FRSZ2 for In-Register Block Compression Inside GMRES on GPUs

Thomas Grützmacher[*] Robert Underwood[†] Sheng Di[†] Franck Cappello[†] Hartwig Anzt [‡]

[*]*Karlsruhe Institute of Technology*
Technical University of Munich
thomas.gruetzmacher@tum.de

[†]*Argonne National Laboratory*
University of Chicago
{runderwood, sdi1, cappello}@anl.gov

[‡]*Technical University of Munich*
University of Tennessee, Knoxville
hartwig.anzt@tum.de

***Abstract*—The performance of the GMRES iterative solver on GPUs is limited by the GPU main memory bandwidth. Compressed Basis GMRES outperforms GMRES by storing the Krylov basis in low precision, thereby reducing the memory access. An open question is whether compression techniques that are more sophisticated than casting to low precision can enable large runtime savings while preserving the accuracy of the final results. This paper presents the lightweight in-register compressor FRSZ2 that can decompress at the bandwidth speed of a modern NVIDIA H100 GPU. In an experimental evaluation, we demonstrate using FRSZ2 instead of low precision for compression of the Krylov basis can bring larger runtime benefits without impacting final accuracy.**

***Index Terms*—compression, FRSZ2, GMRES, CB-GMRES, high-performance, sparse, solver, hpc, GPU**

I. Introduction

The Generalized Minimal Residual Method (GMRES) is a popular method for solving linear systems of equations iteratively. GMRES is widely used in applications that result in large, sparse linear systems that are not symmetric positive definite. Systems of this kind are common in scientific computing applications, ranging from finite element discretizations over combinatorial problems to circuit simulations. GMRES builds up out of matrix-vector operations with the system matrix, vector operations, and orthogonalization, building up the Krylov subspace and a minimization process. All these building blocks are memory-bound, and thus the performance of the GMRES solver is limited by the main memory bandwidth of the processor. Consequently, strategies to accelerate the GMRES method aim to reduce the data transfer. In the Compressed Basis GMRES (CB-GMRES [1]) method, the Krylov basis vectors are compressed by conversion to lower precision formats, e.g., IEEE 754 single precision or half precision. This strategy reduces the data transfers of the individual iterations and allows for faster execution of the iterations while mostly preserving the quality of the final solution. The information loss caused by storing the Krylov basis in low precision can delay convergence, experiments however indicate that in most scenarios the convergence delay is easily compensated by the faster exection [1]. Converting the individual vector values to low precision is a straightforward compression scheme, and a valid question is whether more sophisticated compression schemes operating not on a value level but on a block level allow for higher compression rates or reduced information loss.

We thank the U.S. Department of Energy and the National Science Foundation for funding this work.

In this paper, we investigate this question by employing lossy compression for the compression of the Krylov vectors inside the CB-GMRES algorithm. Lossy compression maps the input data into a different representation with a smaller memory footprint through a series of computations that decorrelate and then encode common patterns in the data leaving a smaller representation. However, the compression has to happen in processor registers and needs to be extremely fast to not incur measurable overhead to the Krylov solver. In particular, all compression and decompression have to be hidden behind the memory access. A quick pen-and-paper calculation reveals this strategy to be viable: The latest Nvidia server line GPU, the Nvidia H100 GPU, has a memory bandwidth of roughly 2 TB/s and a peak double-precision performance of $\approx$ 25 TFLOP/s. That means an algorithm can execute up to 100 double-precision (64-bit) computations per double-precision value retrieved from main memory before hitting the compute peak. For a sparse linear algebra routine executing 4 operations on each retrieved value, one could consider 96 operations to be ”wasted”. Suppose some of these operations could be used to compress the data to consume only 32 bits per value. In that case, the compute-to-read ratio reduces to 50:1 and one could use 46 operations for compression and decompression of the double-precision values. While this sounds viable, several critical aspects constrain the design of the compression strategy. First, implementing compression and decompression in only 46 operations is not straightforward and excludes many sophisticated compression strategies. Second, the compressor must support at least random access by block to support the memory access patterns used within CB-GMRES. Third, the data values to be compressed in CB-GMRES are generally uncorrelated, presenting a challenge to effectively decorrelate.

This paper presents FRSZ2, a highly specialized compressor designed to support CB-GMRES. Our key contributions include:

1) We study the impacts of lossy compression error bounds on this problem to demonstrate that this problem prefers point-wise error bounds.

2) We identify the bottlenecks of using compression techniques for this problem and accommodate these in the compressor design.
3) We describe in detail the design of FRSZ2 and demonstrate how it integrates into Ginkgo's CB-GMRES.
4) We compare FRSZ2 to other compression techniques and demonstrate that FRSZ2 is the fastest compressor, achieving performance 99.6% of loading double precision from memory, which is $1.2 \sim 3.1\times$ faster than the next fastest compressor cuSZp2 at the roofline.
5) We report that for selected problems, FRSZ2 compression can render performance advantages of up to $1.3\times$ over single-precision compression in CB-GMRES.
6) We discuss how lossy compression could be used in a generic framework for accelerating CB-GMRES.

The remainder of the paper is organized as follows: In Section II, we describe CB-GMRES and describe in detail where compression can be applied. In Section III and Section IV, we present the challenges and how we tackle them in the design of FRSZ2. We then present our evaluation methodology in Section V before presenting the experimental results in Section VI. We provide an overview of related work in Section VII and conclude in Section VIII with a summary of the findings, challenges, and potential.

