Reduced-Precision and Reduced-Exponent Formats for Accelerating Adaptive Precision Sparse Matrix-Vector Product

Stef Graillat, Fabienne Jézéquel, Theo Mary, Roméo Molina, Daichi Mukunoki

Reduced-Precision and Reduced-Exponent Formats for Accelerating Adaptive Precision Sparse Matrix–Vector Product

Stef Graillat[1][0000−0001−8954−2276], Fabienne Jézéquel[1,2][0000−0002−8782−7566], Theo Mary[1][0000−0001−9949−4634], Roméo Molina[1,3], and Daichi Mukunoki**[0000−0002−0051−6811]

[1] Sorbonne Université, CNRS, LIP6, Paris, France
[2] Université Paris-Panthéon-Assas, Paris, France
[3] Université Paris-Saclay, Orsay, France

Abstract. Mixed precision algorithms aim at taking advantage of the performance of low precisions while maintaining the accuracy of high precision. In particular adaptive precision algorithms dynamically adapt at runtime the precisions used for different variables or operations. For example Graillat et al (2023) have proposed an adaptive precision sparse matrix–vector product (SpMV) which stores the matrix elements in a precision inversely proportional to their magnitude. In theory, this algorithm can therefore make use of a large number of different precisions, but the practical results previously obtained only achieved high performance using natively supported double and single precisions. In this work we combine this algorithm with an efficient memory accessor for custom reduced precision formats (Mukunoki et al. 2016). This allows us to experiment with a large set of different precision formats with fine variations of the number of bits dedicated to the significand. Moreover we also explore the possibility to reduce the number of bits dedicated to the exponent using the fact that the elements that share the same precision format are of similar magnitude. We experimentally evaluate the performance of using four or seven different custom formats using reduced precision and possibly reduced exponent, and demonstrate their effectiveness compared with the existing version only using double and single precisions.

Keywords: sparse matrix–vector product (SpMV), mixed precision, adaptive precision, reduced-precision, reduced-exponent

** part of this work has been done when this author was working at RIKEN Center for Computational Science, Kobe, Japan

1 Introduction

The use of low precision floating-point formats in scientific computations is becoming more and more common due to the storage and performance gains that they offer. To maintain a rigorous control over the accuracy of the result, mixed precision algorithms combine various precision formats to find the desired tradeoff between performance and accuracy [1]. *Adaptive precision* algorithms [1, sect. 14], a subclass of mixed precision algorithms, have recently attracted interest due to their ability to dynamically detect at runtime opportunities for reducing the precision based on the data at hand. Adaptive precision algorithms have for example been developed for low-rank approximations [2], block Jacobi preconditioners [3, 4], block/tile low-rank factorizations [2, 5], and sparse matrix-vector product (SpMV) [6, 7].

In this work, we are particularly interested in the adaptive precision SpMV proposed by Graillat et al. [6]. SpMV is a key computational kernel in many applications, such as the solution of sparse linear systems with Krylov methods. Accelerating the SpMV while preserving control over its accuracy is therefore an important goal with a wide range of potential applications.

The approach of Graillat et al. [6] proposes to accelerate SpMV by storing some of the nonzero elements of the matrix in reduced precision. Indeed, a theoretical error analysis demonstrates that the accuracy can be rigorously controlled by switching the elements to a precision with a unit roundoff inversely proportional to their magnitude: that is, smaller elements can be stored in lower precisions. This leads to storage reduction which can translate to a corresponding time reduction because SpMV is a memory-bound operation.

In addition to their error analysis, Graillat et al. [6] present a practical implementation of their algorithm. Their implementation achieves significant performance gains compared with SpMV in uniform precision, at a comparable accuracy. However, their performance results are only satisfactory when using precisions with native hardware support, that is, in their case, the standard IEEE FP64 (double) and FP32 (single) precisions.

