Adaptive precision matrix-vector product

Stef Graillat, Fabienne Jézéquel, Théo Mary, Roméo Molina

ADAPTIVE PRECISION SPARSE MATRIX–VECTOR PRODUCT*

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

1. Introduction. Motivated by the growing availability of lower precision arithmetics, mixed precision algorithms are being developed for a wide range of numerical computations [12]. 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. [3], [7] 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. [2], which partitions a low-rank matrix into several low-rank components of decreasing norm and stores each of them in a correspondingly decreasing precision. In the most extreme case, adaptive precision algorithms choose a different precision for each scalar variable of the computation: for example, in a matrix computation, each individual matrix element may have its own independent level of precision. This is for example the case of the sparse matrix–vector product developed by Ahmad et al. [1], in which elements in the range $[-1, 1]$ are switched to single precision while the other elements are kept in double precision. Similarly, Diffenderfer et al. [6] have proposed 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 [12, 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, and obtain speedups of up to an order of magnitude on a multicore computer.

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 will use the standard model of floating-point arithmetic [9, sec. 2.2]

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

where u is the unit roundoff of the precision used.

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

*Version of December 6, 2021.

†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)

computed $\widehat{y}_i$ satisfies

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

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 [13] to obtain more refined bounds, where constants of the form $\gamma_n = nu/(1-nu)$ can be replaced simply by nu. The analysis of [13] assumes the use of rounding to nearest, but it was later shown in [14, 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 [10, 11, 4]. 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 nonzero elements 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 [9, Thm. 7.3], [15] and Rigal–Gaches [9, Thm. 7.1], [16] theorems, we have the following formulas for the componentwise backward error

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

and for the normwise backward error

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

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

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

Moreover, using (2.2), we obtain the bound

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

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 $u_1 > u_2 > \ldots > u_q$. 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_q.

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 $u_1 > \ldots > u_q$.

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_q
end for

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

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

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 $\overline{y}_i = \sum_{k=1}^{q} \widehat{y}_i^{(k)}$ the exact sum of the $\widehat{y}_i^{(k)}$, we have

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

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

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

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

where the conversion of $\widehat{y}_i^{(k)}$ back to precision u_q does not introduce any error since $u_q \leq u_k$ 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

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

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

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

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

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 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 $\epsilon \ge u_q$ 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

$$P_{ik} = \begin{cases} \left(0,\ \epsilon|a_i|^T|x|/u_1\right] & \text{for } k=1, \\ \left(\epsilon|a_i|^T|x|/u_{k-1},\ \epsilon|a_i|^T|x|/u_k\right] & \text{for } k=2\colon q-1, \\ \left(\epsilon|a_i|^T|x|/u_{q-1},\ +\infty\right] & \text{for } k=q, \end{cases} \tag{3.9}$$

which form a partition of $\mathbb{R}^+$, 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}:

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

This construction yields $\alpha_{ik} \le p_{ik}\epsilon/u_k$ and therefore, by (3.7),

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

with

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

We note that we have not taken into account any rounding error occuring in the computation of the intervals P_{ik}, which we assume is 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

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

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 $\|A\|$:

$$P_{ik} = \begin{cases} \big(0,\ \epsilon\|A\|/u_1\big] & \text{for } k = 1, \\ \big(\epsilon\|A\|/u_{k-1},\ \epsilon\|A\|/u_k\big] & \text{for } k = 2\colon q-1, \\ \big(\epsilon\|A\|/u_{q-1},\ +\infty\big] & \text{for } k = q, \end{cases} \tag{3.17}$$

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 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*

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

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

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

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

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

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" $u_1 = 1$, since replacing an element by zero introduces a relative perturbation equal to 1. Thus, taking $u_1 = 1$ 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 $A = e = [1, \dots, 1]$.*

3.2. 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. Our code uses up to seven different precisions: the IEEE fp64 and fp32 formats, the bfloat16 format, and four custom formats using 56, 48, 40, and 24 bits, which we will refer to as $fpxx$, with xx 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 fp16. In addition to these seven precision formats, we also drop the matrix elements that are sufficiently small, as explained in Remark 3.2.

For the cast from fp64 to fp32 we use the Fortran `REAL` function, whereas for casting to the other custom formats (including bfloat16), we use our own implementation that relies on the Fortran `MVBITS` subroutine. 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.

