SIAM J. SCI. COMPUT. © 2024 Society for Industrial and Applied Mathematics
Vol. 46, No. 1, pp. C30–C56

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 various Krylov solvers 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, Krylov solver, GMRES, CG, BiCGstab, iterative solver, linear system

MSC codes. 65G50, 65F05, 65F08, 65F50, 65F10

DOI. 10.1137/22M1522619

1. Introduction.

Motivated by the growing availability of lower precision arithmetics, mixed precision algorithms are being developed for a wide range of numerical computations [18]. 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. [6], [13] 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. [5], 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, Sundar, and Hall [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, Osei-Kuffour, and Menon [12] propose a "quantized"

[*]Submitted to the journal's Machine Learning Methods for Scientific Computing section September 16, 2022; accepted for publication (in revised form) September 5, 2023; published electronically January 25, 2024.
https://doi.org/10.1137/22M1522619

Funding: This work was partially supported by the French National Agency for Research (ANR) projects InterFLOP (ANR-20-CE46-0009) and NuSCAP (ANR-20-CE48-0014), and by the interdisciplinary CNRS project CASSIDI.

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

[‡]Sorbonne Université, CNRS, LIP6, Paris, F-75005, France, 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).

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 [18, 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 (discarding sufficiently small elements), 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 various Krylov solvers 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 and carry out its error analysis. In section 4, we investigate experimentally both its accuracy and performance. In section 5 we apply this algorithm to the solution of linear systems with Krylov solvers. Finally, we provide our concluding remarks in section 6.

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 this article we use the standard model of floating-point arithmetic [15, sec. 2.2]

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

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

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 this article, we have used the analysis of inner products of Jeannerod and Rump [19] to obtain more refined bounds, where constants of the form $\gamma_n = nu/(1-nu)$ can be replaced simply