In this article, we are motivated by the fact that the adaptive precision SpMV can in principle make use of any number of precision levels and, in fact, achieves larger storage reductions when more precisions are available, since this allows for a fine tuning of the precision of each element. We are interested in the potential of emerging technologies for reduced-precision memory accessors [8–10], which allow for efficiently accessing data stored in reduced precision formats. We focus in particular on the work of Mukunoki and Imamura [8], who propose a *custom* reduced precision accessor, thus allowing for many different precision formats. We develop an adaptive precision SpMV algorithm that relies on this memory accessor, and that can use up to seven different precision formats with fine variations of the number of bits dedicated to the significand. Moreover we also explore the possibility to reduce the number of bits dedicated to the exponent using the fact that the elements that share the same precision format are of similar magnitude. We provide numerical experiments on a multicore CPU architecture and with a range of real-life matrices. We evaluate the performance

of adaptive precision SpMV with varying numbers of precision formats, using reduced precision and possibly reduced exponent, and demonstrate the effectiveness of the custom memory accessor compared with the existing version that only uses natively supported double and single precisions.

2 Methods

2.1 Adaptive precision SpMV

The adaptive precision SpMV proposed by Graillat et al. [6] decomposes the input matrix A into several matrices A_k:

$$A = \sum_{k=1}^{q} A_k,$$

where each A_k is stored in a different precision format with unit roundoff u_k, and the sparsity patterns of the A_k are all mutually disjoint (that is, each nonzero element of A is assigned to exactly one matrix A_k). The q precision levels satisfy $u_1 < u_2 < \cdots < u_q$.

The SpMV $y = Ax$ is then computed as the sum of the partial SpMVs:

$$y = Ax = \sum_{k=1}^{q} A_k x.$$

The accuracy of the computed vector $\widehat{y}$ can be controlled by suitably building the decomposition. Specifically, the error analysis carried out in [6] proves that, given a prescribed accuracy $\epsilon \geq u_1$, the computed $\widehat{y}$ satisfies the normwise backward error bound

$$\widehat{y} = (A + \Delta A)x, \quad \|\Delta A\| \leq c\epsilon\|A\|,$$

where c is a modest constant, under the condition that all the nonzero elements a_{ij} of A_k satisfy the criterion

$$a_{ij} \in A_k \Leftrightarrow |a_{ij}| \in \Big(\frac{\epsilon}{u_{k+1}}\|A\|, \ \frac{\epsilon}{u_k}\|A\|\Big]. \tag{1}$$

This criterion shows that the precision u_k used to store each nonzero element should be chosen to be inversely proportional to the magnitude of the element. One special case is when $|a_{ij}| \leq \epsilon\|A\|$: in this case, the element can be *dropped*, that is, replaced by zero (this can be interpreted as using a "unit roundoff" $u_q = 1$, see also [6, Remark 3.2]). Moreover, we mention that [6] also proposes an alternative criterion which bounds the componentwise backward error, instead of the normwise one.

The error analysis in [6] also accounts for the possibility of performing the partial SpMVs $A_k x$ in precision u_k, but since the SpMV is a memory-bound operation, this does not bring any significant performance improvement. We will therefore only use the reduced precision formats for storage, while keeping the arithmetic operations in double precision (which corresponds to the highest precision u_1).

2.2 Custom reduced-precision formats

Mukunoki and Imamura [8] have proposed a reduced-precision memory accessor, called RPFP, that allows for representing custom floating-point numbers with a reduced significand. This is achieved by truncating the IEEE FP64 (double) or FP32 (single) formats, as shown in Table 1: the RP56, RP48, and RP40 formats are truncated versions of FP64, whereas the RP24 and RP16 formats are truncated versions of FP32. Note that RP16 is equivalent to the bfloat16 format, although in our case we do not have native support for bfloat16 operations on our target hardware.

Table 1. List of IEEE formats and RPFP's reduced-precision formats

Format	Numbers of bits			Unit roundoff
	Sign	Exponent	Significand	
FP64	1	11	52+1	$2^{-53} \approx 1 \times 10^{-16}$
RP56	1	11	44+1	$2^{-45} \approx 3 \times 10^{-14}$
RP48	1	11	36+1	$2^{-37} \approx 7 \times 10^{-12}$
RP40	1	11	28+1	$2^{-29} \approx 2 \times 10^{-9}$
FP32	1	8	23+1	$2^{-24} \approx 6 \times 10^{-8}$
RP24	1	8	15+1	$2^{-16} \approx 2 \times 10^{-5}$
RP16	1	8	7+1	$2^{-8} \approx 4 \times 10^{-3}$

This RPFP accessor is implemented in the C/C++ language and relies internally on a structure with multiple words composed of one, two, or four bytes: for example RP40 is represented using a 32-bit integer and an 8-bit one. When dealing with an array of RPFP numbers, each integer composing the RPFP format is allocated separately from the other integers (so-called structure-of-arrays layout). The decoding of an RP40 number to an FP64 one is illustrated in Figure 1. It is a relatively lightweight operation that consists in copying the 8-bit and 32-bit integers into 64-bit integers, suitably realigning them with a bit shift, and combining them with a binary `or` operation.

