ADAPTIVE PRECISION SPARSE MATRIX–VECTOR PRODUCT AND ITS APPLICATION TO KRYLOV SOLVERS*

STEF GRAILLAT†, FABIENNE JÉZÉQUEL‡, THEO MARY†, AND ROMÉO MOLINA§

Abstract. We introduce a mixed precision algorithm for computing sparse matrix-vector products and use it to accelerate the solution of sparse linear systems by iterative methods. Our approach is based on the idea of adapting the precision of each matrix element to their magnitude: we split the elements into buckets and use progressively lower precisions for the buckets of progressively smaller elements. We carry out a rounding error analysis of this algorithm that provides us with an explicit rule to decide which element goes into which bucket and allows us to rigorously control the accuracy of the algorithm. We implement the algorithm on a multicore computer and obtain significant speedups (up to a factor 7×) with respect to uniform precision algorithms, without loss of accuracy, on a range of sparse matrices from real-life applications. We showcase the effectiveness of our algorithm by plugging it into a GMRES solver for sparse linear systems and observe that the convergence of the solution is essentially unaffected by the use of adaptive precision.

Key words. mixed precision, adaptive precision, multiple precision, matrix–vector product, sparse matrix, SpMV, numerical linear algebra, rounding error analysis, floating-point arithmetic, GMRES, Krylov solver, iterative solver, linear system

AMS subject classifications. 65G50, 65F05, 65F08, 65F50, 65F10

1. Introduction.

Motivated by the growing availability of lower precision arithmetics, mixed precision algorithms are being developed for a wide range of numerical computations [15]. One subclass of mixed precision algorithms that has recently and increasingly proven successful is what we call adaptive precision algorithms. These algorithms are based on the idea of adapting the precision to the data involved in the computation, by selecting a level of precision proportional to the importance of the data, where the definition of "importance" is application dependent. For example, Anzt et al. [4], [10] have proposed an adaptive precision block Jacobi preconditioner in which the precision of each block is chosen based on its condition number. Another example is the mixed precision low-rank compression proposed by Amestoy et al. [3], which partitions a low-rank matrix into several low-rank components of decreasing norm and stores each of them in a correspondingly decreasing precision. Ahmad et al. [1] develop a sparse matrix–vector product algorithm in which elements in the range $[-1, 1]$ are switched to single precision while the other elements are kept in double precision. Diffenderfer et al. [9] propose a "quantized" dot product algorithm that adapts the precision of each vector element based on its exponent. For a unified presentation of these adaptive precision algorithms, see [15, sect. 14].

In this article, we propose an adaptive precision algorithm at the element level for matrix–vector products. Specifically, our matrix–vector product algorithm partitions the elements into several buckets and uses a different precision for each bucket. We perform a rounding error analysis of this algorithm that reveals how the precisions should be chosen: we prove that it suffices to take the precisions to be proportional to the magnitude of the elements, that is, elements of large magnitude should be kept in high precision, but elements of smaller magnitude can be switched to correspondingly lower precisions. Intuitively, this discovery can be explained by the fact that the least significant bits of the smaller elements end up being lost when they are summed to the larger elements: hence, we might as well avoid computing those bits to begin with.

Based on this analysis, we develop an adaptive precision sparse matrix–vector product and evaluate experimentally its performance and accuracy on a range of real-life large sparse matrices. We show that the storage and hence the data movement costs of the product can be significantly reduced for many matrices, while preserving a user-prescribed accuracy target. We develop an implementation for CPUs that uses double and single precision arithmetic as well as dropping,

*Version of September 15, 2022

†Sorbonne Université, CNRS, LIP6, Paris, F-75005, France (stef.graillat@lip6.fr,theo.mary@lip6.fr)

‡Sorbonne Université, CNRS, LIP6, Paris, F-75005 and Université Paris-Panthéon-Assas, Paris, F-75006, France (fabienne.jezequel@lip6.fr)

§Sorbonne Université, CNRS, LIP6 and Université Paris-Saclay, Paris, F-75005, France (romeo.molina@lip6.fr)

and obtain speedups of up to an order of magnitude on a multicore computer. We then apply our algorithm to the solution of sparse linear systems by plugging it into a GMRES solver with iterative refinement. Our experiments demonstrate that the convergence of the solution is essentially unaffected by the use of adaptive precision.

The rest of this paper is organized as follows. We begin by recalling the error analysis of the standard matrix–vector product in uniform precision in section 2. Then, we propose in section 3 an adaptive precision matrix–vector product algorithm, carry out its error analysis, and investigate experimentally both its accuracy and performance. In section 4 we apply this algorithm to the solution of linear systems with GMRES. Finally, we provide our concluding remarks in section 5.

2. Uniform precision matrix–vector product. Before proposing an adaptive precision matrix–vector product, let us recall the error analysis of the uniform precision case, where the same precision is used across all operations.

Throughout the article we use the standard model of floating-point arithmetic [12, sec. 2.2]

$$\mathrm{fl}(a \operatorname{op} b) = (a \operatorname{op} b)(1+\delta), \quad |\delta| \le u, \quad \operatorname{op} \in \{+,-,\times,/\}, \tag{2.1}$$

where u is the unit roundoff of the precision used and fl represents the computed results in floating point.

Let $y_i = \sum_{j\in J_i} a_{ij}x_j$ be the inner product between the ith row of A and x, where J_i is the set of the column indices of the nonzero elements in row i of A. In uniform precision u, the computed $\widehat{y}_i$ satisfies

$$|\widehat{y}_i - y_i| \le \#J_i u \sum_{j\in J_i} |a_{ij}x_j|, \tag{2.2}$$

where $\#J_i$ denotes the cardinality of J_i. Note that here, and throughout the article, we have used the analysis of inner products of Jeannerod and Rump [16] to obtain more refined bounds, where constants of the form $\gamma_n = nu/(1-nu)$ can be replaced simply by nu. The analysis of [16] assumes the use of rounding to nearest, but it was later shown in [17, Corollary 3.3] that these refined bounds also hold for directed roundings by replacing u with $2u$. We also note that constants n could be further reduced to $\sqrt{n}$ to obtain probabilistic bounds that hold with high probability [13, 14, 7]. In this article the size of the constants is not the main focus (as they are typically small for sparse matrices), and so we use the more general worst-case error bounds.

Algorithm 2.1 Uniform precision matrix–vector product.

1: **Input:** $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$. J_i is the set of column indices of the nonzeros in row i of A.
2: **Output:** $y = Ax$
3: **for** $i = 1\colon m$ **do**
4: $\quad y_i = 0$
5: $\quad$ **for** $j \in J_i$ **do**
6: $\quad\quad y_i \leftarrow y_i + a_{ij}x_j$
7: $\quad$ **end for**
8: **end for**

As a consequence of the Oettli–Prager [12, Thm. 7.3], [21] and Rigal–Gaches [12, Thm. 7.1], [24] theorems, we have the following formulas for the componentwise backward error

$$\varepsilon_{\mathrm{cw}} = \min\{\varepsilon : \widehat{y} = (A+\Delta A)x,\ |\Delta A| \le \varepsilon |A|\} = \max_i \left[\frac{|\widehat{y}_i - y_i|}{\sum_{j\in J_i}|a_{ij}x_j|}\right] \tag{2.3}$$

and for the normwise backward error

$$\varepsilon_{\mathrm{nw}} = \min\{\varepsilon : \widehat{y} = (A+\Delta A)x,\ \|\Delta A\| \le \varepsilon \|A\|\} = \frac{\|\widehat{y}-y\|}{\|A\|\|x\|}, \tag{2.4}$$

respectively. Throughout this article, the unsubscripted norm $\|\cdot\|$ denotes the infinity norm

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|.$$

Note that the componentwise error is always larger than the normwise one, since we have

$$\varepsilon_{\mathrm{nw}} = \frac{\|\widehat{y}-y\|}{\|A\|\|x\|} \le \frac{\|\widehat{y}-y\|}{\||A||x|\|} = \frac{\max_i |\widehat{y}-y|_i}{\max_i (|A||x|)_i} \le \max_i \frac{|\widehat{y}-y|_i}{(|A||x|)_i} = \varepsilon_{\mathrm{cw}}. \tag{2.5}$$

Moreover, using (2.2), we obtain the bound

$$\varepsilon_{\mathrm{nw}} \le \varepsilon_{\mathrm{cw}} \le pu, \tag{2.6}$$

where $p = \max_i \#J_i$ is the maximum number of nonzero elements per row of A.

3. Adaptive precision matrix–vector product. In this section we propose an adaptive precision matrix–vector product algorithm. We begin, in section 3.1, by performing the error analysis of a general mixed precision matrix–vector product that partitions the nonzero elements of the matrix into buckets and computes the partial inner products associated with each bucket in a different precision. Our analysis shows how to build these buckets so as to minimize the precisions used while achieving a prescribed backward error. In section 3.2 we then experimentally assess the performance of this algorithm on a wide range of real-life matrices.

3.1. Error analysis. In this section we analyze Algorithm 3.1 which computes a mixed precision matrix–vector product $y = Ax$ using q precisions $u_1 < u_2 < \cdots < u_q$. Each row i of A is partitioned into q buckets $B_{ik} \subset [\![1, n]\!]$, $k = 1\colon q$, and the inner product $y_i^{(k)} = \sum_{j\in B_{ik}} a_{ij}x_j$ associated with bucket B_{ik} is computed in precision u_k. All the partial inner products are then summed in precision u_1.