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:   $y_i = 0$
5:   for $j \in J_i$ do
6:     $y_i \leftarrow y_i + a_{ij}x_j$
7:   end for
8: end for
```

by nu. The analysis of [19] assumes the use of rounding to the nearest, but it was later shown in [20, 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 [16, 17, 10]. 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.

As a consequence of the Oettli–Prager [15, Thm. 7.3], [24] and Rigal–Gaches [15, Thm. 7.1], [27] 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: Error analysis.

In this section we propose an adaptive precision matrix–vector product algorithm. To do so we perform 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

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

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

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.

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.

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

(3.5)

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

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

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 to be 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

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

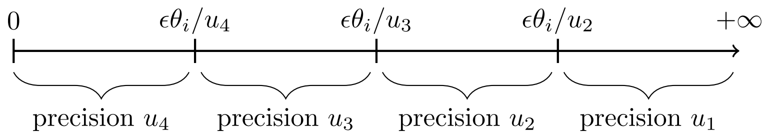


FIG. 3.1. *Illustration of the bucket construction with four precisions $u_1 < u_2 < u_3 < u_4$. The real line $[0, +\infty)$ is partitioned into intervals P_{ik} defined by* (3.9) *(componentwise criteria, $\theta_i = |a_i|^T|x|$) or* (3.17) *(normwise criteria, $\theta_i = \|A\|$).*

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. Indeed, as a sum of positive values, the problem is well conditioned.

Since, by (2.5), $\varepsilon_{\rm nw} \le \varepsilon_{\rm cw}$, this bucket construction also yields a normwise backward error in $O(\epsilon)$. However, if we need only 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

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

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 $\|A\|$:

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

This is sufficient to ensure that $\beta_{ik} \le p_{ik}\epsilon/u_k$ and thus that $\varepsilon_{\rm nw} = O(\epsilon)$. However, in this case we can no longer guarantee a small $\varepsilon_{\rm cw}$, 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*

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

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

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

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 $A = e = [1, \dots, 1]$.

3.1. 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 one 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:

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

where $e = [1, \dots, 1]^T$, 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)

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

Table 4.1
List of precision formats used in our experiments.

	Numbers of bits				
	Sign	Exponent	Significand	Range	Unit roundoff
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	$10^{\pm 308}$	$2^{-53} \approx 1 \times 10^{-16}$

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

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

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 $\varepsilon_{\mathrm{cw}}$ than construction (3.16)–(3.17). It is important to note that, unfortunately, construction (3.19)–(3.20) cannot always guarantee a small $\varepsilon_{\mathrm{cw}}$, since the ratio $|a_i|^T e/|a_i|^T|x|$ can be arbitrarily large for an unlucky choice of vector x.

4. Adaptive precision SpMV: 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.

4.1. Implementation.

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 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 eight 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 eight 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 4.1.

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 cast implementation, which uses the `MVBITS` subroutine of the GNU Fortran compiler. To be specific, for each coefficient we chop the desired bits by moving the bits that are to be kept in a variable of smaller size; for example, to cast an fp32 variable to fp24 format, we move the leading 24-bit to a 3-byte variable. Our

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

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 "cast back" operation also relies on `MVBITS`: we simply move all the bits into an fp32 or fp64 variable and add as many zeros as needed. For example, to cast an fp24 variable back to fp32, we must add one byte of zeros.

We must mention that this cast implementation is far from optimized, and leads to a heavy performance overhead. We aim to use it only to validate the numerical behavior of our approach, rather than to provide acceleration with custom precision formats. However, it is important to note that achieving performance gains from the use of custom precisions is certainly possible, by relying on more efficient, lightweight cast implementations. For example, such implementations are described by Mukunoki and Imamura [23], or more recently by Grützmacher, Anzt, and Quintana-Ort\'i [14]. This 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. The implementation of the adaptive precision SpMV on top of such an efficient accessor is, therefore, one of the main research perspectives of this work.

Our SpMV implementation uses the CSR format for all matrices and is multi-threaded by parallelizing the loop on the row indices with OpenMP. We recall that the CSR format consists of a row index array of size $n+1$, a column index array of size nnz, and a value array of size nnz. As a result, in the uniform precision case, the total storage for the matrix is equal to

$$(nnz + n + 1)s_{\text{int}} + nnz s_{\text{fp}}, \tag{4.1}$$