The RPFP accessor has been shown to accelerate various types of memory-bound operations, such as dot products, dense matrix–vector products, and SpMVs. In particular, the recent work [10] focuses on the SpMV and shows that the performance is often proportional to the storage and thus to the number of bits. Naturally, in a uniform precision setting where the same RPFP format is used for all the elements, the accuracy is also proportional to the number of bits. In the following, we investigate the use of this kind of accessor in an adaptive precision setting which preserves a controlled accuracy.

2.3 Reduced-precision formats for adaptive precision SpMV

The adaptive precision SpMV is particularly amenable to the use of custom precision formats, for two reasons. First, the reduced precisions are only used as

```
union union64 {
        uint64_t i;
        double f;
};
__inline__ double
RpArrayToFp (rpfp64in40barray sa, size_t i){
        union union64 u64;
        uint64_t i64h, i64l;
        i64h = (uint64_t)sa.i32[i];
        i64h = i64h << 32;
        i64l = (uint64_t)sa.i8[i];
        i64l = i64l << 24;
        u64.i = i64h | i64l;
        return u64.f;
}
```

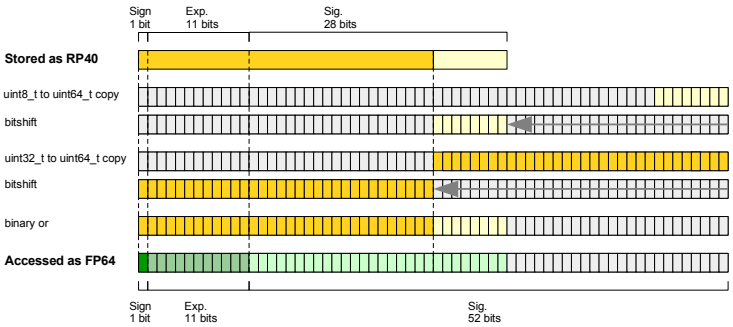

Fig. 1. Conversion from RP40 to FP64

storage formats, and hence do not require native support for arithmetic operations. Second, because the precisions should be set to be inversely proportional to the magnitude of the elements, we can in principle exploit a continuous level of precisions: the finer we can tune the precision level, the higher the storage reduction.

The implementation presented in [6] achieves significant performance gains, but unfortunately only when using the natively supported FP64 and FP32 precisions available in the target architecture. Experiments with custom precision formats are presented, but these lead to a heavy performance penalty due to an unoptimized memory accessor implemented in Fortran.

The goal of this work is therefore to implement the adaptive precision SpMV algorithm from [6] with the much more efficient RPFP accessor from [8], and to evaluate to what extent custom precision formats can improve the performance.

To do so, we ported the Fortran implementation of adaptive precision SpMV from [6] to the C language and combined it with the RPFP implementation for CPUs from [8].

To represent the sparse matrices A and A_k we use the compressed sparse row (CSR) format with 32-bit row and column indices. For each matrix A_k, this leads to $n + nnz_k$ 32-bit values for the indices, and nnz_k RPFP values for the

nonzero elements, where n is the number of rows of A_k and nnz_k its number of nonzero elements.

The SpMV is parallelized with OpenMP using a static schedule: the rows of the matrix are distributed among the available threads.

Figure 2 presents an excerpt of the adaptive precision SpMV code with seven precision levels.

```
void ap_csrmv (int n, rpMultiCSR A, double* x, double* y) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double tmp = 0.;
        for (int k = A.ia16[i]; k < A.ia16[i+1]; k++) { // RP16
            float aij_r = RpArrayToFp(A.a16, k);
            tmp += aij_r * x[A.ja16[k]];
        }
        for (int k = A.ia24[i]; k < A.ia24[i+1]; k++) { // RP24
            float aij_r = RpArrayToFp(A.a24, k);
            tmp += (double)(aij_r * x[A.ja24[k]]);
        }
    ...
        for (int k = A.ia56[i]; k < A.ia56[i+1]; k++) { // RP56
            double aij = RpArrayToFp(A.a56, k);
            tmp += aij * x[A.ja56[k]];
        }
        for (int k = A.ia64[i]; k < A.ia64[i+1]; k++) { // FP64
            double aij = A.a64[k];
            tmp += aij * x[A.ja64[k]];
        }
        y[i] = tmp;
    }
}
```

Fig. 2. Adaptive precision SpMV with seven precision levels (excerpt). `RpArrayToFp` converts a reduced-precision format to the IEEE FP64 format.