II. Background

Krylov methods are an essential building block in scientific high-performance computing for the rapid solution of large, sparse linear systems. They approximate the solution in a subspace that is generated iteratively in the Krylov solver iteration. Starting from an initial guess, each iteration adds a Krylov basis vector by orthogonalizing a new search direction against the already computed basis until the subspace spanned by the basis is large enough to contain a solution approximation of sufficient accuracy. For a problem of dimension n, the exact solution is available after n iterations, as then the Krylov basis spans the whole space. However, in practice, a much smaller number of iterations is typically sufficient to find a suitable solution approximation. Long recurrence Krylov methods, like the popular GMRES solver we focus on in this paper, build up the Krylov basis in main memory, and for each new search direction, the pre-existing basis has to be retrieved from main memory for the orthogonalization procedure such that it can then be appended to the extended basis. This makes the orthogonalization a memory-bound step that often dominates the overall solver runtime. A strategy to accelerate the Krylov solver can thus be to compress the data of the Krylov basis to reduce the main memory access volume. In [1], the authors realize this idea by storing the Krylov basis in low precision – a simple lossy compression technique. The GMRES algorithm and the compression potential is shown in Figure 1. Reducing the precision may incur perturbations in the Krylov basis, thereby harming the convergence. The experimental results, however, reveal that the slower convergence can typically be compensated by faster execution. Hence, more Krylov basis vectors can be generated in less time, thereby still accelerating the solution process. In this paper, we replace the compression of the vector entries based on casting to lower precision with a more sophisticated block-based compression. The hope is that higher compression ratios can be achieved while still hiding all compression and decompression behind the memory access.

```
1.  Compute $r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, and $v := r_0/\beta$. Set $V_1 = [v]$
2.  for $j := 1, 2, \ldots, m$
3.      Compute $w := A(M^{-1}v)$
4.      $\omega := \|w\|_2$
5.      Orthogonalize $h_{1:j,j} := V_j^T w, w := w - V_j h_{1:j,j}$
6.      $h_{j+1,j} := \|w\|_2$
7.      if $(h_{j+1,j} < \eta\,\omega)$ then
8.          Re-orthogonalize $u := V_j^T w, w := w - V_j u$
9.          $h_{1:j,j} := h_{1:j,j} + u$
10.         $h_{j+1,j} := \|w\|_2$
11.     endif
12.     if $(h_{j+1,j} = 0)$ or $(h_{j+1,j} < \eta\,\omega)$ then set $m := j$ and go to step 17, endif
13.     $v := w/h_{j+1,j}$
14.     Set $V_{j+1} := [V_j, v]$
15. endfor
16. Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = (h_{ij})_{1 \le i \le m+1, 1 \le j \le m}$
17. Compute $y_m$ the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$ and $x_m := x_0 + M^{-1}(V_m y_m)$
18. if satisfied then Stop, else set $x_0 := x_m$ and go to step 1, endif
```

Fig. 1: Algorithmic formulation of the restarted GMRES algorithm for solving sparse linear systems. Sections where compression can be used are highlighted.

III. Problem Formulation

Designing block compression inside of GMRES on the GPU presents several critical requirements on the compressor to produce a usable solution: 1) Quality: the compression error must not affect the solution accuracy, 2) Decorrelation: the compression needs to succeed in reducing the data volume without sacrificing too much information, and 3) Performance: The compression and decompression needs to be hidden behind the memory access to not incur any overhead to the CB-GMRES algorithm.

In the following subsections, we expand on the definition of the critical requirements of decorrelation and performance and how they impact the design of a compressor. We discuss quality later in Section VI.

A. Decorrelation

The Krylov vectors compressed in GMRES are difficult to decorrelate. Krylov vectors are normalized, which means all values are in $[-1, 1]$. Figure 2a-2d show the values and distribution of the values. While the first iteration of a solver may show some patterns in the data to be compressed[1], the values become uncorrelated in the subsequent iterations. There is no particular pattern to their ordering or values with both uniformly distributed over the domain.

To understand why these vectors are hard to decorrelate and the impact that has on the compressor design, it is helpful to consider how lossy compressors work. Modern lossy compressors feature three key stages to achieve high compression ratios: decorrelation, quantization, and encoding [2]. Decorrelation is a class of techniques that reduces autocorrelation in data and produces a new version with a distribution

[1]For example, the solver might initialize the vectors using values from a sin function.

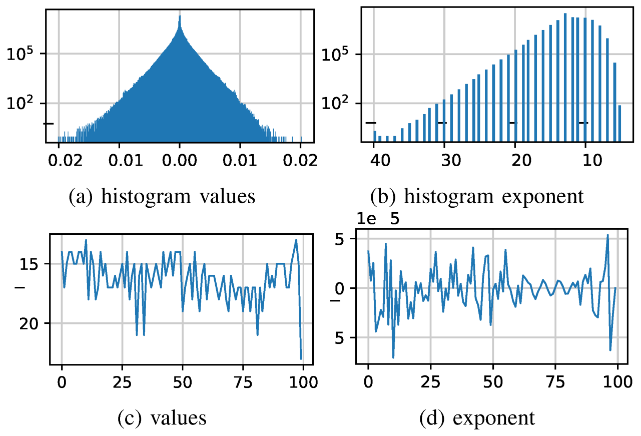


Fig. 2: Histogram of Exponents and values from the atmosmodd matrix. Only the exponent has a few common values, but values are normally distributed making decorrelation difficult.

that ideally has a much smaller variance that can then be quantized into fewer values, finally reducing entropy and allowing improved compression ratios. Each leading lossy compressor uses different decorrelation mechanisms. For example, SZ uses a collection of predictors (e.g., cubic spline [3], Lorenzo [4], block linear regression [5]) to predict later values with earlier values, and ZFP [6] uses a near orthogonal transform similar to JPEG. However, all of these methods rely on patterns in the values to reduce the entropy. Figure 2a reveals that these patterns do not exist for Krylov vectors. In consequence, the compression will be ineffective at best or counterproductive at worst. In the worst case, an ineffective decorrelation mechanism can introduce systematized decompression errors or increase compressed size [2].

Some compression is still possible, as, for example, a substantial fraction of the values in the Krylov basis share common exponents, see Figure 2d. This inspires a design that attempts to decorrelate the exponents but not the values. To the best of our knowledge, we present the first design for this kind of decorrelation scheme.

B. *Performance*

For this paper, we will define the performance challenge in terms of the speedup relative to the standard GMRES using IEEE 754 double precision for all arithmetic operations and for storing the Krylov basis vectors. Additionally, we will compare against the CB-GMRES algorithm storing the Krylov basis in single(float32)- and half(float16)-precision, respectively.

The performance envelope for compression is more aggressive than converting to single- or half-precision without increased information loss, is extremely tight. As explained in the introduction, there is time for only about 46 instructions to perform compression and decompression without affecting the runtime of the solver. This eliminates entire classes of encoding stages that use methods such as Huffman encoding and the embedded encoding used by ZFP that require far too many instructions, leaving only designs with more primitive truncation-based encoding schemes.

Notwithstanding the challenges with the decorrelation schemes used in other ultra-fast GPU compressors discussed previously in Section III-A, even the fastest compression schemes with simple encoding schemes such as cuSZp2 [7] are far too computationally complex and exceed the instruction limits. Even at its fastest configuration, cuSZp2 achieves only 1241GB/s on an A100 GPU, which is $\approx 80\%$ of its bandwidth. In a more typical case, it achieves closer to 500GB/s, which is $\approx 32\%$ of its bandwidth, making it too slow to be used in GMRES without an unacceptable slowdown.

Consequently, a highly specialized high throughput compression algorithm is required to compete with compression based on converting to low precision.