where s_{int} is the size of the integer type and s_{fp} is the size of the floating-point type. For all our matrices, 4-byte integers suffice. For the adaptive precision SpMV, we use a different CSR matrix for each precision. Since each nonzero element belongs to a unique CSR matrix, the column index and value arrays of size nnz are splitted among the different CSR matrices, and so do not require any extra storage. However, the row index array of size $n+1$ must be duplicated. This represents a storage increase of approximately qns_{int}, where q is the number of precisions. In most cases this increase is compensated by the storage reduction of the floating-point values, but for matrices with low potential for low precisions and a small number of nonzeros per row (small nnz/n ratio), this may lead to a noticeable overhead cost. In our experiments we take into account the cost of reading the indices in addition to the one of reading the floating-point values when measuring the storage cost of the SpMV. In particular, the index access cost explains why the use of dropping may have a huge impact on the performance: storing an element in any precision does not change the need to store its column index, whereas dropping it allows for dropping its index too. We will further analyze this effect in section 4.4.

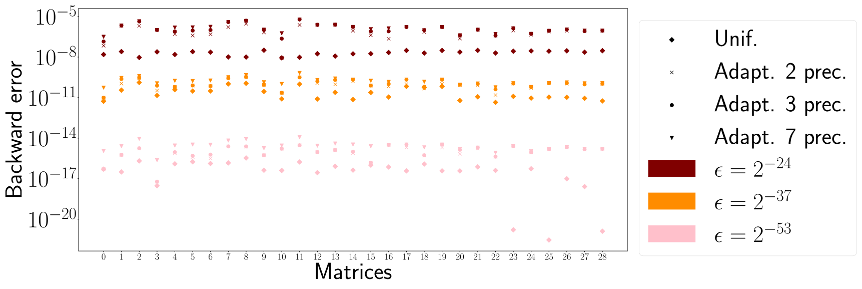
4.2. Experimental setting. All the experiments were performed on one node of the Olympe supercomputer, which is equipped with two 18-core Intel Skylake 6140 processors (for a total of 36 cores). We use 18 threads thoughout the experiments, as this seems to be the optimal setting as we will observe in section 4.5. For the time measurements, we perform one hundred products and report the average timings. We do not include the time for reading the matrix from a file and putting it into

TABLE 4.2
List of matrices used in our experiments.

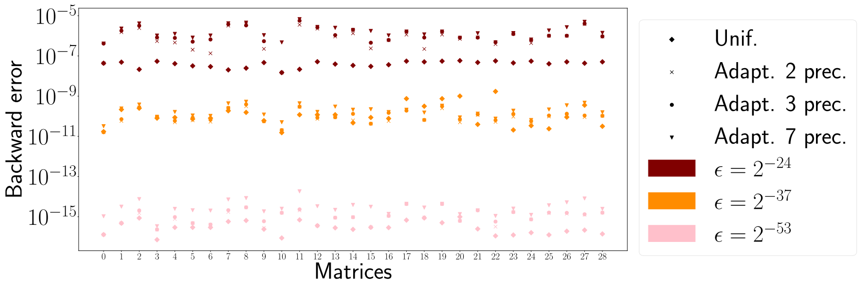
Number	Matrix	n	nnz
0	Transport	$1.6e+06$	$2.4e+07$
1	Serena	$1.4e+06$	$3.3e+07$
2	A_DGO3.mtx	$1.5e+05$	$1.8e+07$
3	vas_stokes_4M	$4.4e+06$	$1.3e+08$
4	A_DGO4.mtx	$2.6e+05$	$5.1e+07$
5	Emilia_923	$9.2e+05$	$2.1e+07$
6	A_DGO5.mtx	$3.8e+05$	$1.1e+08$
7	Hook_1498	$1.5e+06$	$3.1e+07$
8	ML_Geer	$1.5e+06$	$1.1e+08$
9	ML_Laplace	$3.8e+05$	$2.8e+07$
10	vas_stokes_1M	$1.1e+06$	$3.5e+07$
11	stokes	$1.1e+07$	$3.5e+08$
12	Geo_1438	$1.4e+06$	$3.2e+07$
13	ss	$1.7e+06$	$3.5e+07$
14	vas_stokes_2M	$2.1e+06$	$6.5e+07$
15	Fault_639	$6.4e+05$	$1.5e+07$
16	Queen_4147	$4.1e+06$	$1.7e+08$
17	PFlow_742	$7.4e+05$	$1.9e+07$
18	Flan_1565	$1.6e+06$	$5.9e+07$
19	Cube_Coup_dt0	$2.2e+06$	$6.5e+07$
20	Long_Coup_dt6	$1.5e+06$	$4.4e+07$
21	CoupCons3D	$4.2e+05$	$2.2e+07$
22	Long_Coup_dt0	$1.5e+06$	$4.4e+07$
23	StocF-1465	$1.5e+06$	$1.1e+07$
24	nv2	$1.5e+06$	$5.3e+07$
25	power9	$1.6e+05$	$2.5e+06$
26	test1	$3.9e+05$	$1.3e+07$
27	imagesensor	$1.2e+05$	$1.9e+06$
28	mosfet2	$4.7e+04$	$1.5e+06$
29	dgreen	$1.2e+06$	$3.8e+07$
30	radiation	$2.2e+05$	$7.6e+06$
31	nv1	$7.5e+04$	$2.4e+06$

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} (which is 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.

Most of the matrices used in these experiments come from the SuiteSparse collection [11]. The others come from our industrial partners (see Table 4.2) and are described below. 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



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

ONERA Aghora DG solver [26] 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 4.2. As for the vector x, we set it to $e = [1, \dots, 1]^T$ throughout the experiments. We emphasize that our adaptive precision SpMV is guaranteed to deliver the requested accuracy ϵ, and so must "work" for any matrix. The worst possible behavior is obtained for a matrix that presents no potential for mixed precision, which will lead the adaptive precision algorithm to use the highest precision for all elements, becoming equivalent to the uniform precision algorithm.

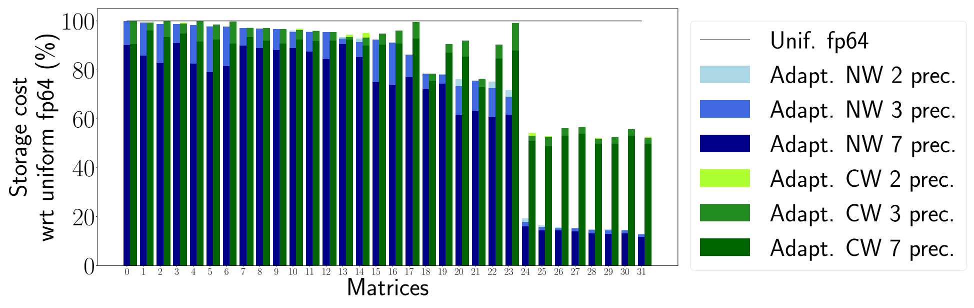
4.3. Main results. We begin in Figure 4.1 by evaluating the accuracy of our adaptive precision algorithm to confirm that we are able to control the backward error, which, according to Theorem 3.1, should remain of order ϵ. We check this both for the normwise and componentwise backward errors by plotting, in Figure 4.1a, the normwise backward error for the algorithm with the normwise bucket criteria (3.16)–(3.17), and, in Figure 4.1b, 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 seven precision formats, we only use two (fp64 and fp32) or three (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.

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.

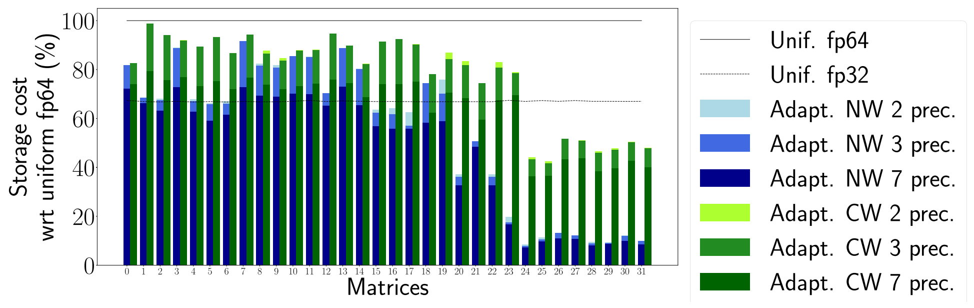
Figure 4.2 plots the storage cost of the adaptive precision algorithm as a percentage of the uniform precision fp64 algorithm. As for Figure 4.1, several configurations of the adaptive precision algorithm are tested, depending on the accuracy target ($\epsilon = 2^{-24}$, $\epsilon = 2^{-37}$, $\epsilon = 2^{-53}$), 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 4.2c) is nearly divided by two when adding bfloat16 (three precisions instead of two). 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 4.2a) to fp32 (Figure 4.2c) 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 seven 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 $36\times$ in the best case.

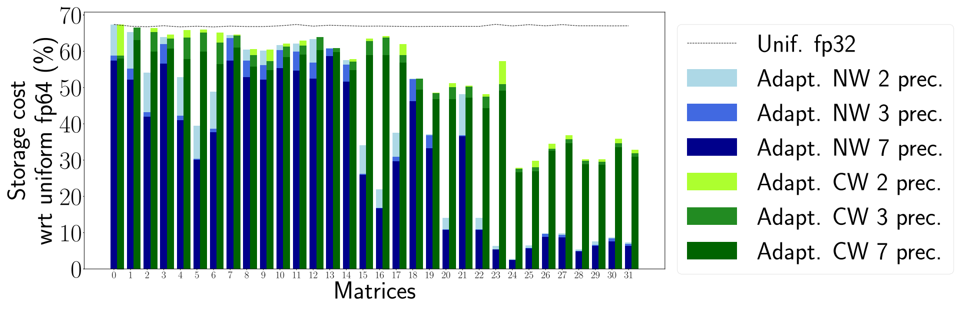
Finally, we measure the execution time of the algorithms. Since SpMV is memory bound, in principle we can hope the time gains to roughly follow the storage gains, even though the execution time depends on several other factors such as the overhead cost of the cast operations and the latency costs. In 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, as mentioned in section 4.1, our cast implementation is not optimized and is only designed to validate the numerical behavior of the adaptive precision algorithm. As a result, we have found the use of other custom precision formats to lead to slowdowns due to a heavy performance penalty associated with our