For Algorithm 3.1 to be well defined, we require that the B_{ik} form a partition of J_i (the nonzero elements in row i of A), that is, that they are disjoint and that their union is equal to J_i.

Algorithm 3.1 Adaptive precision matrix–vector product in q precisions $u_1 < \cdots < u_q$.

Input: $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, a partitioning of A into buckets B_{ik}
Output: $y = Ax$
for $i = 1\colon m$ **do**
  for $k = 1\colon q$ **do**
    $y_i^{(k)} = 0$
    for $j \in B_{ik}$ **do**
      $y_i^{(k)} \leftarrow y_i^{(k)} + a_{ij}x_j$ in precision u_k
    end for
  end for
  $y_i = \sum_{k=1}^q y_i^{(k)}$ in precision u_1
end for

According to (2.2) the computed partial inner product $\widehat{y}_i^{(k)}$ satisfies

$$|\widehat{y}_i^{(k)} - y_i^{(k)}| \le p_{ik}u_k(1+u_k)^2 \sum_{j\in B_{ik}} |a_{ij}x_j|, \tag{3.1}$$

where $p_{ik} = \#B_{ik}$ and where the $(1+u_k)^2$ term accounts for the need to first convert both a_{ij} and x_j to precision u_k. Then, defining $\overline{y}_i = \sum_{k=1}^q \widehat{y}_i^{(k)}$ as the exact sum of the $\widehat{y}_i^{(k)}$, and as $y_i = \sum_{k=1}^q y_i^{(k)}$, we have

$$|\overline{y}_i - y_i| \le \sum_{k=1}^q \left[p_{ik}u_k(1+u_k)^2 \sum_{j\in B_{ik}} |a_{ij}x_j| \right], \tag{3.2}$$

and the computed $\widehat{y}_i$ satisfies

$$|\widehat{y}_i - \overline{y}_i| \le (q-1)u_1 \sum_{k=1}^{q} |\widehat{y}_i^{(k)}| \tag{3.3}$$

$$\le (q-1)u_1 \sum_{k=1}^{q} \left[\left(1 + p_{ik}u_k(1+u_k)^2\right) \sum_{j \in B_{ik}} |a_{ij}x_j| \right], \tag{3.4}$$

where the conversion of $\widehat{y}_i^{(k)}$ back to precision u_1 does not introduce any error since $u_1 \le u_k$ for all k. Using the fact that the B_{ik} form a partition of J_i, we have that

$$\sum_{k=1}^{q} \sum_{j \in B_{ik}} |a_{ij}x_j| = \sum_{j \in J_i} |a_{ij}x_j|$$

and we therefore obtain

$$|\widehat{y}_i - y_i| \le |\widehat{y}_i - \overline{y}_i| + |\overline{y}_i - y_i| \tag{3.5}$$

$$\le (q-1)u_1 \sum_{j \in J_i} |a_{ij}x_j| + (1 + (q-1)u_1) \sum_{k=1}^{q} \left[p_{ik}u_k(1+u_k)^2 \sum_{j \in B_{ik}} |a_{ij}x_j| \right]. \tag{3.6}$$

Dividing both sides by $\sum_{j \in J_i} |a_{ij}x_j|$, we obtain the componentwise backward error bound

$$\varepsilon_{\mathrm{cw}} \le (q-1)u_1 + (1 + (q-1)u_1) \max_i \left[\sum_{k=1}^{q} p_{ik}u_k(1+u_k)^2 \alpha_{ik} \right], \tag{3.7}$$

which shows that the ratios

$$\alpha_{ik} = \frac{\sum_{j \in B_{ik}} |a_{ij}x_j|}{\sum_{j \in J_i} |a_{ij}x_j|} \tag{3.8}$$

play a fundamental role in controlling the size of the backward error.

Now we want to determine how to build the buckets B_{ik} such that the backward error is at most in $O(\epsilon)$, where $\epsilon \ge u_1$ is a user-prescribed target accuracy. The analysis above shows that to do so, we need to control the ratios α_{ik}, which are essentially a measure of how large the elements in bucket B_{ik} are with respect to all the elements in J_i. Thus, the analysis tells us that elements smaller in magnitude can be placed in lower precision buckets. Specifically, writing a_i the ith row of A so that $\sum_{j \in J_i} |a_{ij}x_j| = |a_i|^T|x|$, let us define the intervals

$$P_{ik} = \begin{cases} \left(\epsilon|a_i|^T|x|/u_2,\ +\infty\right) & \text{for } k = 1, \\ \left(\epsilon|a_i|^T|x|/u_{k+1},\ \epsilon|a_i|^T|x|/u_k\right] & \text{for } k = 2\colon q-1, \\ \left[0,\ \epsilon|a_i|^T|x|/u_q\right] & \text{for } k = q, \end{cases} \tag{3.9}$$

which form a partition of $[0, +\infty)$, and let us define the buckets B_{ik} as the column indices of the nonzero elements of A such that $|a_{ij}x_j|$ belongs to the corresponding interval P_{ik}:

$$B_{ik} = \{ j \in J_i : |a_{ij}x_j| \in P_{ik} \}. \tag{3.10}$$

The definition of the P_{ik} intervals is illustrated with four precisions in Figure 3.1. This construction yields $\alpha_{ik} \le p_{ik}\epsilon/u_k$; note that this holds for $k = 1$ since $\epsilon \ge u_1$. Therefore, by (3.7),

$$\varepsilon_{\mathrm{cw}} \le (q-1)u_1 + c\epsilon = O(\epsilon), \tag{3.11}$$

with

$$c = (1 + (q-1)u_1) \max_i \sum_{k=1}^{q} p_{ik}^2(1+u_k)^2. \tag{3.12}$$

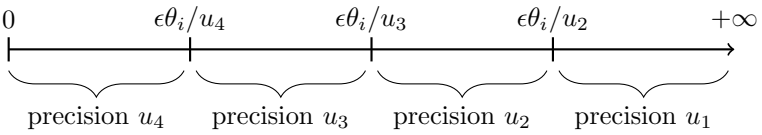


Fig. 3.1: Illustration of the bucket construction with four precisions $u_1 < u_2 < u_3 < u_4$. The real line $[0, +\infty)$ is partitioned into intervals P_{ik} defined by (3.9) (componentwise criteria, $\theta_i = |a_i|^T|x|$) or (3.17) (normwise criteria, $\theta_i = \|A\|$).

We note that we have not taken into account any rounding error occuring in the computation of the intervals P_{ik}, which we assume to be evaluated in sufficiently high precision to be considered exact.

Since, by (2.5), $\varepsilon_{\mathrm{nw}} \le \varepsilon_{\mathrm{cw}}$, this bucket construction also yields a normwise backward error in $O(\epsilon)$. However, if we only need to bound the normwise backward error, and can afford a potentially large componentwise error, we can improve the use of low precisions by modifying the buckets as follows. Taking norms in (3.6) shows that

$$\varepsilon_{\mathrm{nw}} \le (q-1)u_1 + (1+(q-1)u_1)\max_i \Big[\sum_{k=1}^{q} p_{ik}u_k(1+u_k)^2\beta_{ik}\Big], \tag{3.13}$$

where it is now the ratios

$$\beta_{ik} = \frac{\sum_{j\in B_{ik}}|a_{ij}x_j|}{\|A\|\|x\|} \tag{3.14}$$

that play a role in controlling the size of the normwise backward error. Importantly, unlike the ratios α_{ik} in (3.8), the ratios β_{ik} can be bounded above independently of x:

$$\beta_{ik} \le \frac{\sum_{j\in B_{ik}}|a_{ij}|}{\|A\|}. \tag{3.15}$$

As a result, we can redefine the buckets as

$$B_{ik} = \{j \in J_i : |a_{ij}| \in P_{ik}\}. \tag{3.16}$$

with the intervals P_{ik} as in (3.9) with $|a_i|^T|x|$ replaced with $\|A\|$:

$$P_{ik} = \begin{cases} \big(\epsilon\|A\|/u_2,\ +\infty\big) & \text{for } k = 1, \\ \big(\epsilon\|A\|/u_{k+1},\ \epsilon\|A\|/u_k\big] & \text{for } k = 2\colon q-1, \\ \big[0,\ \epsilon\|A\|/u_q\big] & \text{for } k = q. \end{cases} \tag{3.17}$$

This is sufficient to ensure that $\beta_{ik} \le p_{ik}\epsilon/u_k$ and thus that $\varepsilon_{\mathrm{nw}} = O(\epsilon)$. However, in this case we can no longer guarantee a small $\varepsilon_{\mathrm{cw}}$, since the ratios $\alpha_{ik}/\beta_{ik} = \|A\|\|x\|/|a_i|^T|x|$ can be arbitrarily large for some rows i.

We summarize the main conclusions of our analysis in the next theorem.

THEOREM 3.1. *Let $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$ and let $y = Ax$ be computed with Algorithm 3.1. If the bucket partitioning is defined by (3.9)–(3.10), then we have*

$$\varepsilon_{\mathrm{nw}} \le \varepsilon_{\mathrm{cw}} \le (q-1)u_1 + c\epsilon,$$

where the expression of c is given by (3.12). If instead it is defined by (3.16)–(3.17), then we only have

$$\varepsilon_{\mathrm{nw}} \le (q-1)u_1 + c\epsilon.$$

REMARK 3.1. *For sparse matrices, since the performance of SpMV is memory bound, in principle we could only store the elements of A in lower precisions and keep the floating-point*

operations in precision u_1 in order to avoid error accumulation. The error analysis above can be easily adapted to this scenario by replacing (3.1) with

$$|\widehat{y}_i^{(k)} - y_i^{(k)}| \leq \left(p_{ik}u_1(1+u_k) + u_k\right) \sum_{j \in B_{ik}} |a_{ij}x_j|, \tag{3.18}$$

which roughly reduces the p_{ik}^2 term in (3.12) to p_{ik}.