IV. Design

In this section, we detail the new compression format FRSZ2 that is fast in decompression while retaining more information per value than IEEE float32 for GMRES. In order to decorrelate the exponent information of values efficiently, the format initially evaluates the universally used IEEE 754 double-precision format float64. Each float64 can be separated into its sign s, 11-bit unsigned exponent e, and 52 significand bits $b_{51} \dots b_1 b_0$. Formula 1 is used to compute the represented value for the most common format:

$$\text{value} = (-1)^s \cdot (1.b_{51} \dots b_1 b_0)_2 \cdot 2^{e-1023} \tag{1}$$

The exponent is stored in an offset-binary representation, which means it is stored as an unsigned integer and needs to be subtracted by an offset number to get the actual value. For float64, this offset is 1023. The significand in (Equ. 1) has a leading 1 bit, which is not explicitly stored.

The compression aims to group multiple values into a block and extract their exponent [3]. Freely choosing which values to group is impossible because that would reduce our decompression speed for consecutive values. The idea is that neighboring Krylov vector values are likely close in magnitude, which means they can be grouped together. The block size is fixed to avoid global synchronization points and increase the throughput on GPUs, which are massively parallel architectures. This block size BS is one of two optimization parameters of FRSZ2. An effective value will be determined in Section IV-C.

To account for small differences in the magnitude of values in a block, the maximum IEEE exponent $e_{\max}$ is identified, and all values are normalized with this exponent. This implies that, in contrast to the IEEE format, the significands are not normalized to sub-unit values, but the integer part of the

[2]e.g. from space usage overheads from the unpredictable data correction mechanisms in prediction scheme-based methods like SZ

[3]block floating point implementations are not novel. They are used in ZFP [6] and proposed as early as 1964 in [8]. We differ from ZFP in that we do not feature a decorrelation stage which is counterproductive for this data and our unique data layout. Our approach differs from [8] in our data layout

significand of the represented numbers has to be stored. For each number in a block that has a smaller exponent $e < e_{\max}$, we need to prefix the significand with $k = e_{\max} - e$ zeros in order to represent the same value. We limit the number of bits per value, which includes the sign and significand, to a fixed length l. l adjusts the compression ratio and the maximum precision retained per value. We evaluate and advocate for different l in Section IV-C.

We represent the compressed value, consisting of the sign bit and the significand, with the symbol c. For l bits, the compressed value $c_{l-1} \dots c_1 c_0$ represents the following number:

$$\text{value} = (-1)^{c_{l-1}} \cdot (c_{l-2}.c_{l-3} \dots c_1 c_0)_2 \cdot 2^{e_{\max}} \tag{2}$$

To summarize, the sign bit is stored as the most significant bit of c, followed by the significand's integer part and then the significand's fractional part. BS and l are the two optimization parameters of FRSZ2. For increased memory access speed, we read and write our memory as integers with at least l bits, which requires the beginning of a block to be aligned to that value. The exponent is stored in integer representation. For $16 < l \leq 32$, we use an integer type with 32 bits, so the memory needs to be aligned to 4 bytes. The storage requirement in bytes for n elements and an assumed integer representation with 4 bytes is:

$$\underbrace{\left\lceil \frac{n}{\text{BS}} \right\rceil \cdot \left\lceil \frac{\text{BS} \cdot l}{4} \right\rceil \cdot 4}_{\text{compressed values}} + \underbrace{\left\lceil \frac{n}{\text{BS}} \right\rceil \cdot 4}_{\text{exponents}} \tag{3}$$

A. Compression

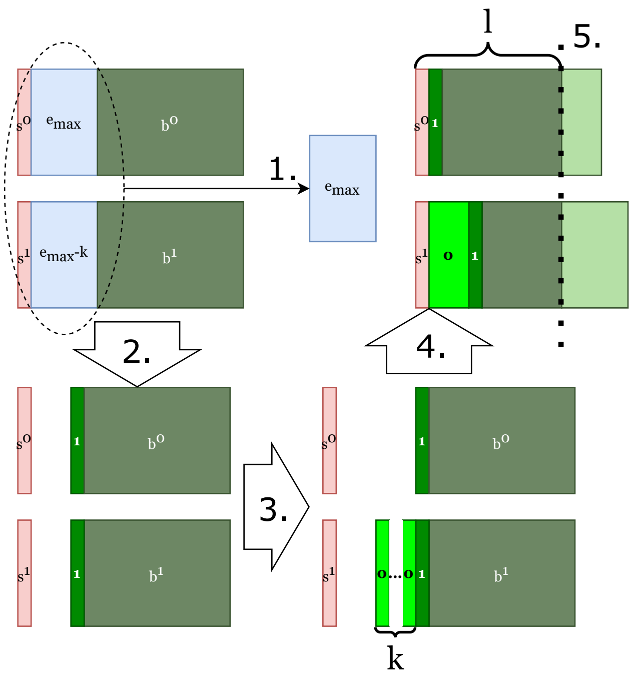


Fig. 3: FRSZ2 compression steps (BS = 2 and arbitrary $l > 2$).

The compression algorithm performs the following 6 steps:

1) Extract the exponent e and find the maximum exponent $e_{\max}$ from all values in the block;
2) Extract the sign s and significand; add the usually implicit 1 bit to the significand representation;
3) Normalize the significand to the maximum exponent $e_{\max}$ by prefixing the significand with $k = e_{\max} - e$ 0 bits;
4) Put the sign bit to the left of the normalized significand;
5) Cut the new representation to the appropriate length l. Now, we have c;
6) Store $e_{\max}$ and all c of the block.

An illustration of this process is provided in Figure 3. If l does not match the size of the integer representation type exactly (e.g. $l = 21$), step 6 needs to merge neighboring values before storing them since GPUs can only store values at a byte level.

The compression must be performed on all BS elements simultaneously to efficiently utilize the GPU bandwidth. Updating just a single element would require reading $e_{\max}$ before writing the compressed value. If $e_{\max}$ changes as a result, all values of the same block need to be read from memory, renormalized to this new exponent, and written back to memory.

B. Decompression

Decompression is an easier procedure and does not require the full block to be read simultaneously. The following steps retrieve a value at index i:

1) Read $e_{\max}$ for the corresponding block and read the correct compressed value c at index i;
2) Separate c into the sign bit s and the significand; Count the number of inserted zeros k at the beginning of the significand;
3) Remove the inserted zeros and the explicit 1 bit from the significand; Compute the actual exponent $e = e_{\max} - k$;
4) Merge s, e, and the corrected significand back to an IEEE double-precision value.

As not the complete block has to be decompressed to retrieve one value, random access is possible. To decompress value c at index i, the only overhead is that e_{max} must also be retrieved from the main memory. However, the most efficient access is to read the whole block and reuse the cached $e_{\max}$ to decompress all values in the block.