(a) accuracy target $\epsilon = 2^{-53}$



(b) accuracy target $\epsilon = 2^{-37}$



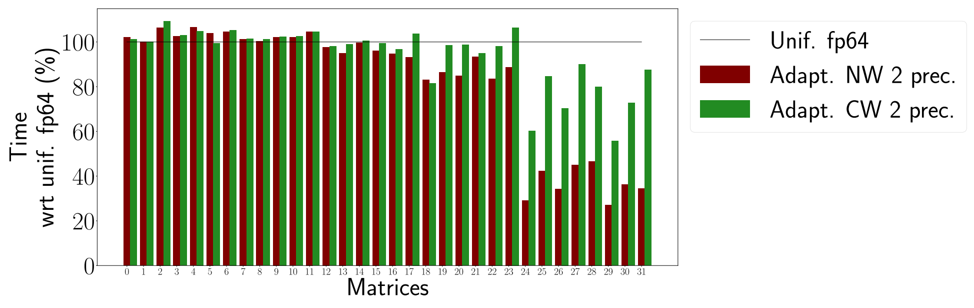
(c) accuracy target $\epsilon = 2^{-24}$

FIG. 4.2. *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. (Figure in color online.)*

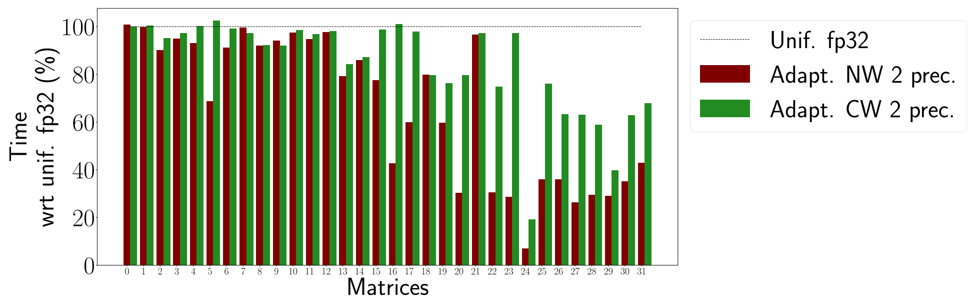
cast implementation, and restrict our time performance analysis to the two-precision version plus dropping.

Figure 4.3 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



(a) accuracy target $\epsilon = 2^{-53}$



(b) accuracy target $\epsilon = 2^{-24}$

FIG. 4.3. *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.*

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.

Finally, we also report the execution time in the case of an $\epsilon = 2^{-37}$ accuracy target in Figure 4.4. The figure also plots the time for the $\epsilon = 2^{-24}$ and $\epsilon = 2^{-53}$ targets (already presented in Figure 4.3) as a point of comparison. Figure 4.4 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.

4.4. Effect of dropping. The performance gains achieved by the adaptive precision SpMV are obtained thanks to the use of lower precisions but also the use of dropping. As noted in Remark 3.2, our error analysis fully accounts for the use of dropping, which effectively behaves as a precision format with unit roundoff $u_q = 1$. Nevertheless, the effect of dropping on the performance of the SpMV is quite different from the effect of lower precisions. This is because dropping increases the sparsity of the matrix and therefore allows for reducing the storage for indices too. For example, for 4-byte indices, using the two precision formats fp64 and fp32 but not dropping, the adaptive precision storage can be no less than 66% of the uniform precision one, since we must still store about $8nnz$ bytes ($4nnz$ for the indices, and $4nnz$ for the

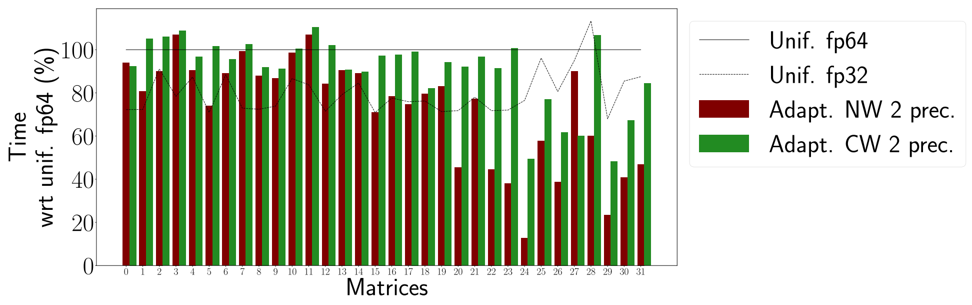


FIG. 4.4. *Execution time of the adaptive precision SpMV for an $\epsilon = 2^{-37}$ target accuracy, as a percentage of the execution time of the uniform precision fp64 SpMV. Both the normwise (NW) or componentwise (CW) criteria are reported.*

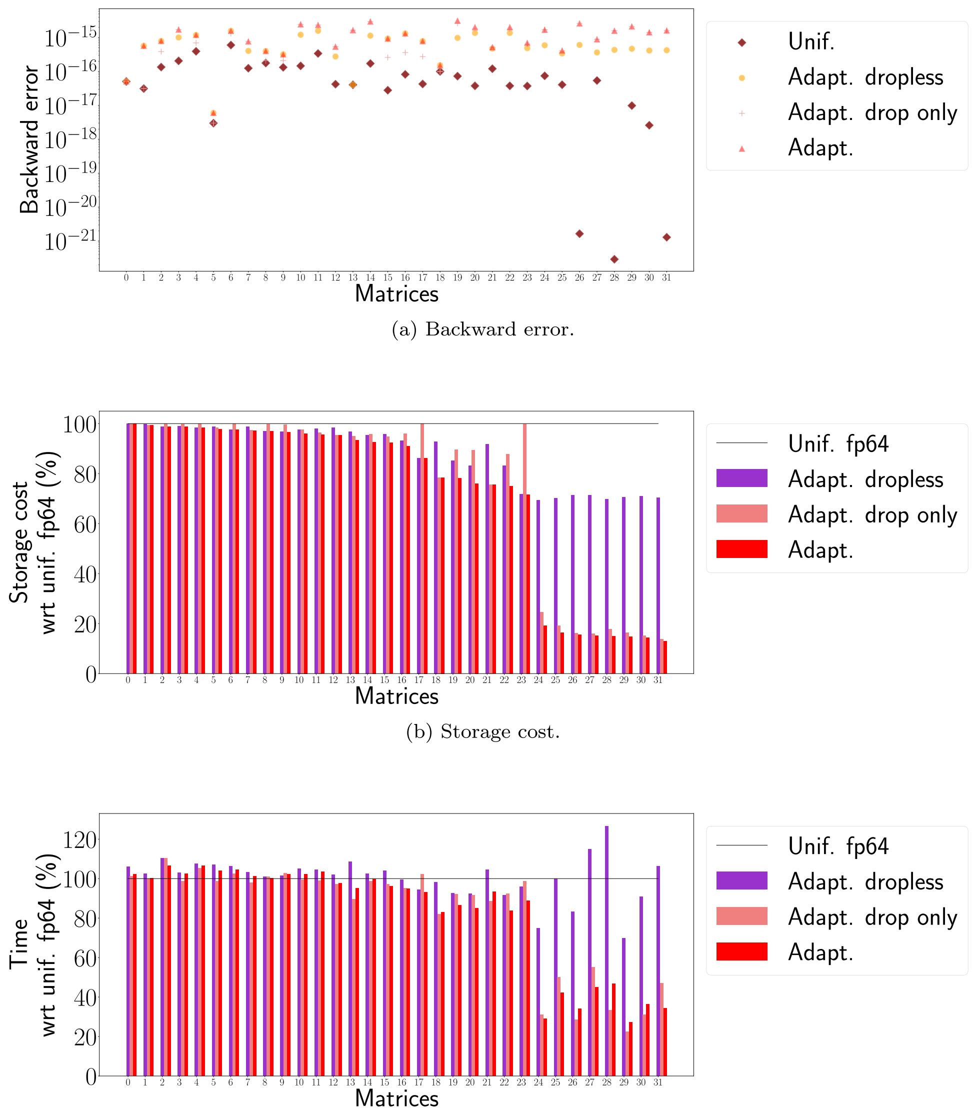
values, in the best case where all can be switched to fp32). In contrast, dropping the elements allows for dropping the associated indices, and therefore much larger gains can be obtained. The goal of this section is to analyze this effect more precisely by evaluating the impact of both dropping and low precisions separately.

We plot in Figures 4.5a, 4.5b, and 4.5c the accuracy, storage, and time, respectively, of four SpMV variants: uniform fp64 (Unif. fp64), adaptive with two precisions (fp64 and fp32) but no dropping (Adapt. dropless), adaptive with only one precision (fp64) and dropping (Adapt. drop only), and adaptive with two precisions and dropping at the same time (Adapt.). All three adaptive variants use an accuracy target $\epsilon = 2^{-53}$.

Figure 4.5a shows that both approximation tools used by the adaptive method (dropping and precision reduction) each slightly increase the error, but all variants remain of the order of the requested accuracy ϵ. As expected, Figures 4.5b and 4.5c show that the adaptive SpMV benefits both from the use of multiple precisions and of dropping, separately or combined. In some cases, dropping has a massive impact and is the main contributor to the performance gains, but in other cases, dropping has almost no effect and it is the use of multiple precisions that is responsible for most of the gains. All in all, this confirms the relevance of using an adaptive SpMV that combines both techniques.

4.5. Parallel scaling analysis. We conclude by analyzing the scalability of our SpMV implementation. For this analysis we use matrix Cube_Coup_dt0, which is one of the largest in our set; we have observed similar trends on other matrices. Figure 4.6a compares the uniform and adaptive precision methods with a number of threads increasing from 1 to 36 (the total number of cores on the shared-memory node). The figure shows that both methods scale well up to 18 threads, and suffer a slowdown going from 18 to 19 threads. This is due to the NUMA architecture of the node, which consists of two 18-core sockets. This is particularly visible on Figure 4.6b, which plots the parallel efficiency of the methods and shows a major loss of efficiency between 18 and 19 threads. These observations have led us to choose a number of threads equal to 18 for all experiments, in order to maximize the data locality and performance of the methods.

5. Application to Krylov 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



(a) Backward error.

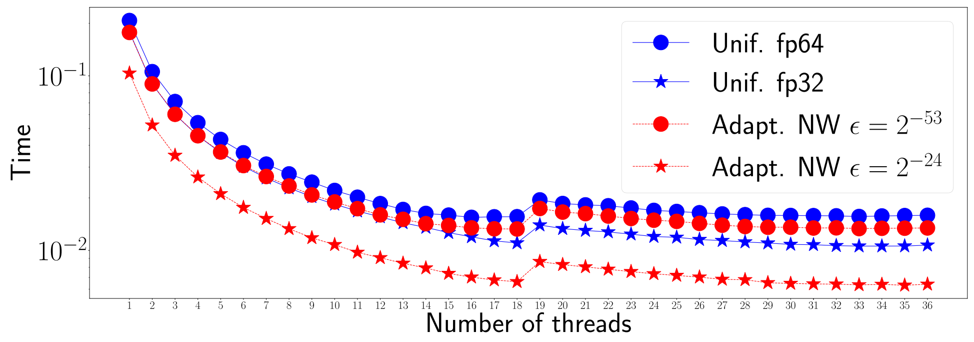
(b) Storage cost.

(c) Time cost.

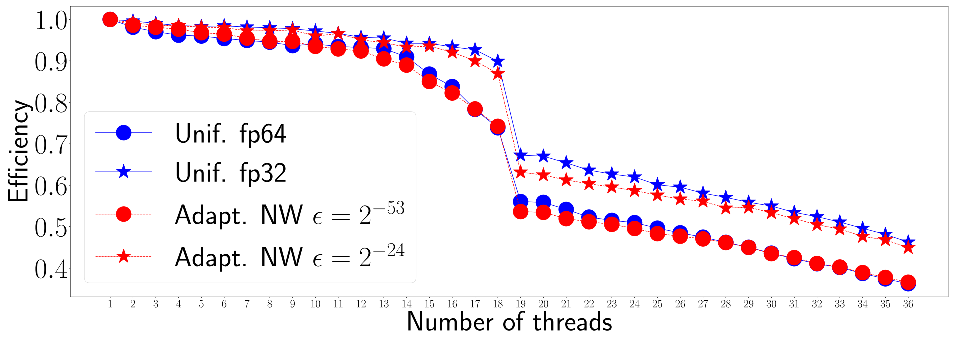
FIG. 4.5. *Backward error, storage cost, and time cost of four SpMV variants: fp64 uniform precision (Unif. fp64), adaptive precision with two precisions but no dropping (Adapt. dropless), adaptive precision with only one precision and dropping (Adapt. drop only), and adaptive precision with both two precisions and dropping (Adapt.). All three adaptive variants use $\epsilon = 2^{-53}$ as target accuracy.*

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).

5.1. Adaptive precision Krylov solvers. We will focus our discussion and experiments on three choices of Krylov solvers [28]: GMRES, CG, and BiCGStab, respectively outlined in Algorithms 5.1, 5.2, and 5.3. CG is specifically designed for



(a) Scaling on 1 to 36 threads



(b) Parallel efficiency on 1 to 36 threads

FIG. 4.6. *Parallel scaling experiments on Cube_Coup_dt*0.

Algorithm 5.1. 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:    $y = Aq_k$
6:    for $j = 1: k$ do
7:      $h_{jk} = q_j^T y$
8:      $y = y - h_{jk} q_j$
9:    end for
10:   $h_{k+1,k} = \|y\|_2$
11:   $q_{k+1} = y/h_{k+1,k}$
12:   Solve the least squares problem $\min_{c_k} \|Hc_k - \beta e_1\|_2$.
13:   $x_k = x_0 + Q_k c_k$
14: end for
```