REMARK 3.2. *Our analysis allows for the case where some elements of A are simply dropped. Indeed, this can be modeled as using a "precision" $u_q = 1$, since replacing an element by zero introduces a relative perturbation equal to 1. Thus, taking $u_q = 1$ in (3.9) or (3.17) shows that elements of magnitude smaller than $\epsilon|a_i|^T|x|$ or $\epsilon\|A\|$ can be dropped while preserving a componentwise or normwise backward error of order ϵ, respectively.*

REMARK 3.3. *Our analysis can be trivially specialized to adaptive precision inner products, for which A is a row vector, and to adaptive precision summation, for which $A = e = [1, \ldots, 1]$.*

3.1.1. A more practical componentwise bucket criteria. The approach presented above presents a practical limitation: to guarantee *componentwise* backward stability, the adaptive precision representation of matrix A must depend on the vector x we want to multiply it with, as shown by (3.9)–(3.10). Unfortunately, taking the values of x into account is unrealistic, since it would require to change the representation of A every time we want to compute its product with a different vector. A more practical scenario is to compute an adaptive precision representation of A independent of x and use it to accelerate many SpMVs with different vectors. The bucket construction defined by (3.16)–(3.17) satisfies this practical constraint, but can only guarantee normwise stability.

This motivates us to propose a bucket construction

$$B_{ik} = \{j \in J_i : |a_{ij}| \in P_{ik}\} \tag{3.19}$$

with the definition of the intervals P_{ik} modified as follows:

$$P_{ik} = \begin{cases} \left(\epsilon|a_i|^T e/u_2,\ +\infty\right) & \text{for } k = 1, \\ \left(\epsilon|a_i|^T e/u_{k+1},\ \epsilon|a_i|^T e/u_k\right] & \text{for } k = 2\colon q-1, \\ \left[0,\ \epsilon|a_i|^T e/u_q\right] & \text{for } k = q, \end{cases} \tag{3.20}$$

where $e = [1, \ldots, 1]^T$, so that $|a_i|^T e = \sum_{j \in J_i} |a_{ij}|$. This modified definition essentially amounts to drop x in the componentwise bucket construction (3.9)–(3.10). With this bucket construction, we can bound the ratios α_{ik} (3.8)

$$\alpha_{ik} \leq \frac{p_{ik}\epsilon}{u_k} \frac{|a_i|^T e}{|a_i|^T|x|} \|x\|, \tag{3.21}$$

whereas with the normwise bucket construction (3.16)–(3.17), the best bound on α_{ik} we can get is

$$\alpha_{ik} \leq \frac{p_{ik}\epsilon}{u_k} \frac{\|A\|}{|a_i|^T|x|} \|x\|. \tag{3.22}$$

Clearly, the right-hand side of (3.22) can be larger than that of (3.21), especially for badly scaled matrices with rows such that $\|a_i\| \ll \|A\|$. Therefore, we can expect that at least in some cases, construction (3.19)–(3.20) can lead to much smaller ε_{cw} than construction (3.16)–(3.17). It is important to note that, unfortunately, construction (3.19)–(3.20) cannot always guarantee a small ε_{cw}, since the ratio $|a_i|^T e/|a_i|^T|x|$ can be arbitrarily large for an unlucky choice of vector x.

3.2. Numerical experiments. We now evaluate the performance of our adaptive precision matrix–vector product, Algorithm 3.1, by applying it to a range of real-life large sparse matrices.

We have developed a Fortran code that implements Algorithm 3.1 and made it publicly available[1]. Our code uses up to seven different precisions: the IEEE binary64 and binary32

[1] https://gitlab.com/romeomolina/adaptive-spmv

formats (hereinafter denoted as fp64 and fp32), the bfloat16 format, and four custom formats using 56, 48, 40, and 24 bits, which we will refer to as "fp$x$", with x the number of bits. The fp56, fp48, and fp40 formats use 11 bits for the exponent and thus have unit roundoffs 2^{-45}, 2^{-37}, and 2^{-29}, whereas the fp24 format uses 8 bits for the exponent, which corresponds to a unit roundoff 2^{-16}. This choice of formats aims at spanning as uniformly as possible the range of precisions used. In principle, we could have used many more precision formats by adapting the precision bit by bit, but focusing on formats that use multiples of 8 bits simplifies the implementation of the cast operations. We also do not experiment with formats using a reduced number of bits for the exponent, such as IEEE binary16. In addition to these seven precision formats, we also drop the matrix elements that are sufficiently small, as explained in Remark 3.2. The list of precision formats is summarized in Table 3.1.

Table 3.1: List of precision formats used in our experiments.

	Numbers of bits			Range	Unit roundoff
	Sign	Exponent	Significand		
bfloat16	1	8	7	$10^{\pm 38}$	$2^{-8} \approx 4 \times 10^{-3}$
fp24	1	8	15	$10^{\pm 38}$	$2^{-16} \approx 2 \times 10^{-5}$
fp32	1	8	23	$10^{\pm 38}$	$2^{-24} \approx 6 \times 10^{-8}$
fp40	1	11	28	$10^{\pm 308}$	$2^{-29} \approx 2 \times 10^{-9}$
fp48	1	11	36	$10^{\pm 308}$	$2^{-37} \approx 7 \times 10^{-12}$
fp56	1	11	44	$10^{\pm 308}$	$2^{-45} \approx 3 \times 10^{-14}$
fp64	1	11	52	$10^{\pm 308}$	$2^{-53} \approx 1 \times 10^{-16}$

For the cast from fp64 to fp32 we use the Fortran `REAL` function, whereas for casting to the other custom formats (including bfloat16, which our hardware does not support), we use our own implementation that consists in chopping the desired bits by using the `MVBITS` subroutine of the GNU Fortran compiler. Our environment only supports floating-point operations in fp64 or fp32. As a result, after casting the matrix elements to these custom precision formats, we must cast them back during the computation, either to fp32 (in the case of bfloat16 and fp24) or to fp64 (in the case of fp40, fp48, and fp56). As mentioned in Remark 3.1, performing the computations in a higher precision than the storage format only affects the constants in the error bounds.

The experiments were performed on an Intel Xeon X5690 processor at 3.47GHz using 24 threads. For the time measurements, we perform one hundred products and report the average timings. We use the CSR format to represent all matrices, and parallelize the loop on the row indices with OpenMP. For the adaptive precision case, we use a different CSR matrix for each precision. We do not include the time for reading the matrix from a file and putting it into CSR format. We also do not include the time for preprocessing the matrix into its adaptive precision representation (that is, for computing the bucket partitioning and creating the corresponding data structures). This preprocessing requires at most two passes over the nonzero elements of the matrix: one to compute the intervals P_{ik} (optional for the normwise criteria if $\|A\|$ is already known or can be cheaply estimated) and another to place the nonzeros into the corresponding bucket (CSR matrix). Therefore the cost of the preprocessing is negligible as long as we require several SpMVs (say, at least a dozen) with the same matrix, which is typically the case in iterative solvers.

The matrices used in these experiments come from the SuiteSparse collection [8] and from our industrial partners (see Table 3.2). The thmgaz matrix corresponds to a coupled thermal, hydrological, and mechanical problem. The series of matrices Aghora_DGO{2,3,4} arise from the resolution of adjoint RANS equations in the context of high-fidelity simulations of turbulent compressible flows in aerodynamics. The spatial discretization of these equations relies on a high-order discontinuous Galerkin (DG) method with third, fourth, and fifth order accurate schemes. The test case corresponds to a subsonic laminar flow over a NACA0012 airfoil. Jacobian matrices have been built with the ONERA Aghora DG solver [23] and are real, nonsymmetric, not positive

definite, with a blockwise structure and a symmetric pattern.

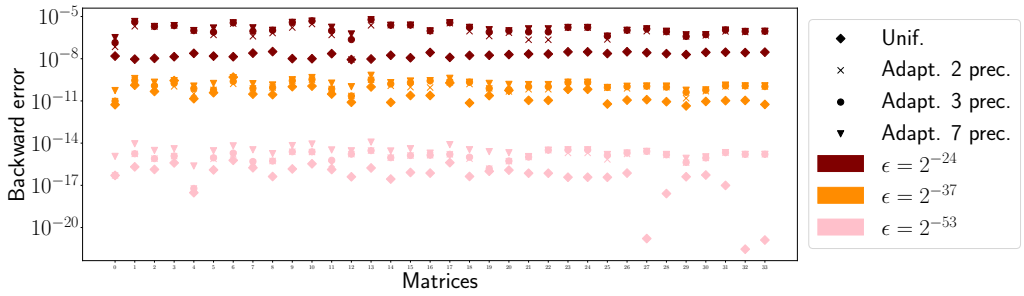
Clearly, by its very design, the potential of the adaptive precision algorithm completely depends on the matrix values: there must be sufficient variations in their magnitudes. For example, SuiteSparse has several binary matrices (with only zeros and ones) that present no potential at all. In our experiments, we have selected a range of matrices that present at least some potential, listed in Table 3.2. As for the vector x, we set it to $e = [1, \ldots, 1]^T$ throughout the experiments.