C. Implementation and synthetic performance

To ensure good performance on the Nvidia H100 GPU, we perform the following optimizations: (1) We utilize intrinsic functions to convert between the IEEE format and the appropriate integer type so we can analyze it bit-by-bit. The intrinsic function to count the leading zeros of an integer: *count_zero* is also mandatory for good performance. (2) We mandate a block size $BS = 32$ for Nvidia GPUs. This allows us to use warp-shuffles, the fastest communication between threads, to determine $e_{\max}$ during compression. Additionally, it guarantees that $e_{\max}$ is cached for all threads of the warp during decompression of the same block. (3) Have separate compression and decompression routines for $l = 2^x$ and $l \neq 2^x$. For $l = 2^x$, compression and decompression are much simpler because values do not interleave in memory, making reading and writing them significantly faster. (4) Perform all index computations in 32-bit integer types. Originally, we used 64-bit, but those are noticeably slower than 32-bit on an H100. (5) Store the exponent for the blocks and the compressed values in separate memory locations, which simplifies the index computations substantially.

We implement the FRSZ2 format in C++ and CUDA. Additionally, for the decompression, we can utilize the Accessor interface in Ginkgo [1], [9], which is a software interface that decouples the storage format from the arithmetic format. So far, it has been used to store values in half- or single-precision while performing all computations in double-precision. The same interface is used for reading and decompressing data in FRSZ2 while computing in double-precision because decompression does not require any form of communication with other threads. We need to read the complete block of values to maximize cache usage, but that is already the access pattern for the Krylov vectors, so we do not need to treat them differently when we use FRSZ2. Writing and compressing data can not be handled through the Accessor interface because it was designed for random access in both directions. Our compression must be applied to a full block of values and requires local communication to find $e_{\max}$. Figure 1 is the implemented GMRES algorithm, which also highlights all sections that compress or decompress data.

We evaluate the usage and efficiency of the FRSZ2 decompression inside the Accessor interface with a synthetic benchmark that reads consecutive elements from the main memory and executes a pre-defined number of arithmetic operations on each value retrieved from main memory to vary the arithmetic intensity. For each storage format, we run this benchmark for 27 arithmetic intensity settings. In Figure 4, we report for increasing arithmetic intensity the performance of the kernel using different storage formats. We use an array with 2^{28} randomized elements to utilize the full GPU. Each data point in the plot is the minimum from 10 individual executions. The resulting roofline analysis allows us to compare the performance and memory bandwidth of the different storage and compression formats. We note that float64 and float32 do not use the Accessor interface but read and compute in their respective precision directly. This allows us to evaluate the overhead of the Accessor. *Acc<float64>* and *Acc<float32>* compute in double-precision while storing the values in float64 and float32, respectively. The performance of the Accessor is identical to the native implementation as long as they are memory-bound, which proves the zero-cost abstraction. For FRSZ2, we always use BS = 32 and three different bit lengths: $l \in \{16, 21, 32\}$. We chose 16 and 32 for their value alignment, making their decompression less complex, and 21 to observe the penalty of the additional decompression steps.

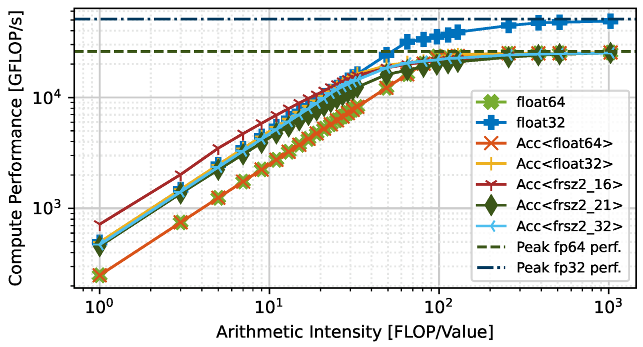


Fig. 4: Performance on the H100

$l = 16$ shows the highest performance per value. However, it is not a factor of 2 faster than the single-precision storage, which means we do not saturate the full bandwidth. Additionally, the gap between float64 and *Acc<frsz2_16>* decreases rapidly with higher arithmetic intensity, so it is unsuitable for workloads with slightly higher arithmetic intensity. *Acc<frsz2_32>* achieves slightly lower performance than *Acc<float32>*. The reason is simple: frsz2_32 needs 33 bits per value on average because it needs to store one exponent value, which occupies 32 bits as well, per block, and since BS = 32, the average bits per value is: $(\mathrm{BS}{\cdot}l{+}32)/\mathrm{BS} = (32{\cdot}32{+}32)/32 = 33$. This difference could be explained by the additional exponent that needs to be read by block, which translates to roughly 33 bits per value for frsz2_32. When measuring the bandwidth instead of performance, *Acc<frsz2_32>* reaches 1991GB/s, which is $\approx 99.6\%$ of the reachable bandwidth. This confirms the viability of our compression target: Our decompression algorithm can saturate the bandwidth and has cycles to spare for many additional floating-point computations.

Acc<frsz2_21> displays a similar performance to *Acc<frsz2_32>* despite reducing the memory footprint by $\approx 33\%$. This clearly states that the overhead in the more complex index computation and the unaligned memory read operation is too high to translate to higher performance. frsz2_21 will always be less precise than frsz2_32, which means frsz2_21 is only useful in case frsz2_32 would not fit in GPU memory. We have not encountered any problem large enough to reach the GPU memory capacity, so its usefulness is limited.

V. Methodology

This section describes common details to reproduce our experiments and motivations for these choices. We begin with hardware and software choices, then discuss the choice of benchmark problems for evaluation, and finally, we discuss the configuration of compressors used in the comparison.

A. Hardware and Software

We choose the latest NVIDIA H100 GPU for our experimental evaluation. We also considered other GPUs, but we present results only for the H100 for brevity. The results for

Matrix	Size	Non-zeros	target RRN
atmosmodd	1,270,432	8,814,880	$4.0 \cdot 10^{-16}$
atmosmodj	1,270,432	8,814,880	$4.0 \cdot 10^{-16}$
atmosmodl	1,489,752	10,319,760	$4.0 \cdot 10^{-16}$
atmosmodm	1,489,752	10,319,760	$4.0 \cdot 10^{-16}$
cfd2	123,440	3,085,406	$1.8 \cdot 10^{-10}$
HV15R	2,017,169	283,073,458	$1.6 \cdot 10^{-02}$
lung2	109,460	492,564	$1.8 \cdot 10^{-08}$
parabolic_fem	525,825	3,674,625	$4.0 \cdot 10^{-16}$
PR02R	161,070	8,185,136	$4.0 \cdot 10^{-03}$
RM07R	381,689	37,464,962	$8.0 \cdot 10^{-03}$
StocF-1465	1,465,137	21,005,389	$4.0 \cdot 10^{-06}$

TABLE I: Details of the computational fluid dynamic matrices used from SuiteSparse

other GPUs are not meaningfully different in conclusion. The Nvidia H100 is the PCIe variant with 80 GB of RAM, 50 MB of L2 cache, 25.6 TFLOP/s double-precision, and 51.2 TFLOP/s single-precision performance. The peak memory bandwidth is 2000 GB/s. The host system is a server with two Intel Xeon Silver 4309 processors.