2.4 Reduced-exponent formats for adaptive precision SpMV

While the RPFP accessor proposed in [8] only reduces the significand, it may also be beneficial to reduce the number of bits allocated to the exponent field in order to further reduce the storage.

This idea is particularly promising for the adaptive precision SpMV because the matrix elements stored in the same format are by design of similar magnitude. As shown in (1), all elements of A_k stored in a precision with unit roundoff u_k are in the interval $(\epsilon\|A\|/u_{k+1}, \epsilon\|A\|/u_k]$. Therefore, the dynamic range of the elements of A_k is u_{k+1}/u_k, which means that $\lceil \log_2(u_{k+1}/u_k) \rceil$ exponent values are sufficient to represent elements, and so

$$\left\lceil \log_2 \left\lceil \log_2 \frac{u_{k+1}}{u_k} \right\rceil \right\rceil$$

bits are sufficient for the exponent field. In particular, if we use all seven precision formats in Table 1, we have $u_{k+1}/u_k \leq 2^8$ and so we can reduce the number of bits dedicated to the exponent to $\log_2 \log_2 2^8 = 3$ bits. This represents a large storage reduction compared with the 11 or 8 bits used by the formats in Table 1.

Concretely, we represent the elements a_{ij} of A_k as

$$a_{ij} = \frac{\epsilon \|A\|}{u_{k+1}} \cdot \alpha_{ij}$$

where $\alpha_{ij} \in [1, 2^8)$ is represented with reduced precision *and* reduced exponent (RPRE). Note that we must change the interval in (1) to be closed on the left and open on the right, in order to exclude the right endpoint $\alpha_{ij} = 2^8$ which would require four bits. This leads us to define the RPRE formats listed in Table 2. The RPRE32, RPRE40, and RPRE48 formats have respectively the same unit roundoff as RP40, RP48, RP56 but three bits of exponent instead of eleven. The RPRE24, RPRE16, and RPRE8 formats are similar to the FP32, RP24 and RP16 formats, with only three bits of exponent instead of eight and slightly different unit roundoffs.

Table 2. List of RPRE formats used for each interval of values.

Interval ($\epsilon' = \epsilon\|A\|$)	Format	Numbers of bits			Unit roundoff
		Sign	Exponent	Significand	
$[\epsilon' 2^{45}, \|A\|]$	FP64	1	11	52+1	$2^{-53} \approx 1 \times 10^{-16}$
$[\epsilon' 2^{37}, \epsilon' 2^{45})$	RPRE48	1	3	44+1	$2^{-45} \approx 3 \times 10^{-14}$
$[\epsilon' 2^{29}, \epsilon' 2^{37})$	RPRE40	1	3	36+1	$2^{-37} \approx 7 \times 10^{-12}$
$[\epsilon' 2^{21}, \epsilon' 2^{29})$	RPRE32	1	3	28+1	$2^{-29} \approx 2 \times 10^{-9}$
$[\epsilon' 2^{13}, \epsilon' 2^{21})$	RPRE24	1	3	20+1	$2^{-21} \approx 5 \times 10^{-7}$
$[\epsilon' 2^{5}, \epsilon' 2^{13})$	RPRE16	1	3	12+1	$2^{-13} \approx 1 \times 10^{-4}$
$[\epsilon', \epsilon' 2^{5})$	RPRE8	1	3	4+1	$2^{-5} \approx 3 \times 10^{-2}$
$[0, \epsilon')$	dropping	0	0	0	$2^0 = 1$

In the same spirit, we can further reduce the storage by splitting the elements of each A_k into A_k^+ and A_k^- according to their sign. This allows us to drop the sign bit in the floating-point representation, which can instead be used for the significand. This leads to unsigned RPRE formats (RPREU), listed in Table 3, which can be applied to a slightly better interval of values (with a smaller unit roundoff). For example, RPREU8 can be applied to values up to $\epsilon' 2^6$, instead of $\epsilon' 2^5$ for RPRE8. The tradeoff is that we double the number of matrices A_k, which doubles the number of row index arrays of size n (the total size of the column index arrays remains equal to nnz the number of nonzero elements of A). Therefore the RPREU formats can be beneficial only when the matrix is not too sparse (sufficiently large nnz/n ratio).