Table 3.2: List of matrices used in our experiments.

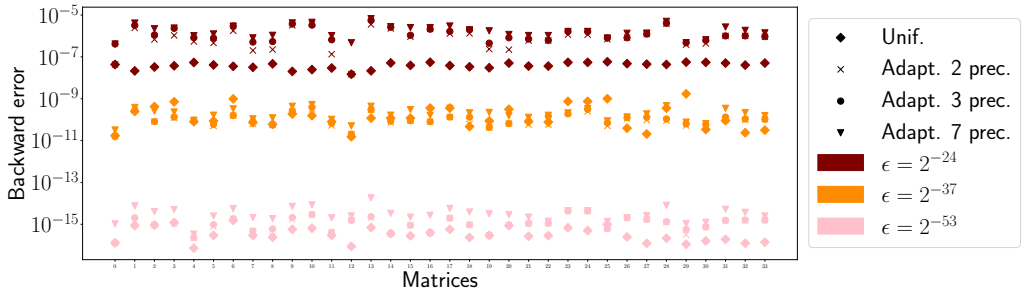
Number	Matrix	n	nnz
0	Transport	$1.6e+06$	$2.4e+07$
1	vas_stokes_4M	$4.4e+06$	$1.3e+08$
2	Aghora_DGO3.mtx	$1.5e+05$	$1.8e+07$
3	Aghora_DGO4.mtx	$2.6e+05$	$5.1e+07$
4	Emilia_923	$9.2e+05$	$2.1e+07$
5	Hook_1498	$1.5e+06$	$3.1e+07$
6	Aghora_DGO5.mtx	$3.8e+05$	$1.1e+08$
7	ML_Geer	$1.5e+06$	$1.1e+08$
8	Geo_1438	$1.4e+06$	$3.2e+07$
9	vas_stokes_1M	$1.1e+06$	$3.5e+07$
10	stokes	$1.1e+07$	$3.5e+08$
11	ML_Laplace	$3.8e+05$	$2.8e+07$
12	ss	$1.7e+06$	$3.5e+07$
13	vas_stokes_2M	$2.1e+06$	$6.5e+07$
14	Fault_639	$6.4e+05$	$1.5e+07$
15	Queen_4147	$4.1e+06$	$1.7e+08$
16	Bump_2911	$2.9e+06$	$6.5e+07$
17	thmgaz	$5.0e+06$	$4.7e+08$
18	PFlow_742	$7.4e+05$	$1.9e+07$
19	Flan_1565	$1.6e+06$	$5.9e+07$
20	CoupCons3D	$4.2e+05$	$2.2e+07$
21	Cube_Coup_dt6	$2.2e+06$	$6.5e+07$
22	Cube_Coup_dt0	$2.2e+06$	$6.5e+07$
23	Long_Coup_dt6	$1.5e+06$	$4.4e+07$
24	Long_Coup_dt0	$1.5e+06$	$4.4e+07$
25	StocF-1465	$1.5e+06$	$1.1e+07$
26	nv2	$1.5e+06$	$5.3e+07$
27	test1	$3.9e+05$	$1.3e+07$
28	radiation	$2.2e+05$	$7.6e+06$
29	power9	$1.6e+05$	$2.5e+06$
30	imagesensor	$1.2e+05$	$1.9e+06$
31	dgreen	$1.2e+06$	$3.8e+07$
32	mosfet2	$4.7e+04$	$1.5e+06$
33	nv1	$7.5e+04$	$2.4e+06$

We begin in Figure 3.2 by evaluating the accuracy of our adaptive precision algorithm to confirm that we are able to control the backward error, which, according to Theorem 3.1, should remain of order ϵ. We check this both for the normwise and componentwise backward errors by plotting, in Figure 3.2a, the normwise backward error for the algorithm with the normwise bucket criteria (3.16)–(3.17), and, in Figure 3.2b, the componentwise backward error for the algorithm with the componentwise bucket criteria (3.9)–(3.10). We use three different target accuracies, that is, two values of ϵ, 2^{-53} and 2^{-24}, which correspond to the unit roundoffs of fp64 and fp32

respectively and an additional intermediate accuracy $\epsilon = 2^{-37}$, and compare its backward error to the one obtained by the uniform precision algorithm in the corresponding precision ($\epsilon = 2^{-24}$, $\epsilon = 2^{-37}$, $\epsilon = 2^{-53}$). Moreover, we also investigate how the backward error is affected if, instead of using all 7 precision formats, we only use 2 (fp64 and fp32) or 3 (fp64, fp32, and bfloat16). As expected, the measured errors remain close to the target accuracy, for all targets ϵ, and for any configuration of precision formats. Using more precision formats slightly increases the error, which is explained by the analysis, since the constant c in (3.12) increases with q.



(a) Normwise backward error (2.4) (the adaptive precision algorithm uses the normwise bucket criteria (3.16)–(3.17)).



(b) Componentwise backward error (2.3) (the adaptive precision algorithm uses the componentwise bucket criteria (3.9)–(3.10)).

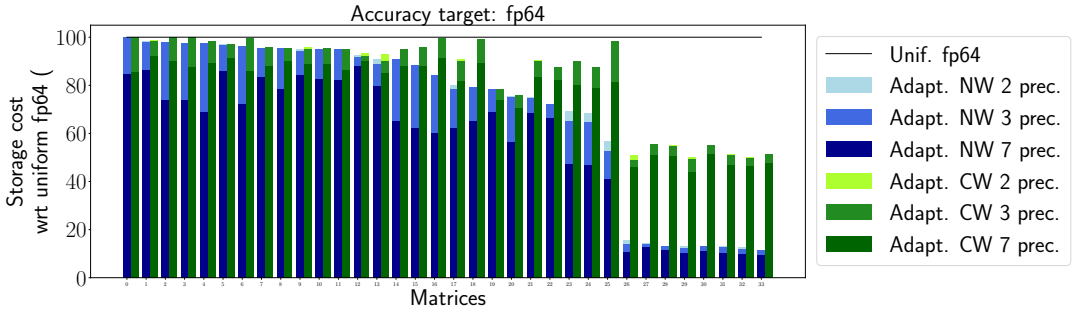
Fig. 3.2: Backward error for the adaptive precision Algorithm 3.1 with different target accuracies ϵ and different number of precisions used, compared with the uniform precision Algorithm 2.1 in the corresponding precision ($\epsilon = 2^{-24}$, $\epsilon = 2^{-37}$, $\epsilon = 2^{-53}$).

Next, we evaluate the performance gains achieved by the adaptive precision algorithm. We first measure the storage gains, that is, the number of bytes necessary to store the matrix in adaptive precision. The storage cost is a relevant metric because it drives the data movement costs of the SpMV, which is a memory-bound algorithm.

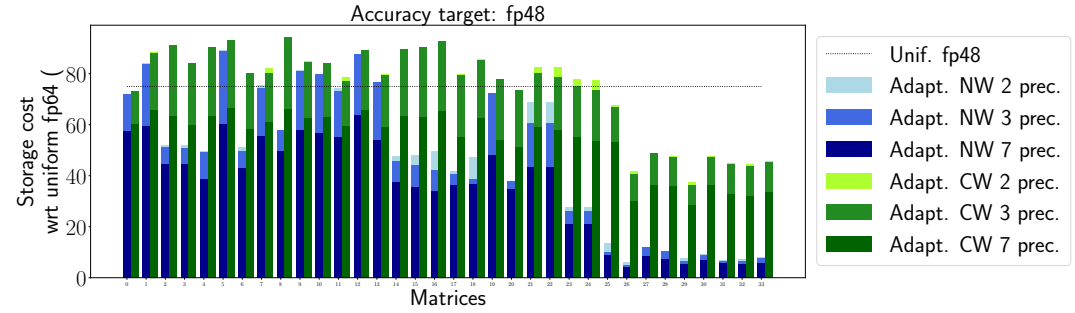
Figure 3.3 plots the storage cost of the adaptive precision algorithm as a percentage of the uniform precision fp64 algorithm. As for Figure 3.2, several configurations of the adaptive precision algorithm are tested, depending on the accuracy target ($\epsilon = 2^{-24}$, $\epsilon = 2^{-37}$, $\epsilon = 2^{-53}$), the number of precisions used (2, 3, or 7, with dropping being used in all cases), and on whether the buckets are built with the componentwise criteria (3.9)–(3.10) or the normwise one (3.16)–(3.17). Clearly, the more precision formats are used, the larger are the gains, since we can better adapt the choice of precisions to each element. In some cases, the use of more than two precisions appears to be critical: for example, the storage cost for matrices 14 or 20 with an fp32 accuracy target (Figure 3.3c) is nearly divided by two when adding bfloat16 (3 precisions instead of 2). Moreover, as expected, the storage gains are always larger with the normwise criteria (blue bars), which offers more room to the use of lower precisions than the componentwise one (green bars). Finally, it is also worth noting that the relative storage gains also become larger as the accuracy target is lowered, even when compared with the uniform precision algorithm in the corresponding

precision. That is, while lowering the accuracy target from fp64 (Figure 3.3a) to fp32 (Figure 3.3c) reduces the storage cost of the uniform precision algorithm by a factor two, it can reduce the cost of the adaptive precision algorithm by a much larger factor. This is for example the case for matrix 16, for which the adaptive precision algorithm (with 7 precisions and a normwise criteria) achieves a cost of about 60% of the uniform fp64 cost for an fp64 target, to be compared with only about 5% of the uniform fp64 cost (and hence 10% of the uniform fp32 cost) for an fp32 target.