We use the default compilers on the system: CUDA 12.1 and GCC 11.4. We choose the default (latest) packages from spack. We utilize LibPressio [10] version 0.98.0 to manage and interact with the other compression algorithms we use: sz version 2.1.12.5, sz3 version 3.1.7 and zfp version 1.0.0.

Our CB-GMRES implementation with FRSZ2 and the benchmark code is open-source and can be accessed in the Ginkgo branch `2024-drbsd-paper`[4].

B. Problem Selection

The matrices we use are from the SuiteSparse matrix collection [11]. It is widely used for various sparse benchmarks because of its vast size (2,893 matrices) and diversity in domains (48). We focus on matrices that solve computational fluid dynamics problems because they worked poorly in [1] and aim to improve the convergence rate with our FRSZ2 compression. As an additional restriction, they need to have a matrix size of more than $100,000$ rows to avoid caching mechanisms blurring the understanding of the performance analysis. The most important properties of the matrices we use can be seen in table I.

For each matrix A, we generate the right hand side b deterministically and identical to [1] to ensure fair comparisons: First, we take a vector s and set the i-th entry to its sin value: $s[i] = \sin(i)$ for $i \in \{0, 1, \ldots, n-1\}$. The expected solution x_{sol} is gained by normalizing s with its unit norm: $x_{\text{sol}} = s/\|s\|_2$. The right-hand side b is computed by multiplying the expected result with the matrix A: $b = A \cdot x$. All GMRES algorithms are started with the initial guess $x_0 = \vec{0}$, using a restart parameter $m = 100$[5], and are stopped when the solution approximation x fulfills the relative residual norm specified in Table I: $\|A \cdot x - b\|_2 \leq \text{RRN} \cdot \|b\|_2$.

[4]https://github.com/ginkgo-project/ginkgo/tree/2024-drbsd-paper

[5]To limit the memory requirements, GMRES is restarted after a Krylov basis of 100 vectors has been built up. The restart uses the latest solution approximation as initial guess, and starts building up a new Krylov search space.

C. Solver Configuration

GMRES is an iterative solver that tries to solve the equation $A \cdot x = b$ to a predefined accuracy. The sparse matrix A and the right-hand-side vector b are problem-dependent and immutable inputs to the solver. GMRES also needs an initial guess x_0 as a starting vector, which is then iteratively improved to get closer and closer to the solution x_{sol}. The residual r is the difference vector between the targeted result b and the result achieved with the current approximation of x. To quantify the quality of the current approximation of x in a single number, we compute the relative residual norm (RRN):

$$\text{RRN} = \frac{\|r\|_2}{\|b\|_2} = \frac{\|b - A \cdot x\|_2}{\|b\|_2} \tag{4}$$

This value is given to the GMRES algorithm to determine when the computed solution approximation is sufficiently accurate. The lower RRN, the closer we are to the exact solution. The ideal case is RRN $= 0$. However, we compute with finite IEEE double-precision, with a unit roundoff of $u \approx 10^{-16}$, thus RRN $= 0$ may not be achievable. Additionally, some problems are inherently difficult to solve, so we adjust our target accuracy for each problem. For this, we solve each problem with $20,000$ iterations of a standard double-precision GMRES. The solution accuracy achieved is then used with some wiggle room as the stopping criterion for the CB-GMRES variants using different storage formats for the Krylov basis.

Table I lists the obtained targeted relative residual norms we use from this point forward.

We do not use any preconditioner to not blur the numerical impact by the use of a sophisticated preconditioner.

D. Compression Configuration

Details about the FRSZ2 compression are outlined in Section IV. Every decompression happens through the Accessor interface, while the compression is called directly without an intermediate interface. *frsz2_XX* corresponds to the FRSZ2 format with BS $= 32$ and $l = XX$. We also experimented with different block sizes, but the end-to-end runtime worsens with block sizes different than 32 elements, so we focus purely on BS $= 32$.

We also run all experiments with the original CB-GMRES storage formats: float64, which stores the values in IEEE double-precision storage format; float32, which stores values in single-precision; and float16, which store values in half-precision. All these options use the Accessor to have varying storage formats, but all arithmetic calculations are still done in IEEE 754 double precision.

While we also want to evaluate the compression efficiency of other compression schemes, implementing these in the Accessor interface would require a substantial amount of work. Thus, we decided to simulate the effect of other compression schemes on the CB-GMRES convergence by using them via LibPressio [10]. We do this by compressing and immediately decompressing the Krylov vectors through the LibPressio interface. This helps us to analyze the loss of information of various compressors without the need to implement any of them. We focus on SZ, SZ3, and ZFP because they are leading lossy compressors with multiple error bounds supported while using different decorrelation strategies.

We experiment with many error-bound settings for SZ, SZ3, and ZFP. Many behave the same or at least very close to the others, so we chose the settings shown in Table II.

Name	**error-bound type**	**error-bound**
sz3_06	absolute	10^{-06}
sz3_07	absolute	10^{-07}
sz3_08	absolute	10^{-08}
zfp_06	absolute	$1.4 \cdot 10^{-06}$
zfp_10	absolute	$4.0 \cdot 10^{-10}$
sz_pwrel_04	relative	10^{-04}
sz3_pwrel_04	relative	10^{-04}
zfp_fr_16	fixed rate	16 bits
zfp_fr_32	fixed rate	32 bits

TABLE II: Compressor name and requested bounds.

VI. Evaluation

There are two classes of experiments to perform: comparisons of the quality of the solution and end-to-end performance. GMRES requires a compression format that reduces data transfers without impacting neither the accuracy of the final result nor requiring significantly more iterations to reach convergence. Lastly, we consider the total algorithm runtime.

A. *Quality of Solution*

First, we compare the accuracy of FRSZ2 with the other compression schemes presented in [3], [6], [12] for different absolute error-bound settings to understand their impact on conversion rates. We choose the matrix *atmosmodd* here because this is one of the rare test problems where storing the Krylov basis in single precision impacts the accuracy of the final result [1]. Figure 5 displays the relative residual norm development throughout the GMRES solve. For *atmosmodd*, we target a relative residual norm of $4 \cdot 10^{-16}$. This target is reached faster if the compression preserves more information.