Algorithm 5.2. CG.

```
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: $z = M^{-1}r$
 3: $p = z$
 4: $k = 0$
 5: for $k = 1, 2, \ldots$ do
 6:     $\alpha_k = \frac{{r_k}^T z_k}{{p_k}^T A p_k}$
 7:     $x_{k+1} = x_k + \alpha_k p_k$
 8:     $r_{k+1} = r_k - \alpha_k A p_k$
 9:     $z_{k+1} = M^{-1} r_{k+1}$
10:     $\beta_k = \frac{{r_{k+1}}^T (z_{k+1} - z_k)}{{r_k}^T z_k}$
11:     $p_{k+1} = z_{k+1} + \beta_k p_k$
12: end for
```

Algorithm 5.3. BiCGStab.

```
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: $rho_0 = \alpha = \omega_0 = 1$
 3: $v_0 = p_0 = 0$
 4: for $k = 1, 2, \ldots$ do
 5:     $\rho_i = \hat{r_0}^T r_{i-1}$
 6:     $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
 7:     $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
 8:     $v_i = Ap_i$
 9:     $\alpha = \rho_i/(\hat{r_0}^T v_i)$
10:     $h = x_{i-1} + \alpha p_i$
11:     $s = r_{i-1} - \alpha v_i$
12:     $t = As$
13:     $\omega_i = (t^T s)/(t^T t)$
14:     $x_i = h + \omega_i s$
15:     $r_i = s - \omega_i t$
16: end for
```