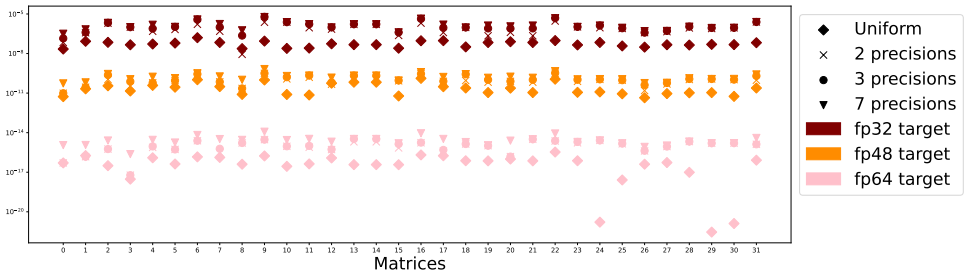
The matrices used in these experiments come from the SuiteSparse collection [5] and from our industrial partners (see Table 3.1). Clearly, by its very design, the potential of the adaptive precision algorithm completely depends on the matrix values: there must be sufficient variations in their magnitudes. 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 a minimum potential, listed in Table 3.1. As for the vector x, we set it to $e = [1, \ldots, 1]^T$ throughout the experiments. The role of the vector x is discussed in section 3.3.

Table 3.1: List of matrices used in our experiments.

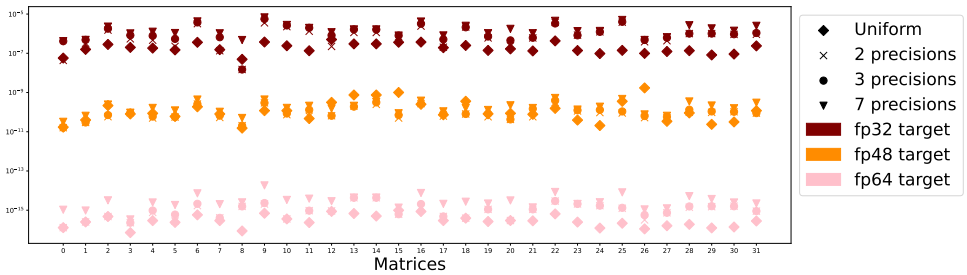
Number	Matrix	n	nnz
0	Transport	1.6e+06	2.4e+07
1	ss1	2.1e+05	8.5e+05
2	Serena	1.4e+06	3.3e+07
3	Emilia_923	9.2e+05	2.1e+07
4	Hook_1498	1.5e+06	3.1e+07
5	Geo_1438	1.4e+06	3.2e+07
6	vas_stokes_1M	1.1e+06	3.5e+07
7	ML_Laplace	3.8e+05	2.8e+07
8	ss	1.7e+06	3.5e+07
9	vas_stokes_2M	2.1e+06	6.5e+07
10	Fault_639	6.4e+05	1.5e+07
11	PFlow_742	7.4e+05	1.9e+07
12	CoupCons3D	4.2e+05	2.2e+07
13	Long_Coup_dt6	1.5e+06	4.4e+07
14	Long_Coup_dt0	1.5e+06	4.4e+07
15	StocF-1465	1.5e+06	1.1e+07
16	vas_stokes_4M	4.4e+06	1.3e+08
17	ML_Geer	1.5e+06	1.1e+08
18	Bump_2911	2.9e+06	6.5e+07
19	Cube_Coup_dt6	2.2e+06	6.5e+07
20	Flan_1565	1.6e+06	5.9e+07
21	Cube_Coup_dt0	2.2e+06	6.5e+07
22	stokes	1.1e+07	3.5e+08
23	nv2	1.5e+06	5.3e+07
24	test1	3.9e+05	1.3e+07
25	radiation	2.2e+05	7.6e+06
26	power9	1.6e+05	2.5e+06
27	imagesensor	1.2e+05	1.9e+06
28	dgreen	1.2e+06	3.8e+07
29	mosfet2	4.7e+04	1.5e+06
30	nv1	7.5e+04	2.4e+06
31	Queen_4147	4.1e+06	1.7e+08

We begin in Figure 3.1 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 3.1a, the normwise backward error for the algorithm with the normwise bucket criteria (3.16)–(3.17), and, in Figure 3.1b, the componentwise backward error for the algorithm with the componentwise bucket criteria (3.9)–(3.10). We use three different target accuracies,

that is, three values of ϵ, 2^{-53}, 2^{-37}, and 2^{-24}, which correspond to the unit roundoffs of fp64, fp48, and fp32, respectively, and compare its backward error to the one obtained by the uniform precision algorithm in the corresponding target format (fp64, fp48, or fp32). 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)).