For *atmosmodd*, the convergence rates for the different compression schemes vary substantially. The convergence rate of frsz2_32 is close to matching the convergence rate of the uncompressed float64 storage. frsz2_32 improves over float32 despite using the same space. We can attribute this improvement in convergence over float32 to the increased space to store precision information created by externalizing the exponent to the block.

Considering the other absolute error bounded compressors, none of the ZFP or SZ3 settings manage to match the convergence of the float32 compression, even though *sz3_08* uses 46 bits per value on average, compared to the 32 from float32. *zfp_10* outperforms *sz_08* both in convergence and compression rate as it only uses 28 bits per value. We attribute the slower convergence of SZ and ZFP to these compressors' ill-fated attempts to predict or decorrelate the uncorrelated Krylov vectors, resulting in a bias in the reconstructed values.

Next, we investigate the pointwise relative error bounds in Figure 6. We observe that the pointwise error bounds

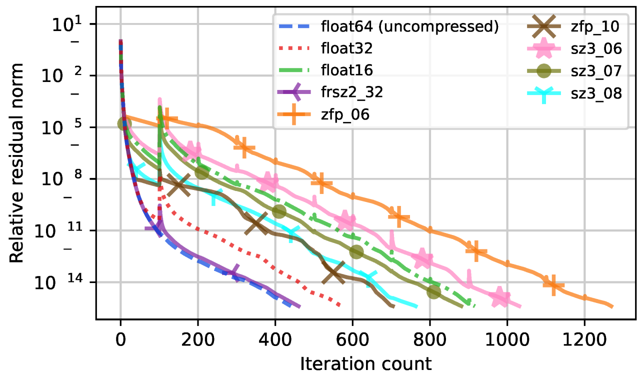


Fig. 5: Residual norm development for the *atmosmodd* matrix with various compressions.

enable better convergence rates than absolute error bounds. The pointwise relative error preserves $x(1-\epsilon) \leq \tilde{x} \leq x(1+\epsilon)$. Consequently, the values' magnitude is better preserved than using the absolute error bound, which is more similar to our FRSZ2 approach. Still, though the convergence rates are improved, none of the compressors can match thefloat32 compression in terms of GMRES convergence. The fixed-rate ZFP compression achieved the best convergence rate among the other compressors. It occupies the same memory as float32 but retains slightly less information for this application. frsz2_32 has the best convergence rate among all tested compression techniques.

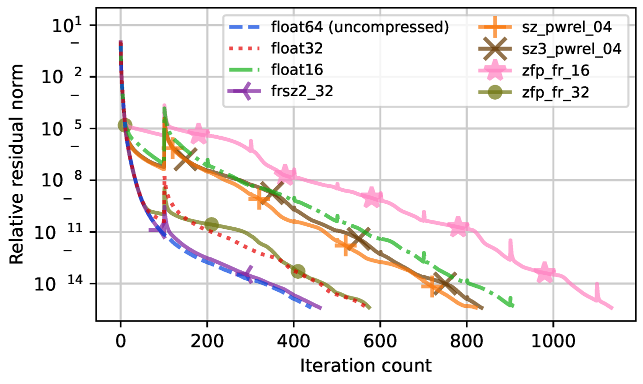


Fig. 6: Residual norm development for the *atmosmodd* matrix with pointwise relative error settings.

After showing that frsz2_32 improves convergence for one test problem, we investigate whether this effectiveness generalizes to other problems. Figure 7 presents the target and the achieved relative residual norm for the storage formats float64, float32, float16, and frsz2_32 for all considered matrices. The exact target relative residual norm is listed in Table I. We observe two instances where we do not reach the targeted relative residual norm with float16: *PR02R* and *StocF-1465*. The loss of information is too significant for these problems. For the other matrices, all settings converge to the target

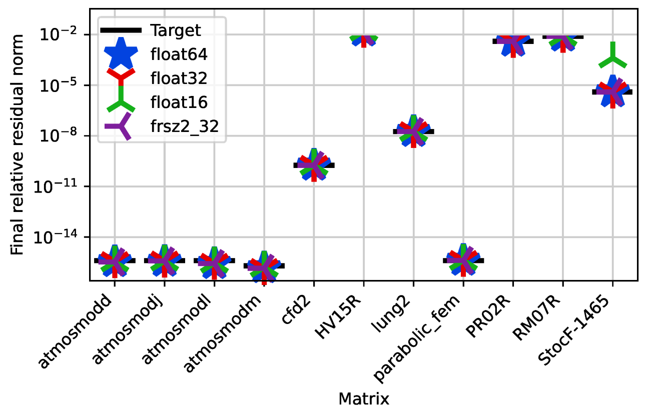


Fig. 7: Final relative residual norm for various matrices on the H100.

precision.

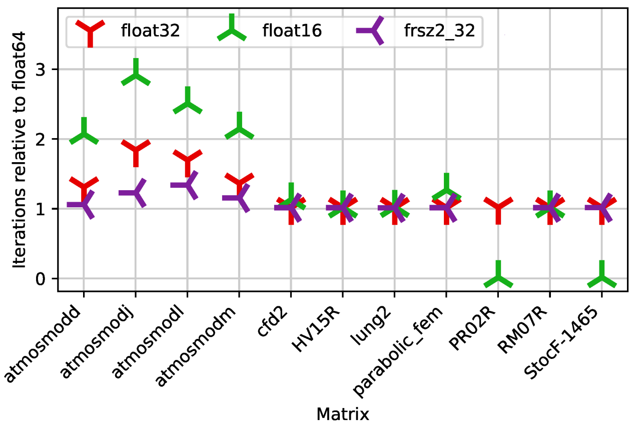
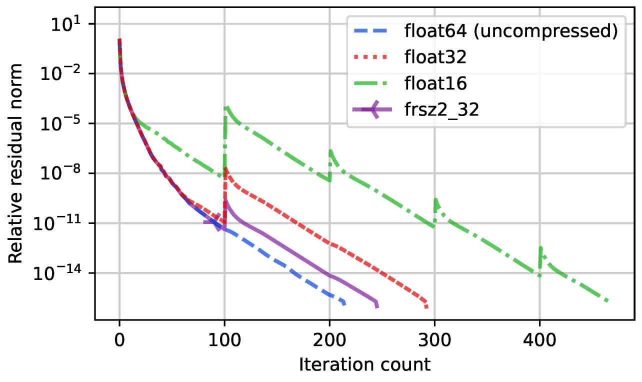