symmetric positive-definite matrices. BiCGStab is designed to handle general matrices by building two Krylov subspaces (thus requiring two SpMVs per iteration); it incorporates a stabilization step compared with the original BiCG algorithm. Finally, GMRES is the most robust Krylov method; it relies on the construction of an orthonormal basis for the Krylov subspace, whose size grows at each iteration. This requires computationally expensive orthogonalization operations, whose cost can be limited by restarting the method. In the case of CG and BiCGStab, the SpMV is usually the computational bottleneck; for the GMRES algorithm, the orthogonalization

Algorithm 5.4. Krylov-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:     $r_i = b - Ax_{i-1}$
3:     Solve $\widetilde{A}d_i = r_i$ by a Krylov method (Algorithms 5.1, 5.2, or 5.3) using SpMVs
       with a lower precision matrix $\widetilde{A}$.
4:     $x_i = x_{i-1} + d_i$
5: end for
```

of the Krylov basis is also expensive, but nevertheless the SpMV still represents a significant fraction of the total. Therefore, by accelerating the SpMV in these Krylov methods we can expect significant speedups on the whole solution, provided that the convergence 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 [25], 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. More formally, we can prove this by relying on the recent analysis of Amestoy et al. [3]. Indeed, [3, 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

$$\widehat{y} = Aq + f, \quad \|f\| \le \epsilon\|A\|\|q\|, \tag{5.1}$$

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. Moreover, neither CG nor BiCGStab are backward stable. Nevertheless, we will test GMRES with both the normwise and componentwise criteria for SpMV, because we have experimentally observed that using a componentwise stable SpMV can in some cases improve the convergence behavior of GMRES compared with using an only normwise stable SpMV. We will experiment with both criteria for the CG and BICGStab algorithms too.

5.1.1. Iterative refinement. In section 4, 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 iterative refinement based on Krylov solvers, such as GMRES-IR [18, sect. 8], [8, 9, 3, 21, 22]. Iterative refinement, described in Algorithm 5.4, 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 a Krylov method (Algorithms 5.1, 5.2, or 5.3, the inner loop). Note that for GMRES, Algorithm 5.4 is equivalent to restarted GMRES when the inner GMRES on line 3 is initialized with $d_0 = 0$.

Importantly, it is known that Algorithm 5.4 can converge to a high accuracy even when the inner Krylov method is performed entirely in low precision [18, sect. 8], [3, 9].

In our adaptive precision context, we can, therefore, leave the outer loop SpMV (line 2 of Algorithm 5.4) in high (uniform) precision, and perform the inner loop SpMV of Algorithms 5.1, 5.2, and 5.3 with an approximate matrix $\widetilde{A}$ that exploits adaptive precision with a low accuracy target ϵ. Since the inner loop SpMV is called many more times than the outer loop one, we can expect the cost of the overall iterative refinement solution to be determined by the cost of the low accuracy inner loop SpMV.

In the following, we will assess experimentally the impact of using an adaptive precision SpMV in the inner loop on the convergence and performance of the solution. We incorporate a row scaling 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.

Finally, we note that using adaptive precision for the SpMV is not the only possible strategy to exploit mixed precision in Krylov solvers; many other approaches have been proposed in the literature. In addition to the approaches that belong to the iterative refinement class mentioned above [8, 9, 3, 4], other possible ones use the low precision for the Krylov basis [2], or adaptively decreasing the precision as the iterations go based on inexact Krylov theory [29]. We emphasize that our adaptive precision SpMV algorithm is complementary to these strategies, and could be combined with them.

5.2. Adaptive GMRES-IR convergence analysis. We begin by analyzing how the use of adaptive precision SpMV affects the convergence of GMRES-IR. The goal of this section is to compare the use of uniform and adaptive precision SpMV and to analyze the effect of different parameters, mainly the accuracy target ϵ and the choice between componentwise (CW hereinafter) or normwise (NW) criteria. We illustrate different aspects of the behavior of adaptive precision GMRES-IR by using three examples, matrices ML_Laplace, CoupCons3D, and Geo_1438.

Figure 5.1 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 and with CW criteria; for this matrix, the use of NW criteria significantly degrades the convergence (not shown). In the legend, we indicate the adaptive precision SpMV cost as a percentage of the fp32 uniform precision one. The figure shows that ϵ has an effect on both the SpMV cost (and therefore, the cost per iteration of GMRES-IR) and the convergence

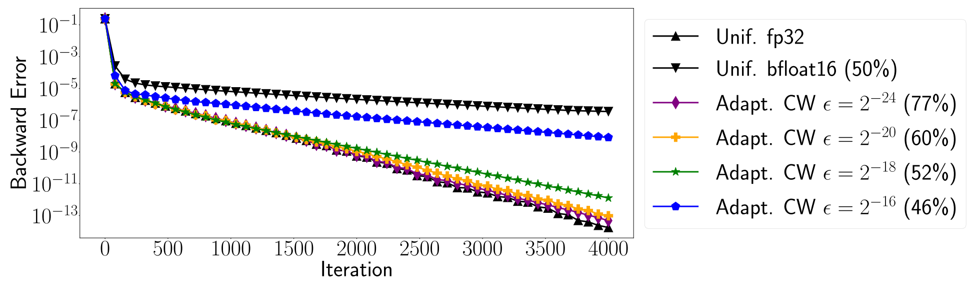


FIG. 5.1. *Convergence of GMRES-IR for matrix ML_Laplace: illustration of the effect of the ϵ parameter.*

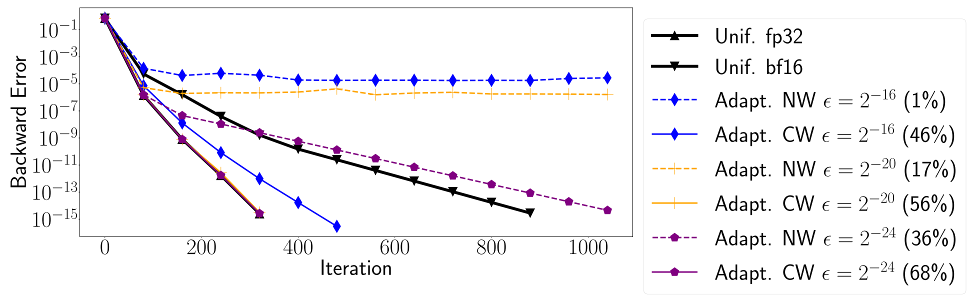


FIG. 5.2. *Convergence of GMRES-IR for matrix CoupCons3D: illustration of the difference between CW and NW criteria.*

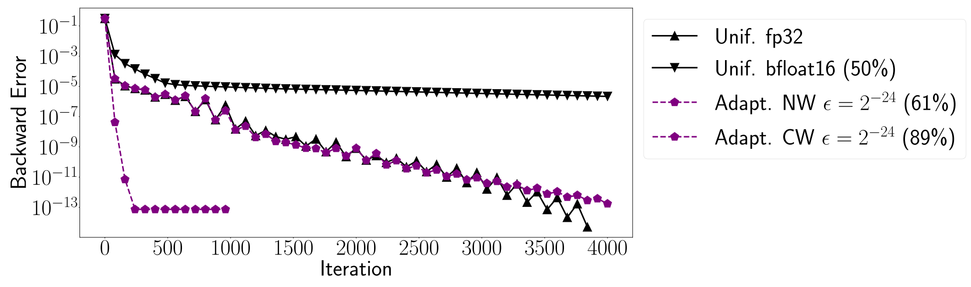


FIG. 5.3. *Convergence of GMRES-IR for matrix Geo_1438: illustration of a surprising behavior of NW variants.*

speed (and therefore, the total number of iterations). For example, with $\epsilon = 2^{-24}$, 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 $\epsilon = 2^{-16}$, the SpMV cost is only 46% of the fp32 uniform precision one, but GMRES-IR converges much slower. The optimal choice of ϵ lies in between these two values; for this matrix, $\epsilon = 2^{-20}$, for example, is a good choice.

Figure 5.2 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 tradeoffs that each option offers: for a fixed value of ϵ, NW variants achieve a lower cost but a slower convergence than CW ones. Therefore, the best choice of ϵ can be different for the NW and CW variants. In this example, $\epsilon = 2^{-24}$ leads to the best NW variant, which converges in 1040 iterations with an SpMV cost of 36% of the fp32 uniform one, whereas $\epsilon = 2^{-20}$ 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.

Finally, Figure 5.3 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

Table 5.1
Results with GMRES-IR, BiCGStab-IR, and CG-IR for various matrices and SpMV variants.