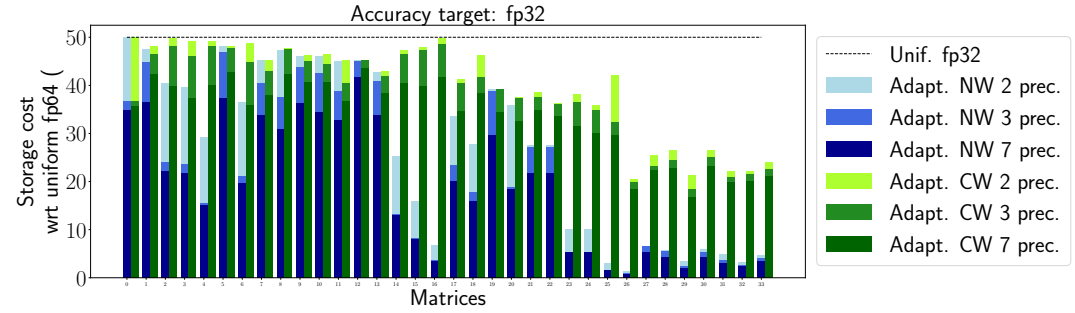
In any case, the storage gains are overall significant in all configurations and for several matrices, with reductions of up to a factor 36× in the best case.



(a) fp64 accuracy target



(b) fp48 accuracy target



(c) fp32 accuracy target

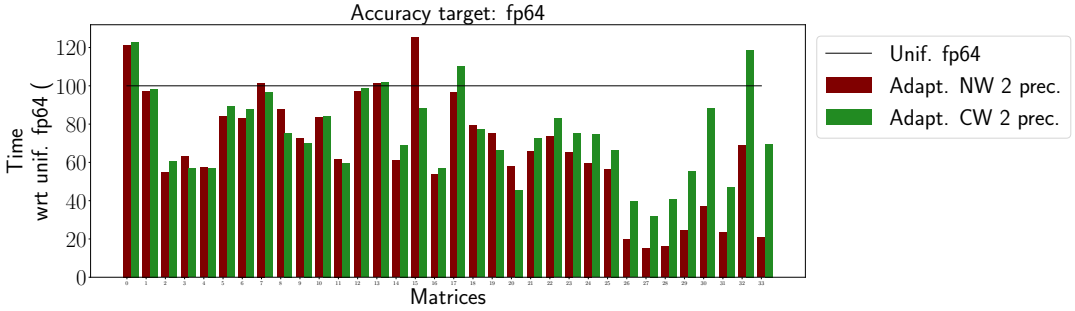
Fig. 3.3: Storage cost of the adaptive precision SpMV, as a percentage of the storage cost of the uniform precision fp64 SpMV, for three different accuracy targets. For each plot, we report the storage gains depending on which of the componentwise ("CW") or normwise ("NW") criteria is considered and on how many precision formats are used.

Finally, we measure the execution time of the algorithms. Since SpMV is memory bound, in principle we can hope the time gains to roughly follow the storage gains, even though the

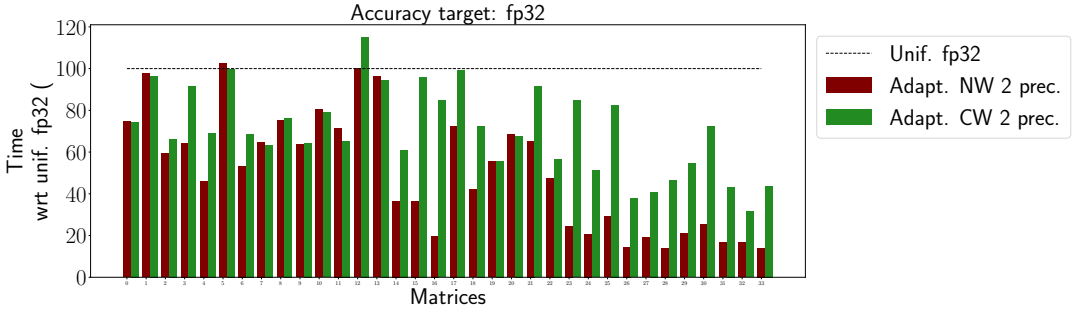
execution time depends on several other factors such as the overhead cost of the cast operations and the latency costs. In our experiments, we have found the time cost of the adaptive precision SpMV to roughly match its storage cost in the case where we only use precision formats that are natively supported in our environment, that is, the fp64 and fp32 formats (which corresponds to the two-precision version plus dropping). Unfortunately, we have found the use of other custom precision formats to lead to slowdowns due to a heavy performance penalty associated with our cast implementation, which is not optimized and designed to validate the numerical behavior of our approach, rather than to provide acceleration with custom precision. However, there has been work describing such efficient implementations, such as [20] or more recently the memory accessor of Grützmacher et al. [11], which suggests that the three- and seven-precision versions could meet their potential with a more optimized implementation. Moreover lower precision formats such as bfloat16 are increasingly supported in hardware. We however leave these ideas for future work and focus here on the results obtained with the two-precision version.

Figure 3.4 reports the execution time of the adaptive precision SpMV for $\epsilon = 2^{-24}$ and $\epsilon = 2^{-53}$ target accuracies, as a percentage of the execution time of the uniform precision SpMV in the corresponding precision (fp64 or fp32). The time cost of the algorithm follows a trend similar to the storage cost, with the gains being in general smaller but still significant, with speedups of up to 7× in the best case.

Interestingly, for some matrices, the time reduction is larger than the storage one, and this effect is not explained by measurement noise and can be consistently reproduced across several runs. A possible explanation is that the smaller storage cost of the matrix reduces the number of cache misses and hence benefits from the doubled effect of a lower volume of data movement and higher bandwidth.



(a) fp64 accuracy target



(b) fp32 accuracy target

Fig. 3.4: Execution time of the adaptive precision SpMV for $\epsilon = 2^{-24}$ and $\epsilon = 2^{-53}$ target accuracies, as a percentage of the execution time of the uniform precision SpMV in the corresponding precision. Both the normwise ("NW") and componentwise ("CW") criteria are reported.

Finally, we also report the execution time in the case of an $\epsilon = 2^{-37}$ accuracy target in

Figure 3.5. The figure also plots the time for the $\epsilon = 2^{-24}$ and $\epsilon = 2^{-53}$ targets (already presented in Figure 3.4) as a point of comparison. Figure 3.5 illustrates a valuable feature of our adaptive precision algorithm: it is able to achieve a flexible level of accuracy that does not necessarily correspond to any natively supported precision format, while only using such supported formats (here fp64 and fp32). This is because the accuracy of the adaptive precision algorithm is determined by ϵ, rather than directly by the unit roundoffs of the precision formats that are used.

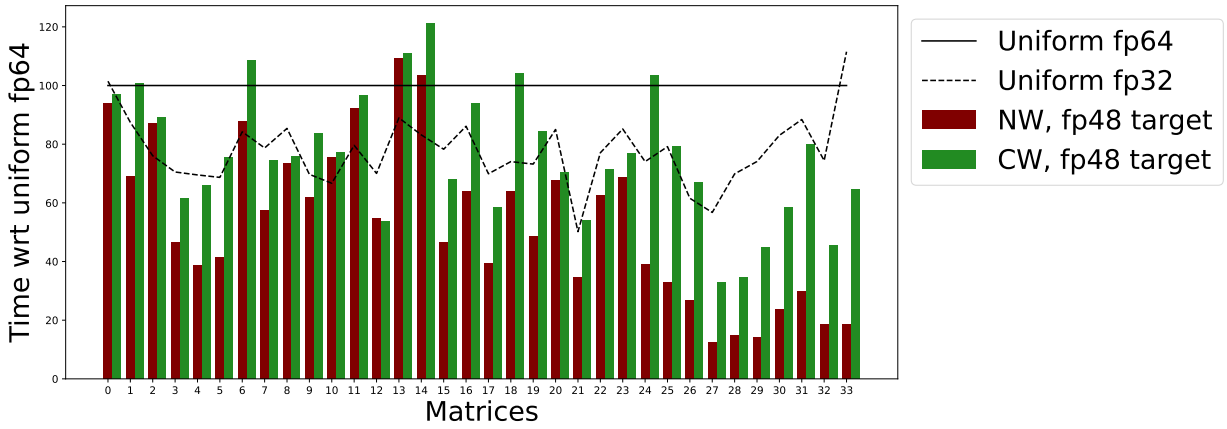


Fig. 3.5: Execution time of the adaptive precision SpMV for an $\epsilon = 2^{-37}$ target accuracy, as a percentage of the execution time of the uniform precision fp64 SpMV. Both the normwise ("NW") or componentwise ("CW") criteria are reported.

4. Application to iterative solvers. We now apply our adaptive precision SpMV (Algorithm 3.1) to the solution of linear systems $Ax = b$ by Krylov methods. Iterative solvers are indeed a natural application for our algorithm: since the matrix A remains fixed throughout the computation, we can partition it into adaptive precision form only once before using it throughout the iterations in potentially many matrix–vector products, as long as we rely on either the normwise bucket criteria (3.16)–(3.17) or the relaxed componentwise one (3.19)–(3.20).