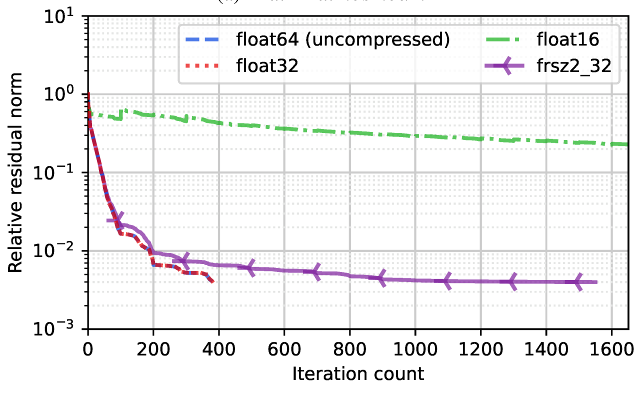
Fig. 8: Mean number of iterations to the solution of various matrices for the H100 over 10 runs. Zero means the solver does not reach the target precision.

Next, we compare the convergence rate with the float64 convergence rate. We do this by showing the number of iterations each storage type needs to achieve the target precision as a factor of the reference float64 in Figure 8. We set the relative value to zero if the target accuracy is not achieved. We observe that all matrices with the prefix *atmosmod* behave similarly: float64 converges fastest, followed by frsz2_32, then float32, and finally float16. Here, frsz2_32 is clearly the best compression format because it comes with the smallest iteration overhead. In contrast, *PR02R* is the worst problem for FRSZ2. frsz2_32 eventually converges to the target norm, but the iteration count increases by $3.5\times$. All other matrices barely show a difference in convergence rate.

We now focus on the test cases where frsz2_32 performs extremely well and extremely badly. Figure 9 provides a deeper insight into the convergence rate by plotting the relative residual norm for each iteration for the matrices where frsz2_32 works well, represented by *atmosmodm* in Figure 9a, and the matrix *PR02R* in Figure 9b.



(a) Matrix *atmosmodm*



(b) Matrix *PR02R*

Fig. 9: Relative residual norm development for the best- and worst-performing matrices for FRSZ2.

atmosmodm has a big residual norm correction after the first restart at iteration 100 for all storage formats except for the uncompressed float64. These corrections exist because the residual norms are only explicitly computed at every restart in GMRES, which we do every 100 iterations. For all other iterations, it only adjusts the previous residual norm by the assumed amount of improvement. During the restart, the residual and its norm are explicitly computed and used as the new baseline. These adjustments are the sudden jumps in Figure 9. frsz2_32 seems to recover from that correction the fastest from all the other compressions and only requires 31 more iterations to achieve the same accuracy as the uncompressed at convergence. The order from best to worst seems sorted by the number of significand bits for each compression scheme.

Figure 9b shows a different side of the compression scheme. Here, frsz2_32 follows both the single- and double-precision storage format until a relative residual norm of $2 \cdot 10^{-2}$ is reached, then stagnates. It barely improves the residual norm between iterations 400 and 1600, which might indicate that it reached its maximum accuracy. Half-precision does not even reach an accuracy of 10^{-2}. Even after $20,000$ iterations, it only managed to go down to $5 \cdot 10^{-1}$.

Part of the reason might be the huge range of non-zero values the matrix *PR02R* has. Figure 10 visualizes the exponent distribution of matrix *PR02R*, which ranges from -178 to 36. If a block of Krylov basis values contains exponents with a large range, we lose a lot of precision in values with smaller exponents when we fill the significands with zeros in the normalization step. However, this is not the only contributing factor. The matrix *HV15R* has an extremely similar value distribution to *PR02R*. The ordering of non-zero values in *HV15R* may lead neighboring Krylov vector values to have a similar magnitude, mitigating the effects observed in *PR02R*.

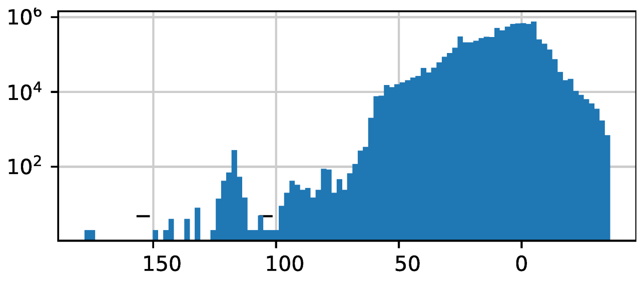


Fig. 10: Base-2 exponent histogram of all non-zero values of *PR02R*.

B. End-to-End Performance

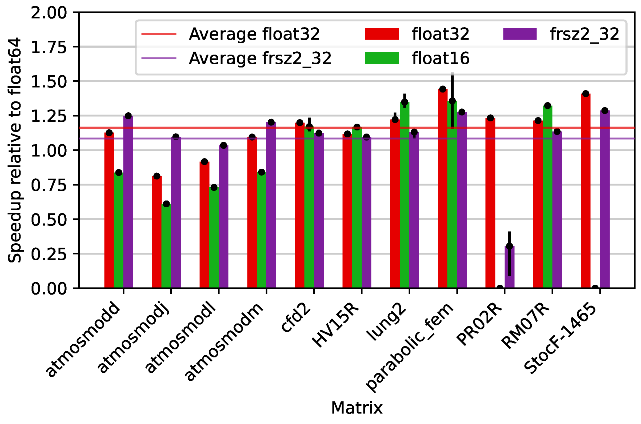


Fig. 11: Mean speedup of various matrices for the H100 with an error bar. Each matrix was solved ten times.

Finally, we investigate the end-to-end speedup achieved when using frsz2_32. Looking back at the read performance in Figure 4 and combining that with the usually same number of iterations in Figure 8, we expect the frsz2_32 performance to be similar to using single precision for compression. The mean speedup compared to double-precision storage is shown in Figure 11 with error bars. The entire bar is removed from a matrix if a storage format does not reach the targeted relative residual norm. As expected, frsz2_32 performs well in the *atmosmod* group. It is faster than single-precision storage and also beats the double-precision storage format. For problems outside the *atmosmod* group, frsz2_32 is consistently slower than the float32 storage format.

The average speedup over all matrices for the float32 storage format is 1.16, while the average is 1.09 for frsz2_32. Ignoring matrix *PR02R*, the average speedup increases to 1.16.

We also experimented with a variant of our method using 21 bits, the frsz2_21 storage format. Experiments revealed that the convergence for frsz2_21 is superior to float16, but dramatically slower than frsz2_32 due to worse alignment. We, therefore, omit the experimental results from the evaluation.