			GMRES(80)	GMRES(40)	BiCGstab	CG
CoupCons3D	Time	Uniform	2.09 (0.71)	1.35 (0.62)	1.26 (0.88)	5.31 (2.84)
		Adaptive CW	2.04 (0.71)	1.32 (0.62)	1.47 (1.04)	5.12 (2.75)
		Adaptive NW	4.86 (1.74)	1.86 (0.89)	4.13 (2.96)	5.12 (2.75)
	Error	Uniform	3e-15	4e-13	6e-13	1e-12
		Adaptive CW	4e-15	3e-13	3e-14	1e-12
		Adaptive NW	3e-13	4e-13	3e-13	2e-12
Geo_1438	Time	Uniform	38.76 (11.69)	29.11 (11.82)	38.52 (24.04)	—
		Adaptive CW	38.24 (11.49)	28.54 (11.68)	38.12 (23.71)	—
		Adaptive NW	5.02 (1.38)	1.97 (0.70)	—	—
	Error	Uniform	4e-10	5e-08	9e-07	—
		Adaptive CW	4e-10	3e-08	4e-05	—
		Adaptive NW	7e-14	3e-13	—	—
ML_Laplace	Time	Uniform	11.97 (5.04)	9.36 (5.11)	13.66 (10.23)	—
		Adaptive CW	10.63 (3.69)	7.96 (3.75)	10.79 (7.39)	—
		Adaptive NW	10.51 (3.59)	7.90 (3.68)	—	—
	Error	Uniform	6e-10	2e-08	7e-08	—
		Adaptive CW	4e-09	3e-08	6e-03	—
		Adaptive NW	9e-04	3e-02	—	—
Serena	Time	Uniform	32.25 (10.72)	29.03 (12.94)	39.77 (26.23)	26.70 (13.34)
		Adaptive CW	29.66 (9.80)	29.15 (12.91)	39.74 (25.97)	26.67 (13.29)
		Adaptive NW	8.11 (2.56)	23.17 (10.84)	—	—
	Error	Uniform	1e-13	5e-12	2e-12	2e-05
		Adaptive CW	4e-13	7e-12	2e-12	8e-04
		Adaptive NW	4e-14	9e-08	—	—
ss1	Time	Uniform	0.04 (0.00)	0.03 (0.00)	0.24 (0.18)	0.10 (0.01)
		Adaptive CW	0.03 (0.00)	0.03 (0.00)	0.22 (0.15)	0.16 (0.02)
		Adaptive NW	0.03 (0.00)	0.03 (0.01)	0.23 (0.17)	0.17 (0.04)
	Error	Uniform	6e-13	6e-13	3e-16	7e-09
		Adaptive CW	6e-13	6e-13	3e-15	2e-12
		Adaptive NW	2e-13	6e-13	3e-14	2e-11

aggressively dropping small coefficients from the matrix, the NW variant leads to a "nicer" matrix for which GMRES can converge quickly.

5.3. Performance comparison for different Krylov solvers. To conclude these experiments, we present execution time results in Tables 5.1 and 5.2. We compare four different Krylov solvers: CG, BiCGstab, and GMRES with two different restart sizes (40 or 80). Table 5.1 presents results on the matrices for which GMRES and BiCGStab both converge; for some of these matrices, CG converges too. Table 5.2 presents results on the matrices for which only GMRES converges. For each solver, the tables report the time and the backward error after convergence for different matrices and different SpMV variants: uniform or adaptive precision, with either the CW or NW criteria; we have tested three accuracy targets $\epsilon = 2^{-24}$, 2^{-20}, and 2^{-16}, and report the best for each variant and matrix. We report the total time, as well as the time spent in the SpMV calls between parentheses.

We first note that the total time of BiCGStab (which requires two SpMVs per iteration), and to a lesser extent that of CG (which requires only one), is dominated by the SpMV time. In contrast, the SpMV time represents a smaller, but still significant, fraction of the total time of GMRES; unsurprisingly, this fraction is larger for a smaller restart size. It is also worth noting that for a given matrix, the best solver is not

TABLE 5.2
Results with GMRES-IR for various matrices and SpMV variants.

		GMRES(80)	GMRES(40)	GMRES(80)	GMRES(40)
		Time (s)		Backward error	
Cube_Coup_dt0	Uniform	65.69 (23.43)	49.75 (23.59)	4e-10	5e-10
	Adaptive CW	59.78 (17.64)	44.74 (18.15)	7e-09	8e-09
	Adaptive NW	56.28 (14.03)	41.10 (14.15)	4e-09	4e-09
Emilia_923	Uniform	24.74 (7.53)	18.50 (7.68)	7e-07	8e-07
	Adaptive CW	24.64 (7.69)	18.44 (7.79)	7e-07	8e-07
	Adaptive NW	8.24 (1.90)	3.03 (0.99)	4e-13	5e-13
Fault_639	Uniform	17.38 (5.40)	12.85 (5.46)	3e-07	4e-07
	Adaptive CW	17.25 (5.24)	12.55 (5.27)	5e-07	5e-07
	Adaptive NW	13.99 (2.24)	9.65 (2.30)	2e-06	1e-06
Flan_1565	Uniform	52.34 (22.64)	41.74 (23.02)	5e-07	6e-07
	Adaptive CW	47.91 (18.12)	37.25 (18.46)	7e-07	6e-07
	Adaptive NW	48.02 (18.11)	37.07 (18.15)	6e-07	1e-06
Hook_1498	Uniform	40.38 (11.96)	29.98 (12.15)	1e-06	2e-06
	Adaptive CW	39.96 (11.61)	29.84 (11.78)	2e-06	2e-06
	Adaptive NW	40.40 (11.85)	29.84 (11.99)	2e-06	2e-06
Long_Coup_dt0	Uniform	44.21 (16.27)	33.62 (16.52)	5e-12	5e-12
	Adaptive CW	39.27 (11.81)	29.50 (12.02)	2e-11	8e-12
	Adaptive NW	29.66 (1.53)	19.39 (1.86)	8e-12	2e-11
Long_Coup_dt6	Uniform	44.06 (16.21)	34.07 (16.48)	8e-11	3e-10
	Adaptive CW	40.15 (12.31)	30.45 (12.71)	2e-10	3e-09
	Adaptive NW	29.82 (1.60)	22.91 (5.43)	4e-09	5e-12
ML_Geer	Uniform	48.97 (20.11)	38.72 (20.46)	2e-07	9e-07
	Adaptive CW	45.24 (16.75)	35.23 (17.05)	5e-07	1e-06
	Adaptive NW	44.71 (15.99)	34.46 (16.20)	9e-04	1e-03
PFlow_742	Uniform	20.93 (7.20)	15.83 (7.43)	2e-10	2e-10
	Adaptive CW	19.38 (5.64)	14.21 (5.76)	3e-10	2e-10
	Adaptive NW	15.68 (1.75)	10.27 (1.85)	5e-05	7e-05
Queen_4147	Uniform	164.19 (63.27)	126.04 (64.33)	3e-07	8e-07
	Adaptive CW	159.58 (62.23)	124.42 (63.28)	7e-07	8e-07
	Adaptive NW	110.97 (11.46)	72.69 (12.24)	1e-05	1e-05
StocF-1465	Uniform	31.71 (4.74)	21.88 (4.82)	8e-09	8e-09
	Adaptive CW	30.39 (3.50)	20.60 (3.57)	8e-09	9e-09
	Adaptive NW	28.55 (0.72)	18.06 (0.83)	2e-08	5e-09
Transport	Uniform	35.49 (4.99)	23.92 (4.97)	2e-07	1e-05
	Adaptive CW	33.12 (2.98)	22.03 (3.11)	1e-06	2e-06
	Adaptive NW	33.64 (2.99)	22.06 (3.08)	2e-06	4e-06
dgreen	Uniform	1.78 (0.50)	0.74 (0.26)	3e-15	5e-15
	Adaptive CW	1.47 (0.19)	0.59 (0.11)	5e-16	1e-15
	Adaptive NW	1.42 (0.14)	0.56 (0.06)	1e-15	1e-15
imagesensor	Uniform	0.20 (0.05)	0.07 (0.03)	7e-16	5e-16
	Adaptive CW	0.13 (0.01)	0.05 (0.01)	6e-16	2e-15
	Adaptive NW	0.13 (0.00)	0.05 (0.00)	6e-16	7e-16
mosfet2	Uniform	0.10 (0.04)	0.04 (0.02)	8e-14	6e-14
	Adaptive CW	0.05 (0.01)	0.02 (0.01)	4e-13	1e-13
	Adaptive NW	0.05 (0.01)	0.02 (0.01)	1e-13	3e-14
nv1	Uniform	0.15 (0.06)	0.06 (0.03)	1e-18	4e-18
	Adaptive CW	0.10 (0.01)	0.02 (0.01)	1e-18	2e-18
	Adaptive NW	0.09 (0.01)	0.02 (0.00)	1e-18	1e-17

(continued)

Table 5.2
(Continued)

		GMRES(80)	GMRES(40)	GMRES(80)	GMRES(40)
		Time (s)		Backward error	
nv2	Uniform	2.74 (1.17)	1.20 (0.60)	5e-17	7e-17
	Adaptive CW	1.81 (0.25)	0.73 (0.14)	4e-17	5e-17
	Adaptive NW	1.70 (0.10)	0.68 (0.06)	6e-17	2e-17
power9	Uniform	0.21 (0.04)	0.07 (0.01)	5e-19	6e-19
	Adaptive CW	0.17 (0.01)	0.06 (0.01)	3e-19	9e-20
	Adaptive NW	0.17 (0.01)	0.06 (0.00)	9e-20	6e-19
radiation	Uniform	0.40 (0.15)	0.16 (0.08)	2e-13	2e-13
	Adaptive CW	0.27 (0.03)	0.11 (0.02)	1e-13	6e-13
	Adaptive NW	0.27 (0.02)	0.10 (0.02)	1e-13	2e-13
ss	Uniform	42.98 (11.52)	31.58 (11.65)	2e-10	2e-03
	Adaptive CW	40.84 (9.44)	29.42 (9.62)	2e-10	2e-03
	Adaptive NW	41.35 (9.33)	29.29 (9.39)	9e-11	2e-03
stokes	Uniform	569.80 (169.68)	435.12 (172.14)	2e-05	3e-05
	Adaptive CW	536.18 (133.60)	399.76 (136.11)	3e-05	4e-05
	Adaptive NW	527.09 (122.99)	390.46 (126.10)	3e-05	3e-05
test1	Uniform	0.58 (0.15)	0.23 (0.07)	1e-14	1e-14