4.1. Adaptive precision GMRES. We will focus our discussion and experiments on the GMRES algorithm [25] outlined in Algorithm 4.1, although we expect our conclusions to apply to other iterative solvers. The matrix–vector product with A (line 5) represents a significant fraction of the total cost of GMRES, so by accelerating it we can expect significant speedups on the whole solution, provided that the convergence of GMRES is preserved.

First, from a theoretical point of view, we can state that using an adaptive precision SpMV within a normwise backward stable GMRES solver, such as MGS-GMRES [22], will not endanger the normwise backward stability of the solution. Intuitively, this is not surprising since the adaptive precision SpMV is also backward stable, as we have shown in section 3.1. More formally, we can prove this by relying on the recent analysis of Amestoy et al. [2]. Indeed, [2, Thm. 3.1] proves, under mild assumptions, that if the products $y = Aq$ within MGS-GMRES are performed such that the computed $\widehat{y}$ satisfies

$$\widehat{y} = Aq + f, \quad \|f\| \leq \epsilon \|A\| \|q\|, \tag{4.1}$$

then the computed solution $\widehat{x}$ to $Ax = b$ satisfies a backward error in $O(\epsilon)$. We can therefore conclude from our Theorem 3.1 that setting the SpMV accuracy target to ϵ will also provide a backward error in $O(\epsilon)$ for the solution of $Ax = b$. Note that this theoretical discussion is limited to normwise stability, since GMRES is not known to be componentwise backward stable. Nevertheless, we will experiment with both the normwise and componentwise criteria for SpMV, because we have experimentally observed that using a componentwise stable SpMV can in some

Algorithm 4.1 GMRES.

Input: a matrix $A \in \mathbb{R}^{n\times n}$, a right-hand side $b \in \mathbb{R}^n$, and an initial solution $x_0 \in \mathbb{R}^n$.
Output: a solution $x_k \in \mathbb{R}^n$.

1: $r = b - Ax_0$
2: $\beta = \|r\|_2$
3: $q_1 = r/\beta$
4: **for** $k = 1, 2, \ldots$ **do**
5: $\quad y = Aq_k$
6: $\quad$ **for** $j = 1\colon k$ **do**
7: $\quad\quad h_{jk} = q_j^T y$
8: $\quad\quad y = y - h_{jk} q_j$
9: $\quad$ **end for**
10: $\quad h_{k+1,k} = \|y\|_2$
11: $\quad q_{k+1} = y/h_{k+1,k}$
12: $\quad$ Solve the least squares problem $\min_{c_k} \|Hc_k - \beta e_1\|_2$.
13: $\quad x_k = x_0 + Q_k c_k$
14: **end for**

cases improve the convergence behavior of GMRES compared with using an only normwise stable SpMV.

4.1.1. GMRES-based iterative refinement. In section 3.2, we have shown that the speedups achieved by the adaptive precision SpMV tend to be larger for lower accuracy targets. We now explain why, as a result of this property, the adaptive precision SpMV is particularly attractive in the context of GMRES-based iterative refinement (GMRES-IR) [15, sect. 8], [5, 6, 2, 18, 19]. GMRES-IR, described in Algorithm 4.2, takes the form of an inner–outer scheme, in which the solution x_i is iteratively refined (the outer loop) by solving a correction system $Ad_i = r_i$ using GMRES (Algorithm 4.1, the inner loop). Algorithm 4.2 is equivalent to restarted GMRES when the inner GMRES on line 3 is initialized with $d_0 = 0$.

Algorithm 4.2 GMRES-based iterative refinement.

Input: a matrix $A \in \mathbb{R}^{n\times n}$, a right-hand side $b \in \mathbb{R}^n$, and an initial solution $x_0 \in \mathbb{R}^n$.
Output: a solution $x_i \in \mathbb{R}^n$.

1: **for** $i = 1, 2, \ldots$ **do**
2: $\quad r_i = b - Ax_{i-1}$
3: $\quad$ Solve $Ad_i = r_i$ by GMRES (Algorithm 4.1) in lower precision.
4: $\quad x_i = x_{i-1} + d_i$
5: **end for**

Importantly, it is known that Algorithm 4.2 can converge to a high accuracy even when the inner GMRES is performed entirely in low precision [15, sect. 8], [2, 6]. In our adaptive precision context, we can therefore perform the outer loop SpMV (line 2 of Algorithm 4.2) in adaptive precision with a high accuracy target ϵ_{out}, and the inner loop SpMV (line 5 of Algorithm 4.1) in adaptive precision with a lower accuracy target $\epsilon_{\text{in}} \gg \epsilon_{\text{out}}$. Since the inner loop SpMV is called many more times than the outer loop one, we can expect the cost of the overall GMRES-IR solution to be determined by the cost of the low accuracy inner loop SpMV.

4.2. Experiments. We now assess experimentally how the use of adaptive precision SpMV affects the convergence of GMRES-IR. For the inner solver, we use our own implementation of MGS-GMRES, with a maximum number of inner iterations (restart size) fixed to 80. We incorporate a row scaling in GMRES-IR by solving $D^{-1}Ax = D^{-1}b$, with D a diagonal matrix whose coefficients are defined as $d_{ii} = \max_j |a_{ij}|$. This scaling also serves as a very simple Jacobi preconditioner; we leave the use of more complex preconditioners for future work.

The goal of this section is to compare the use of uniform and adaptive precision SpMV with

different parameters. For the outer loop, we use an fp64 accuracy target $\epsilon_{\text{out}} = 2^{-53}$, that is, the uniform precision GMRES-IR uses a uniform fp64 outer SpMV and the adaptive precision GMRES-IR uses an adaptive precision SpMV with target accuracy ϵ_{out} and componentwise criteria. We focus our experimental analysis on the inner loop SpMV parameters: ϵ_{in} and the choice between componentwise ("CW" hereinafter) or normwise ("NW") criteria.

We first illustrate different aspects of the behavior of adaptive precision GMRES-IR by using three examples, matrices ML_Laplace, CoupCons3D, and Geo_1438.

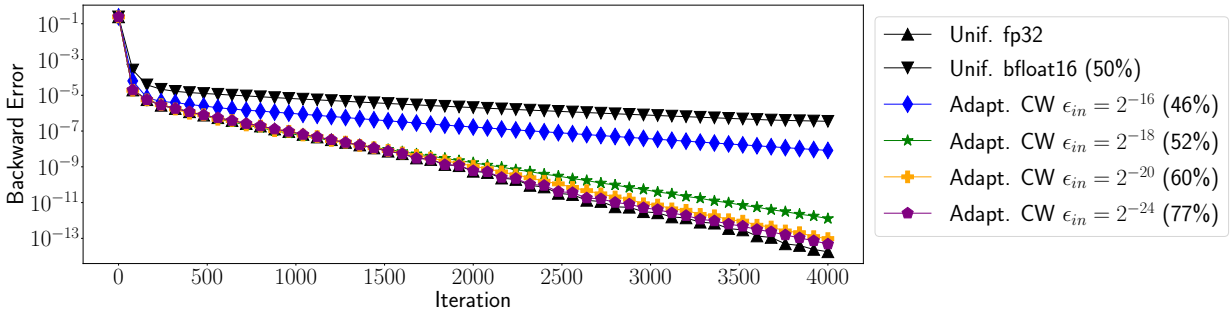


Fig. 4.1: Convergence of GMRES-IR for matrix ML_Laplace: illustration of the effect of the ϵ_{in} parameter.

Figure 4.1 plots the convergence of GMRES-IR for matrix ML_Laplace using either uniform or adaptive precision SpMV. Our reference is the fp32 uniform precision variant, which converges to nearly fp64 accuracy after 4000 iterations. We also test a bfloat16 uniform precision variant, whose convergence is much slower, achieving only a residual of about 10^{-6} after the same number of iterations. Finally, we test the adaptive precision SpMV variant with several values of ϵ_{in} and with CW criteria; for this matrix, the use of NW criteria significantly degrades the convergence (not shown). In the legend, we plot the adaptive precision SpMV cost as a percentage of the fp32 uniform precision one. The figure shows that ϵ_{in} has an effect on both the SpMV cost (and therefore, the cost per iteration of GMRES-IR) and the convergence speed (and therefore, the total number of iterations). For example, with $\epsilon_{\text{in}} = 2^{-24}$, we expect the adaptive precision SpMV to be about as accurate as the fp32 uniform precision one, and indeed, the adaptive precision GMRES-IR converges at roughly the same speed with only 77% of the SpMV cost. For $\epsilon_{\text{in}} = 2^{-16}$, the SpMV cost is only 46% of the fp32 uniform precision one, but GMRES-IR converges much slower. The optimal choice of ϵ_{in} lies in between these two values; for this matrix, $\epsilon_{\text{in}} = 2^{-20}$ for example is a good choice.

