaniel Fout and Kwan-Liu Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[17] Manuel Noronha Gamito and Miguel Salles Dias. Lossless coding of floating point data with jpeg 2000 part 10. In *Applications of Digital Image Processing XXVII*, volume 5558, pages 276–287. SPIE, 2004.

[18] LH Han, T Indinger, XY Hu, and Nikolaus A Adams. Wavelet-based adaptive multiresolution solver on heterogeneous parallel architecture for computational fluid dynamics. *Computer Science-Research and Development*, 26:197–203, 2011.

[19] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

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

[21] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, volume 22-3, pages 343–348. Wiley Online Library, 2003.

[22] TA Johnson and VC Patel. Flow past a sphere up to a reynolds number of 300. *Journal of Fluid Mechanics*, 378:19–70, 1999.

[23] Hyungmin Kang, Dongho Lee, and Dohyung Lee. A study on cfd data compression using hybrid supercompact wavelets. *KSME international journal*, 17:1784–1792, 2003.

[24] Fabian Knorr, Peter Thoman, and Thomas 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.

[25] Elias Konstantinidis and Yiannis Cotronis. A quantitative performance evaluation of fast on-chip memories of gpus. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455. IEEE, 2016.

[26] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. The lattice boltzmann method. *Springer International Publishing*, 10(978-3):4–15, 2017.

[27] Moritz Lehmann, Mathias J. Krause, Giorgio Amati, Marcello Sega, Jens Harting, and Stephan Gekle. Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats. *Phys. Rev. E*, 106:015308, Jul 2022.

[28] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data*, 9(2):485–498, 2022.

[29] Tilman Liebchen. Mpeg-4 als-the standard for lossless audio coding. *the Journal of the Acoustical Society of Korea*, 28(7):618–629, 2009.

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

[31] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[32] Yang Liu, Yueqing Wang, Liang Deng, Fang Wang, Fang Liu, Yutong Lu, and Sikun Li. A novel in situ compression method for cfd data based on generative adversarial network. *Journal of Visualization*, 22:95–108, 2019.

[33] Stéphane Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

[34] Siegfried Müller. *Adaptive multiscale schemes for conservation laws*, volume 27. Springer Science & Business Media, 2002.

[35] Seyyed Mahdi Najmabadi, Philipp Offenhäuser, Moritz Hamann, Guhathakurta Jajnabalkya, Fabian Hempert, Colin W Glass, and Sven Simon. Analyzing the effect and performance of lossy compression on aeroacoustic simulation of gas injector. *Computation*, 5(2):24, 2017.

[36] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[37] NVIDIA. nvbandwidth. `https://github.com/NVIDIA/nvbandwidth`, 2024.

[38] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. A new approach to the lattice boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638, 2011. Mesoscopic Methods for Engineering and Science — Proceedings of ICMMES-09.

[39] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Efficient gpu implementation of the linearly interpolated bounce-back boundary condition. *Computers & Mathematics with Applications*, 65(6):936–944, 2013. Mesoscopic Methods in Engineering and Science.

[40] Thomas Pohl, Frank Deserno, Nils Thurey, Ulrich Rude, Peter Lammers, Gerhard Wellein, and Thomas Zeiser. Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures. In *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 21–21. IEEE, 2004.

[41] Tran Minh Quan and Won-Ki Jeong. A fast mixed-band lifting wavelet transform on the gpu. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 1238–1242. IEEE, 2014.

[42] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[43] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[44] Ryotaro Sakai, Daisuke Sasaki, and Kazuhiro Nakahashi. Parallel implementation of large-scale cfd data compression toward aeroacoustic analysis. *Computers & Fluids*, 80:116–127, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[45] Ryotaro Sakai, Daisuke Sasaki, Shigeru Obayashi, and Kazuhiro Nakahashi. Wavelet-based data compression for flow simulation on block-structured cartesian mesh. *International Journal for Numerical Methods in Fluids*, 73(5):462–476, 2013.

[46] Vijay Sathish, Michael J Schulte, and Nam Sung Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 325–334, 2012.

[47] Jörg Schmalzl. Using standard image compression algorithms to store data from computational fluid dynamics. *Computers & geosciences*, 29(8):1021–1031, 2003.

[48] Kai Schneider and Oleg V Vasilyev. Wavelet methods in computational fluid dynamics. *Annual review of fluid mechanics*, 42:473–503, 2010.

[49] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[50] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A Ross. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 242–247. IEEE, 2016.

[51] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 2001.

[52] Wim Sweldens. Lifting scheme: a new philosophy in biorthogonal wavelet constructions. In *Wavelet applications in signal and image processing III*, volume 2569, pages 68–79. SPIE, 1995.

[53] Wim Sweldens and Peter Schröder. Building your own wavelets at home. In *Wavelets in the Geosciences*, pages 72–107. Springer, 2000.

[54] Shashank S Tiwari, Eshita Pal, Shivkumar Bale, Nitin Minocha, Ashwin W Patwardhan, Krishnaswamy Nandakumar, and Jyeshtharaj B Joshi. Flow past a single stationary sphere, 2. regime mapping and effect of external disturbances. *Powder Technology*, 365:215–243, 2020.

[55] Aaron Trott, Robert Moorhead, and John McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-d curvilinear grids. In *Proceedings of Seventh Annual IEEE Visualization'96*, pages 385–388. IEEE, 1996.

[56] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus. *arXiv preprint arXiv:2304.12557*, 2023.

[57] Baojie Zhu, Yifei Guan, and Jian Wu. Two-relaxation time lattice boltzmann models for the ion transport equation in electrohydrodynamic flow: D2q5 vs d2q9 and d3q7 vs d3q27. *Physics of Fluids*, 33(4), 2021.