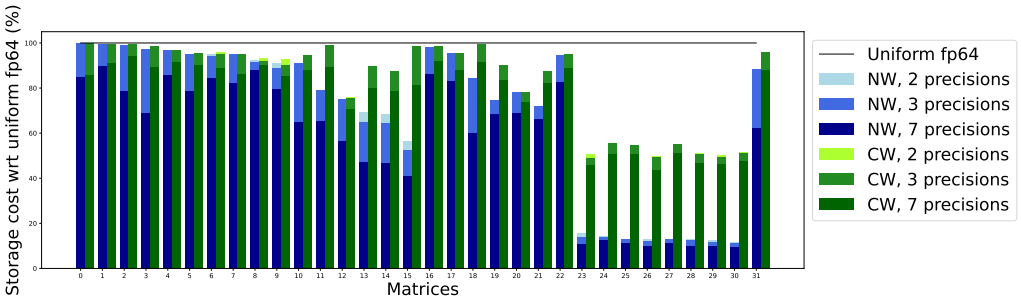
Fig. 3.1: Backward error for the adaptive precision Algorithm 3.1 with different target accuracies ϵ and different number of precisions used, compared with the uniform precision Algorithm 2.1 in the corresponding precision (fp32, fp48, or fp64).

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.

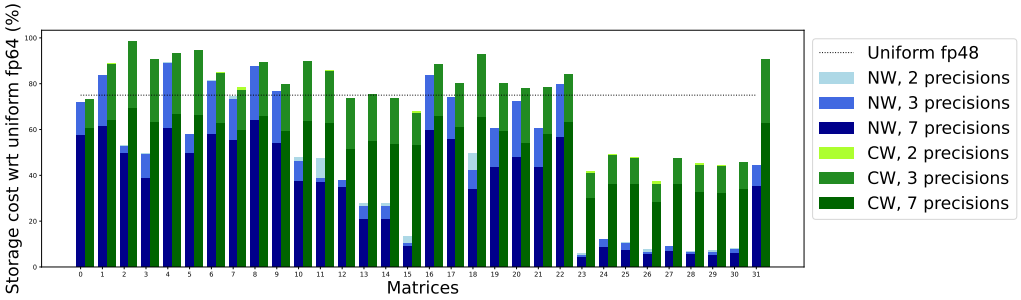
Figure 3.2 plots the storage cost of the adaptive precision algorithm as a percentage of the uniform precision fp64 algorithm. As for Figure 3.1, several configurations of the adaptive precision algorithm are tested, depending on the accuracy target (fp64, fp48, or fp32), the number of precisions used (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 11 and 12 with an fp32 accuracy target (Figure 3.2c) is 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 3.2a) to fp32 (Figure 3.2c) 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 7, 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.

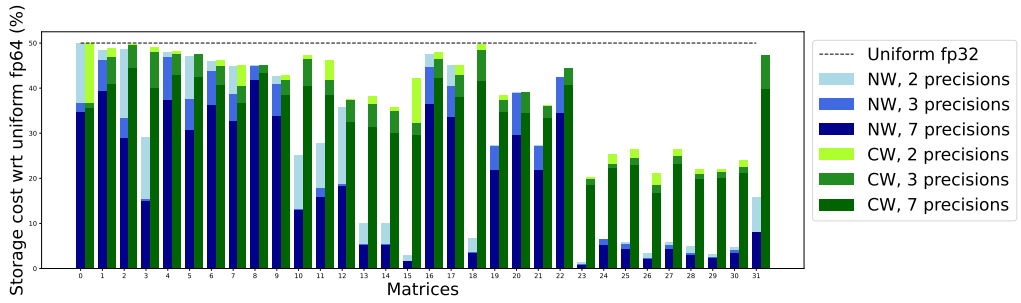
In any case, the storage gains are overall significant in all configurations and for several matrices, with reductions of up to a factor xx in the best case.



(a) fp64 accuracy target



(b) fp48 accuracy target



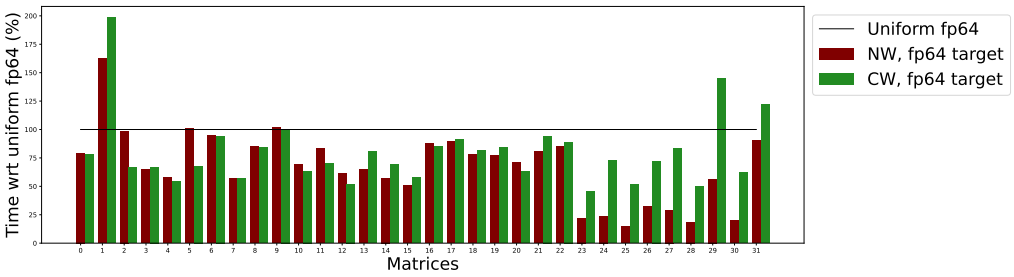
(c) fp32 accuracy target

Fig. 3.2: 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.