	Adaptive CW	0.47 (0.05)	0.18 (0.03)	3e-14	8e-14
	Adaptive NW	0.47 (0.05)	0.18 (0.03)	9e-15	7e-14
vas_stokes_1M	Uniform	36.02 (16.08)	29.03 (16.36)	5e-04	6e-04
	Adaptive CW	33.86 (13.96)	26.74 (14.13)	5e-04	6e-04
	Adaptive NW	33.34 (13.62)	26.32 (13.83)	6e-04	5e-04
vas_stokes_2M	Uniform	70.88 (28.22)	55.27 (28.64)	1e-04	9e-05
	Adaptive CW	63.33 (21.67)	48.64 (22.14)	1e-04	9e-05
	Adaptive NW	61.75 (20.48)	46.93 (20.64)	9e-05	9e-05
vas_stokes_4M	Uniform	159.89 (51.20)	118.03 (51.91)	9e-05	9e-05
	Adaptive CW	152.22 (44.27)	111.38 (45.08)	9e-05	9e-05
	Adaptive NW	151.02 (41.94)	108.61 (42.49)	8e-05	9e-05
Aghora_DGO3	Uniform	6.64 (3.72)	5.56 (3.76)	2e-03	2e-03
	Adaptive CW	6.43 (3.58)	5.34 (3.63)	3e-03	2e-03
	Adaptive NW	5.81 (3.14)	4.76 (3.15)	2e-03	2e-03
Aghora_DGO4	Uniform	15.39 (10.30)	13.40 (10.46)	1e-03	7e-04
	Adaptive CW	14.13 (9.30)	12.33 (9.45)	1e-03	7e-04
	Adaptive NW	12.95 (8.35)	11.26 (8.46)	1e-03	6e-04
Aghora_DGO5	Uniform	30.61 (23.27)	28.12 (23.58)	3e-03	2e-03
	Adaptive CW	27.46 (20.06)	24.89 (20.41)	3e-03	2e-03
	Adaptive NW	18.29 (11.08)	22.09 (17.68)	1e-04	2e-03

always the same depending on the SpMV variant that is used: in particular, the use of adaptive precision with NW criteria often prevents BiCGstab from converging, whereas the more robust GMRES solver can converge, and can sometimes do so faster than using the more expensive CW criteria.

Overall, this range of experiments shows that significant time reductions can be obtained by using an adaptive precision SpMV. In some cases, the speedup with respect to the uniform precision variant is huge because of the unexpected behavior observed in Figure 5.3, in which the adaptive precision NW variant actually converges in much less iterations than the uniform precision one (in addition to Geo_1438, this also happens, for example, for Emilia_923). Not counting these special cases, we still obtain significant speedups for many of the other matrices, especially those in Table 5.2.

Finally, we mention that the use of adaptive precision SpMV will lead to even larger speedups when the SpMV cost relative to the total increases. This is the case when the cost of the orthonormalization is reduced. Various strategies have recently been proposed in this direction, such as using low precision [2] or using faster orthonormalization algorithms, for example based on randomized methods [7] techniques, such as using low precision or faster orthonormalization algorithms. Conversely, the relative cost of the SpMV may increase when part of a preconditioner, for example in the case of polynomial preconditioners (which require multiple SpMVs per iteration) or SPAI preconditioners (which require SpMVs with a matrix M that approximates A^{-1}).

6. 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.

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 Krylov solvers for the solution of sparse linear systems. We have shown that the convergence speed of the solvers 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 solver 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 Krylov solvers 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 flexible GMRES. 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.

Acknowledgments. We thank our industrial partners for providing some of the test matrices used in this paper.

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), 51, https://doi.org/10.1145/3371275.

[2] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás, *Compressed basis gmres on high-performance graphics processing units*, Int. J. High Perform. Comput. Appl., 37 (2022), pp. 82–100.

[3] 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, 2021, http://eprints.maths.manchester.ac.uk/2852/, revised April 2022.

[4] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé, *Combining Sparse Approximate Factorizations with Mixed Precision Iterative Refinement*, MIMS EPrint 2022.2, Manchester Institute for Mathematical Sciences, The University of Manchester, Manchester, UK, 2022, http://eprints.maths.manchester.ac.uk/2845/.

[5] 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., 43 (2023), pp. 2198–2227, https://doi.org/10.1093/imanum/drac037.

[6] 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 Comput. Pract. Exper., 31 (2019), e4460, https://doi.org/10.1002/cpe.4460.

[7] O. Balabanov and L. Grigori, *Randomized Gram–Schmidt process with application to GMRES*, SIAM J. Sci. Comput., 44 (2022), pp. A1450–A1474, https://doi.org/10.1137/20M138870X.

[8] 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.

[9] 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.

[10] 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.

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

[12] J. Diffenderfer, D. Osei-Kuffuor, and H. Menon, *QDOT: Quantized Dot Product Kernel for Approximate High-Performance Computing*, preprint, https://arxiv.org/abs/2105.00115, 2021.

[13] 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), 14, https://doi.org/10.1145/3441850.

[14] 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, 53 (2023), pp. 81–98.

[15] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, 2002, https://doi.org/10.1137/1.9780898718027.

[16] 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.

[17] 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.

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

[19] C.-P. Jeannerod and S. M. Rump, *Improved error bounds for inner products in floating-point arithmetic*, SIAM J.Matrix Anal. Appl., 34 (2013), pp. 338–344, https://doi.org/10.1137/120894488.

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

[21] 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, 2020, pp. 51–66, https://doi.org/10.1007/978-3-030-63393-6_4.

[22] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam, *Experimental Evaluation of Multiprecision Strategies for GMRES on GPUs*, preprint, https://arxiv.org/abs/2105.07544, 2021.

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

[24] 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.

[25] 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.

[26] 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, Cham, 2015, pp. 315–335.

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

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

[29] V. Simoncini and D. B. Szyld, *Theory of inexact krylov subspace methods and applications to scientific computing*, SIAM J. Sci. Comput., 25 (2003), pp. 454–477, https://doi.org/10.1137/S1064827502406415.