VII. Related Work

There is a rich history of using lossy compression to accelerate computations. One paper used lossy compression to effectively expand the memory by compressing key data structures instead of recomputing them when they could not all fit in memory [13]. A more recent paper used compression to speed up I/O [14] even if the resulting operations were slower to make it possible on a given set of resources. For accelerating memory-bound algorithms, the compression has to happen in processor registers, without touching main memory for either compression or decompression. In both of these papers, the operation that was avoided by using compression was substantially slower than a single access from the HBM on the GPU. We in contrast present a uniquely challenging case where the operation being modified is itself fast.

The idea of in-register data compression to speed up memory-bound linear algebra operations was initially realized by casting data to lower precision. In [15], the authors accelerate a block-Jacobi preconditioner by storing the individual block-inverses in lower precision, while keeping high precision for the arithmetic operations. The same strategy was later used for accelerating sparse approximate inverse preconditioners [16]. Similarly, the idea of compressing the Krylov basis of a GMRES iterative solver was initially using low precision for compression of the vector values [1]. Almost at the same time, this concept was proposed also by [17], however following a more sophisticated strategy by storing the preconditioned Krylov vectors inside a flexible GMRES solver in low precision. This improves the numerical stability at the price of reduced runtime benefits. Casting to lower precision can render only a moderate compression ratio without losing too much information. This motivates the use of sophisticated compression strategies. In [18] we investigated the use of advanced compression strategies yielding large compression factors. The results, however, indicated that these methods are too clumsy to operate on GPU registers, and the compression factors were too small to translate to speedup factors. We hence developed a more lightweight compression drawing a good balance between compression ratio and compression cost and presented the results in this paper.

However, the alternative was not to perform the computation at all. In GMRES, the inputs are 1) not bound by GPU memory size indicating the problem would be unlikely if it exists, and 2) have the clear alternative of performing the calculation in mixed or lower precision such as Float32 achieving a similar outcome without the possible overheads of compression.

VIII. Conclusions and Future Work

In this paper, we present FRSZ2, a highly specialized compressor for GMRES, that provides unparalleled performance among modern compressors and is uniquely capable of accelerating end-to-end performance of GMRES for a class of applications up to $1.3\times$ compared to uncompressed methods as well as $1.2 \sim 3.1\times$ faster than existing compressors obtaining 99.6% of the peak bandwidth at the roofline.

However, more work is needed to realize FRSZ2 as a generalizable solution for use in GMRES solvers in packages such as Ginkgo. Either 1) there needs to be continued work to accelerate FRSZ2 even further – this could come in the form of additional hardware improvements for certain assembly instructions (i.e., masked shuffle operations) used in decompression routines or algorithmic improvements that could eliminate the dependence on these slow instructions or changes to the balance between memory and compute bandwidth 2) we need an accurate, robust, and fast method to predict when an application will benefit from FRSZ2 compared to mixed-precision methods.

Given the degree of optimization already applied to FRSZ2, we believe these benefits are most likely to come from predictions that can be applied just before the first restart. We explored this briefly in our work prior to submission. We considered features such as the condition number, value distribution, exponent distribution, and even autotuned methods that detect and observe the convergence per unit time of several candidate methods and then speculatively execute that the best initial method will continue to dominate. We have only scratched the surface of possible methods, and with further work, an appropriate prediction method may be identified.

Acknowledgment

This work was performed on the NHR@KIT Future Technologies Partition testbed funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The authors thank the Innovative Computing Lab for access to their NVIDIA H100 GPU.

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based upon work supported by the National Science Foundation under Grant No. #2311875 and #2104203. This work has been conducted within the Joint Laboratory for Extreme-Scale Computing (JLESC).

References

[1] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás, "Compressed basis GMRES on high-performance graphics processing units," *The International Journal of High Performance Computing Applications*, pp. 1–18, Aug. 2022.

[2] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, pp. 1201–1220, Nov. 2019. Number: 6.

[3] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello, "SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors," *IEEE Transactions on Big Data*, vol. 9, pp. 485–498, Apr. 2023. Conference Name: IEEE Transactions on Big Data.

[4] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 730–739, May 2016.

[5] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1129–1139, May 2017.

[6] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, pp. 2674–2683, Dec. 2014. Number: 12.

[7] Yafan Huang, Sheng Di, Guanpeng Li, and Franck Cappello, "cuSZp2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio," in *roceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, (Atlanta, GA, USA), pp. 1–14, IEEE, Nov. 2024.

[8] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Prentice-Hall, 1964. Google-Books-ID: cBo1AAAAIAAJ.

[9] T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí, "Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS," *Software - Practice and Experience*, no. September, pp. 1–18, 2021.

[10] R. Underwood, V. Malvoso, J. C. Calhoun, S. Di, and F. Cappello, "Productive and Performant Generic Lossy Data Compression with LibPressio," in *2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7)*, (St. Louis, Missouri), pp. 1–10, IEEE, Nov. 2021.

[11] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, Nov. 2011.

[12] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An Efficient Transformation Scheme for Lossy Data Compression with Point-Wise Relative Error Bound," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 179–189, Sept. 2018.

[13] A. M. Gok, S. Di, Y. Alexeev, D. Tao, V. Mironov, X. Liang, and F. Cappello, "PaSTRI: Error-Bounded Lossy Compression for Two-Electron Integrals in Quantum Chemistry," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–11, Sept. 2018.

[14] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), pp. 1–24, Association for Computing Machinery, Nov. 2019.

[15] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, p. e4460, 2019.

[16] F. Göbel, T. Grützmacher, T. Ribizel, and H. Anzt, "Mixed precision incomplete and factorized sparse approximate inverse preconditioning on gpus," in *Euro-Par 2021: Parallel Processing* (L. Sousa, N. Roma, and P. Tomás, eds.), vol. 1, pp. 550–564, 2021.

[17] E. Agullo, F. Cappello, S. Di, L. Giraud, X. Liang, and N. Schenkels, "Exploring variable accuracy storage through lossy compression techniques in numerical linear algebra: a first application to flexible GMRES," Research Report RR-9342, Inria Bordeaux Sud-Ouest, May 2020.

[18] F. Cappello, S. Di, R. Underwood, D. Tao, J. Calhoun, Y. Kazutomo, K. Sato, A. Singh, L. Giraud, E. Agullo, X. Yepes, M. Acosta, S. Jin, J. Tian, F. Vivien, B. Zhang, K. Sano, T. Ueno, T. Grützmacher, and H. Anzt, "Multifacets of lossy compression for scientific data in the Joint-Laboratory of Extreme Scale Computing," *Future Generation Computer Systems*, 2024.