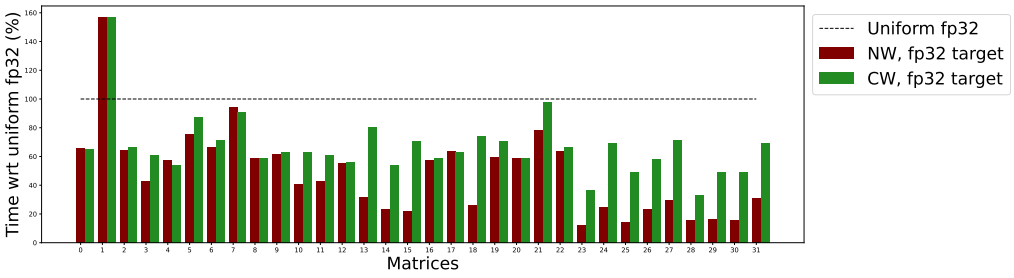
Finally, we measure the execution time of the algorithms. Since SpMV is memory bound, in principle we can hope the time gains to roughly follow the storage gains, even though the execution time depends on several other factors such as the overhead cost of the cast operations and the latency costs. In 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 the natively supported fp64 and

fp32 formats (that is, 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. Our cast implementation is however not optimized and there has been recent work on efficient implementations, such as the memory accessor from [8], which suggests that the three- and seven-precision versions could meet their potential with a more optimized implementation. We however leave these ideas for future work and focus here on the results obtained with the two-precision version.

Figure 3.3 reports the execution time of the adaptive precision SpMV for fp64 and fp32 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 xx 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. Alternatively, it could also be explained by the dropping of sufficiently small elements, which not only reduces the bandwidth costs but also the latency ones, since the dropped elements are not read at all.



(a) fp64 accuracy target



(b) fp32 accuracy target

Fig. 3.3: Execution time of the adaptive precision SpMV for fp64 and fp32 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 fp48 accuracy target in Figure 3.4. The figure also plots the time for the fp64 and fp32 targets (already presented in Figure 3.3) as a point of comparison. Figure 3.4 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 used.

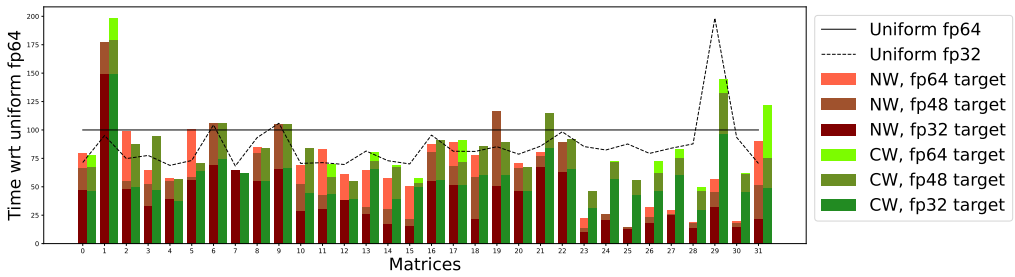


Fig. 3.4: Execution time of the adaptive precision SpMV, as a percentage of the execution time of the uniform precision fp64 SpMV, depending on the target accuracy and on whether the normwise ("NW") or componentwise ("CW") criteria is considered.

3.3. Limitation. 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:

$$P_{ik} = \begin{cases} \left(0,\ \epsilon|a_i|^T e/u_1\right] & \text{for } k = 1, \\ \left(\epsilon|a_i|^T e/u_{k-1},\ \epsilon|a_i|^T e/u_k\right] & \text{for } k = 2\colon q-1, \\ \left(\epsilon|a_i|^T e/u_{q-1},\ +\infty\right] & \text{for } k = q, \end{cases} \tag{3.20}$$

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 the 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) will 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.

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, 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*, 2021. HAL EPrint hal-03251738, June 2021.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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).

[9] 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.

[10] 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.

[11] 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.

[12] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, MIMS EPrint 2021.20, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2021, http://eprints.maths.manchester.ac.uk/2841/. To appear in Acta Numerica.

[13] 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.

[14] 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.

[15] 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.

[16] 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.