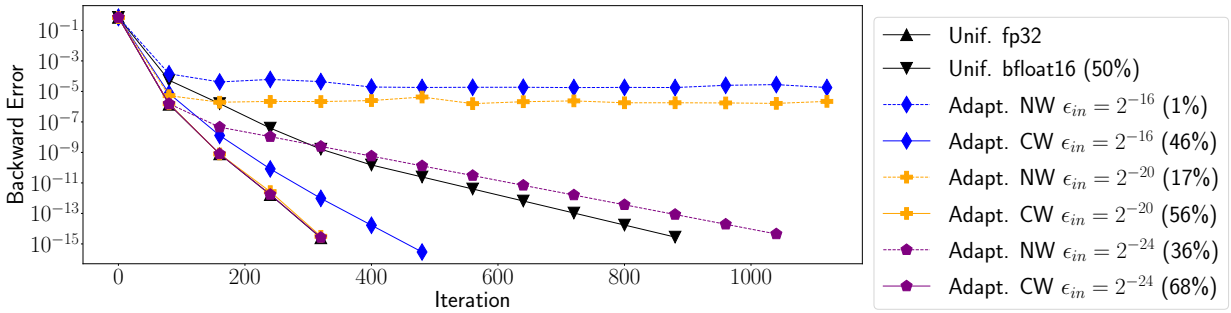


Fig. 4.2: Convergence of GMRES-IR for matrix CoupCons3D: illustration of the difference between CW and NW criteria.

Figure 4.2 plots the convergence of GMRES-IR for matrix CoupCons3D. Here, the adaptive precision variants can converge both for CW and NW criteria, and the figure illustrates the

different tradeoff that each option offers: for a fixed value of ϵ_{in}, NW variants achieve a lower cost but a slower convergence than CW ones. Therefore, the best choice of ϵ_{in} can be different for the NW and CW variants. In this example, $\epsilon_{\text{in}} = 2^{-24}$ leads to the best NW variant, which converges in 1040 iterations with an SpMV cost of 36% of the fp32 uniform one, whereas $\epsilon_{\text{in}} = 2^{-20}$ leads to the best CW variant, which converges in 320 iterations with a corresponding SpMV cost of 56%. Here, the CW variant therefore outperforms the NW one, but the figure illustrates that both options should be considered.

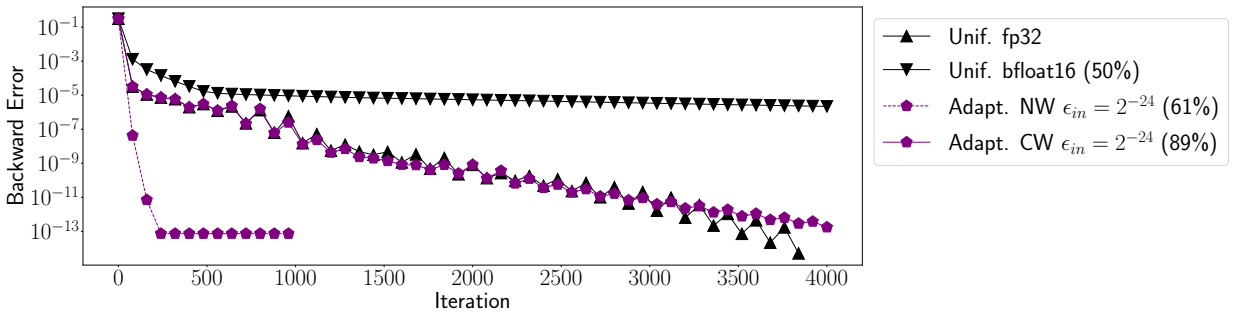


Fig. 4.3: Convergence of GMRES-IR for matrix Geo_1438: illustration of a surprising behavior of NW variants.

Finally, Figure 4.3 plots the convergence of GMRES-IR for matrix Geo_1438, which we use to illustrate a surprising behavior. As the figure shows, the NW adaptive precision variant can converge much faster than all the other variants, including the fp32 uniform precision one. Thus, the NW variant is much more efficient for this matrix since it requires both less iterations and a lower cost per iteration. This behavior can be consistently reproduced and occurs for several other matrices in our set. We do not have a completely satisfactory explanation; one possibility is that by aggressively dropping small coefficients from the matrix, the NW variant leads to a "nicer" matrix for which GMRES can converge quickly.

To conclude this experimental section, we provide in Table 5.1 additional results on a subset of matrices from Table 3.2. For each matrix, we compare the fp32 and bfloat16 uniform precision SpMV with the adaptive precision one, for various values of ϵ_{in} and both the CW and NW criteria. For each variant, we report the total number of iterations required by GMRES-IR, the final accuracy achieved after this number of iterations, the SpMV cost per iteration, and the global cost, which corresponds to the SpMV cost times the number of iterations. A "—" in place of the global cost indicates that the variant has not converged; we consider that a given variant has successfully converged if it achieves an accuracy within a factor 100 of the fp32 uniform precision variant accuracy. For most matrices, this corresponds to an accuracy equivalent to fp64 precision, but not for all, since there are some matrices for which even the fp32 uniform precision variant does not converge to this level of accuracy.

This range of experiments shows that significant cost reductions can be obtained by using an adaptive precision SpMV. In fact, for all matrices, there is at least one adaptive precision variant which outperforms both the fp32 and bfloat16 uniform precision variants. The best setting is matrix dependent; for matrices ML_Laplace and CoupCons3D, the best variant uses the CW criteria, whereas it uses the NW criteria for the other matrices.

5. Conclusions. We have presented a mixed precision algorithm to compute SpMVs and we have used it to accelerate the solution of sparse linear systems by iterative methods. Our algorithm is based on the idea of adapting the precision of each matrix element according to its magnitude: the elements are split into buckets that are summed in progressively lower precisions as their magnitudes decrease. We carried out a rounding error analysis of this algorithm, summarized in Theorem 3.1, which provides us with an explicit rule to build the buckets and to control its accuracy via a user-prescribed parameter ϵ.

Table 5.1: Results with GMRES-IR for various matrices and SpMV variants.

Matrix	Version	ϵ_{in}	Final accuracy	Nb of it.	Cost per it.	Global cost
ML_Laplace	unif. fp32		1e-14	4000	100%	100%
	unif. bfloat16		3e-7	4000	50%	—
	adapt. CW	2^{-24}	2e-14	4000	77%	77%
	adapt. CW	2^{-20}	1e-13	4000	60%	60%
	adapt. CW	2^{-18}	1e-12	4000	52%	52%
	adapt. CW	2^{-16}	8e-9	4000	46%	—
	adapt. NW	2^{-24}	2e-2	240	67%	—
CoupCons3D	unif. fp32		3e-15	320	100%	100%
	unif. bfloat16		3e-15	880	50%	138%
	adapt. CW	2^{-24}	3e-15	320	68%	68%
	adapt. CW	2^{-20}	4e-15	320	56%	56%
	adapt. CW	2^{-16}	3e-16	480	46%	69%
	adapt. CW	2^{-12}	3e-15	1440	35%	158%
	adapt. NW	2^{-24}	4e-15	1040	36%	117%
	adapt. NW	2^{-20}	2e-6	2240	17%	—
	adapt. NW	2^{-16}	1e-5	1920	1%	—
Geo_1438	unif. fp32		5e-15	3840	100%	100%
	unif. bfloat16		2e-6	4000	50%	—
	adapt. CW	2^{-24}	2e-13	4000	89%	89%
	adapt. CW	2^{-20}	5e-12	3920	71%	—
	adapt. CW	2^{-18}	3e-14	4000	67%	70%
	adapt. CW	2^{-16}	2e-10	4000	60%	—
	adapt. NW	2^{-24}	7e-14	240	61%	4%
	adapt. NW	2^{-20}	7e-14	480	47%	6%
	adapt. NW	2^{-18}	1e-5	240	39%	—
Serena	unif. fp32		4e-15	1840	100%	100%
	unif. bfloat16		3e-7	4000	50%	—
	adapt. CW	2^{-24}	3e-15	1840	94%	94%
	adapt. CW	2^{-20}	5e-15	2560	74%	102%
	adapt. CW	2^{-16}	5e-15	2160	64%	75%
	adapt. CW	2^{-12}	5e-7	4000	48%	—
	adapt. NW	2^{-24}	8e-15	480	57%	15%
	adapt. NW	2^{-20}	3e-6	480	43%	—
StocF-1465	unif. fp32		2e-8	720	100%	100%
	unif. bfloat16		4e-7	4000	50%	—
	adapt. CW	2^{-24}	2e-8	720	61%	61%
	adapt. CW	2^{-20}	2e-8	720	45%	45%
	adapt. CW	2^{-16}	2e-8	720	28%	28%
	adapt. CW	2^{-12}	2e-8	1680	20%	47%
	adapt. CW	2^{-8}	3e-7	480	11%	7%
	adapt. NW	2^{-24}	2e-8	80	3%	$< 1\%$
	adapt. NW	2^{-20}	7e-8	960	1%	1%
	adapt. NW	2^{-16}	2e-6	880	1%	1%
	adapt. NW	2^{-12}	4e-5	800	1%	—
Transport	unif. fp32		3e-8	4000	100%	100%
	unif. bfloat16		1e-5	4000	50%	—
	adapt. NW	2^{-24}	5e-8	4000	72%	72%
	adapt. NW	2^{-20}	2e-7	4000	53%	53%
	adapt. NW	2^{-16}	2e-6	4000	31%	31%
	adapt. NW	2^{-12}	4e-5	4000	20%	—

	adapt. CW	2^{-24}	4e-8	4000	74%	74%
	adapt. CW	2^{-20}	3e-7	4000	61%	61%
	adapt. CW	2^{-16}	2e-7	4000	33%	33%
	adapt. CW	2^{-12}	3e-5	4000	24%	—
Long_Coup_dt0	unif. fp32		1e-11	560	100%	100%
	unif. bfloat16		1e-11	480	50%	50%
	adapt. NW	2^{-24}	3e-11	80	10%	1%
	adapt. NW	2^{-20}	3e-11	80	2%	$< 1\%$
	adapt. NW	2^{-16}	3e-11	80	1%	$< 1\%$
	adapt. NW	2^{-12}	2e-11	240	1%	$< 1\%$
	adapt. NW	2^{-8}	3e-11	80	1%	$< 1\%$
	adapt. CW	2^{-24}	1e-11	480	62%	53%
	adapt. CW	2^{-20}	2e-11	880	50%	78%
	adapt. CW	2^{-16}	2e-11	80	41%	6%
	adapt. CW	2^{-12}	1e-11	640	32%	37%
	adapt. CW	2^{-8}	1e-11	1040	14%	26%
Emilia_923	unif. fp32		2e-6	160	100%	100%
	unif. bfloat16		2e-6	1040	50%	325%
	adapt. NW	2^{-24}	7e-15	1200	30%	225%
	adapt. NW	2^{-20}	3e-6	320	6%	12%
	adapt. NW	2^{-16}	6e-5	80	1%	1%
	adapt. NW	2^{-12}	4e-4	1600	1%	—
	adapt. CW	2^{-24}	2e-6	160	82%	82%
	adapt. CW	2^{-20}	2e-6	160	69%	169%
	adapt. CW	2^{-16}	2e-6	160	54%	54%
	adapt. CW	2^{-12}	2e-6	320	39%	78%
	adapt. CW	2^{-8}	1e-6	400	14%	35%
Fault_639	unif. fp32		1e-7	3250	100%	100%
	unif. bfloat16		1e-6	1760	50%	27%
	adapt. NW	2^{-24}	2e-7	160	25%	1%
	adapt. NW	2^{-20}	2e-6	1280	7%	3%
	adapt. NW	2^{-16}	9e-5	400	1%	—
	adapt. CW	2^{-24}	1e-7	3840	85%	100%
	adapt. CW	2^{-20}	3e-7	3360	68%	70%
	adapt. CW	2^{-16}	5e-7	4000	56%	69%
	adapt. CW	2^{-12}	1e-6	960	40%	12%
	adapt. CW	2^{-8}	2e-6	480	21%	3%
Hook_1498	unif. fp32		7e-7	3680	100%	100%
	unif. bfloat16		9e-6	3600	50%	49%
	adapt. NW	2^{-24}	7e-7	3440	76%	71%
	adapt. NW	2^{-20}	8e-7	3940	66%	71%
	adapt. NW	2^{-16}	1e-15	1040	49%	14%
	adapt. NW	2^{-12}	1e-15	1040	37%	10%
	adapt. NW	2^{-8}	7e-3	80	7%	—
	adapt. CW	2^{-24}	7e-7	3680	90%	90%
	adapt. CW	2^{-20}	7e-7	3680	70%	70%
	adapt. CW	2^{-16}	2e-6	3360	60%	55%
	adapt. CW	2^{-12}	1e-5	1760	44%	21%
	adapt. CW	2^{-8}	7e-3	160	27%	—

Our experiments on a wide range of sparse matrices from real-life applications have demonstrated the significant potential of the method. The adaptive precision algorithm achieves storage reductions of up to a factor 36× compared with the uniform precision algorithm, and these reductions translate to large time speedups on a multicore computer, up to a factor 7×; these

gains are achieved while maintaining an accuracy comparable to that of the uniform precision algorithm. We have then investigated the use of our adaptive precision SpMV within a GMRES solver for the solution of sparse linear systems. We have shown that the convergence speed of GMRES is essentially unaffected by the use of adaptive precision SpMV with conservative choices for the value of ϵ, such as $\epsilon = 2^{-24}$, which yields an equivalent accuracy to using a uniform fp32 precision SpMV. Moreover, we have shown that using larger values of ϵ may often be beneficial by reducing the SpMV cost at the expense of a possibly slower convergence. Since ϵ does not need to correspond to the unit roundoff of a floating-point arithmetic, our adaptive precision GMRES is not constrained by the available precisions on the hardware and can achieve a flexible compromise between cost per iteration and total number of iterations.

While we have focused here on the GMRES solver with a simple diagonal preconditioner, our adaptive precision framework is general and we expect it to be usable in other contexts. For example, we expect it to behave similarly with other iterative methods such as CG or FGMRES. In future work we wish to extend the adaptive precision framework to cover other crucial steps of the solver, such as the construction of the Krylov basis or the preconditioner.

Acknowledgements. We thank our industrial partners for providing some of the test matrices used in this paper. This work was supported by the InterFLOP (ANR-20-CE46-0009) project of the French National Agency for Research (ANR) and the interdisciplinary CNRS project CASSIDI.

REFERENCES

[1] K. Ahmad, H. Sundar, and M. Hall, *Data-driven mixed precision sparse matrix vector multiplication for GPUs*, ACM Trans. Archit. Code Optim., 16 (2019), https://doi.org/10.1145/3371275, https://doi.org/10.1145/3371275.

[2] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé, *Five-precision GMRES-based iterative refinement*, MIMS EPrint 2021.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2021, http://eprints.maths.manchester.ac.uk/2852/. Revised April 2022.

[3] P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary, *Mixed precision low rank approximations and their application to block low rank lu factorization*, IMA J. Numer. Anal., (2022), https://doi.org/10.1093/imanum/drac037.

[4] 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 Computat. Pract. Exper., 31 (2019), p. e4460, https://doi.org/10.1002/cpe.4460.

[5] E. Carson and N. J. Higham, *A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems*, SIAM J. Sci. Comput., 39 (2017), pp. A2834–A2856, https://doi.org/10.1137/17M1122918.

[6] E. Carson and N. J. Higham, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM J. Sci. Comput., 40 (2018), pp. A817–A847, https://doi.org/10.1137/17M1140819.

[7] M. P. Connolly, N. J. Higham, and T. Mary, *Stochastic rounding and its probabilistic backward error analysis*, SIAM J. Sci. Comput., 43 (2021), pp. A566–A585, https://doi.org/10.1137/20m1334796.

[8] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Software, 38 (2011), pp. 1:1–1:25, https://doi.org/10.1145/2049662.2049663.

[9] J. Diffenderfer, D. Osei-Kuffuor, and H. Menon, *QDOT: Quantized dot product kernel for approximate high-performance computing*, ArXiv:2105.00115, Apr. 2021, https://arxiv.org/abs/2105.00115.

[10] G. Flegar, H. Anzt, T. Cojean, and E. S. Quintana-Ortí, *Adaptive precision block-Jacobi for high performance preconditioning in the Ginkgo linear algebra software*, ACM Trans. Math. Software, 47 (2021), pp. 1–28, https://doi.org/10.1145/3441850.

[11] 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, (2021).

[12] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002, https://doi.org/10.1137/1.9780898718027.

[13] N. J. Higham and T. Mary, *A new preconditioner that exploits low-rank approximations to factorization error*, SIAM J. Sci. Comput., 41 (2019), pp. A59–A82, https://doi.org/10.1137/18M1182802.

[14] N. J. Higham and T. Mary, *Sharper probabilistic backward error analysis for basic linear algebra kernels with random data*, SIAM J. Sci. Comput., 42 (2020), pp. A3427–A3446, https://doi.org/10.1137/20M1314355.

[15] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, Acta Numerica, 31 (2022), pp. 347–414, https://doi.org/10.1017/s0962492922000022.

[16] C.-P. Jeannerod and S. M. Rump, *Improved error bounds for inner products in floating-point arithmetic*, SIAM J.Matrix Anal. Appl., 34 (2013), https://doi.org/34-2/89448, http://www.siam.org/journals/

simax/34-2/89448.html.

[17] M. Lange and S. M. Rump, *Error estimates for the summation of real numbers with application to floating-point summation*, BIT Numerical Mathematics, 57 (2017), pp. 927–941.

[18] N. Lindquist, P. Luszczek, and J. Dongarra, *Improving the performance of the GMRES method using mixed-precision techniques*, in Communications in Computer and Information Science, J. Nichols, B. Verastegui, A. B. Maccabe, O. Hernandez, S. Parete-Koon, and T. Ahearn, eds., Springer, Cham, Switzerland, 2020, pp. 51–66, https://doi.org/10.1007/978-3-030-63393-6_4.

[19] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam, *Experimental evaluation of multiprecision strategies for GMRES on GPUs*, ArXiv:2105.07544, May 2021, https://arxiv.org/abs/2105.07544.

[20] D. Mukunoki and T. Imamura, *Reduced-precision floating-point formats on GPUs for high performance and energy efficient computation*, in 2016 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2016, pp. 144–145.

[21] W. Oettli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numer. Math., 6 (1964), pp. 405–409, https://doi.org/10.1007/BF01386090.

[22] C. C. Paige, M. Rozložník, and Z. Strakoš, *Modified Gram-Schmidt (MGS), least squares, and backward stability of MGS-GMRES*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 264–284, https://doi.org/10.1137/050630416.

[23] F. Renac, M. de la Llave Plata, E. Martin, J. B. Chapelier, and V. Couaillier, *Aghora: A High-Order DG Solver for Turbulent Flow Simulations*, Springer International Publishing, Cham, 2015, pp. 315–335.

[24] J. Rigal and J. Gaches, *On the compatibility of a given solution with the data of a linear system*, Journal of the ACM, 14 (1967), pp. 526–543.

[25] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, second ed., 2003, https://doi.org/10.1137/1.9780898718003.