In practice, the decoding of RPRE and RPREU formats is a little heavier than that of RP formats. To decode an RPRE number, as shown in Figure 3, we

Table 3. List of RPREU formats used for each interval of values.

Interval ($\epsilon' = \epsilon\|A\|$)	Format	Numbers of bits Sign	Exponent	Significand	Unit roundoff
$[\epsilon' 2^{46}, \|A\|]$	FP64	1	11	52+1	$2^{-53} \approx 1 \times 10^{-16}$
$[\epsilon' 2^{38}, \epsilon' 2^{46})$	RPREU48	0	3	45+1	$2^{-46} \approx 1 \times 10^{-14}$
$[\epsilon' 2^{30}, \epsilon' 2^{38})$	RPREU40	0	3	37+1	$2^{-38} \approx 4 \times 10^{-12}$
$[\epsilon' 2^{22}, \epsilon' 2^{30})$	RPREU32	0	3	29+1	$2^{-30} \approx 9 \times 10^{-10}$
$[\epsilon' 2^{14}, \epsilon' 2^{22})$	RPREU24	0	3	21+1	$2^{-22} \approx 2 \times 10^{-7}$
$[\epsilon' 2^{6}, \epsilon' 2^{14})$	RPREU16	0	3	13+1	$2^{-14} \approx 6 \times 10^{-5}$
$[\epsilon', \epsilon' 2^{6})$	RPREU8	0	3	5+1	$2^{-6} \approx 2 \times 10^{-2}$
$[0, \epsilon')$	dropping	0	0	0	$2^{0} = 1$

need to separate the bit of sign from the exponent, which requires extra binary `and` and `or` operations. We also need to reset the exponent to its actual value but this operation can be realised only once per bucket. Decoding an RPREU number does not require the sign bit separation as there is none.

```
1  __inline__ double
2  RPREArrayToFp (rpre40barray sa, size_t i){
3      union union64 u64;
4      uint64_t i64h, i64m, i64l;
5      i64h = (uint64_t) ((sa.i8[i] & 0x80) | 0x40);
6      i64h = i64h << 56 ;
7      i64m = (uint64_t) (sa.i8[i] & 0x7F);
8      i64m = i64m << (32+16);
9      i64l = (uint64_t) sa.i32[i];
10     i64l = i64l << 16;
11     u64.i = i64h | i64m | i64l;
12     return u64.f;
13 }
```

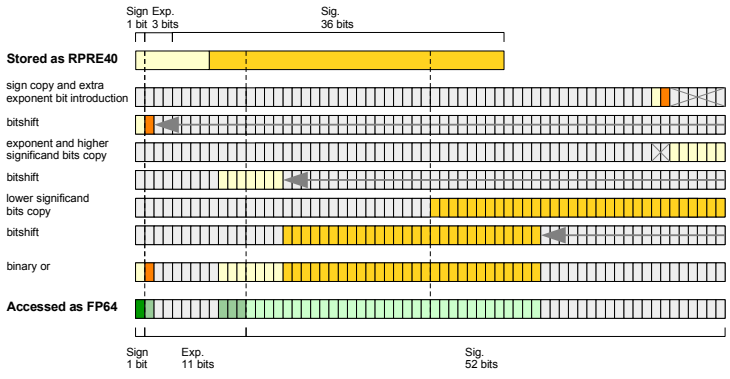

Fig. 3. Conversion from RPRE40 to FP64

Table 4. Test matrices (Sorted by nnz).

#	Matrix	n	nnz
0	vas_stokes_4M	4,382,246	131,577,616
1	Cube_Coup_dt0	2,164,760	127,206,144
2	Flan_1565	1,564,794	117,406,044
3	Long_Coup_dt6	1,470,152	87,088,992
4	bone010	986,703	71,666,325
5	vas_stokes_2M	2,146,677	65,129,037
6	Hook_1498	1,498,023	60,917,445
7	RM07R	381,689	37,464,962

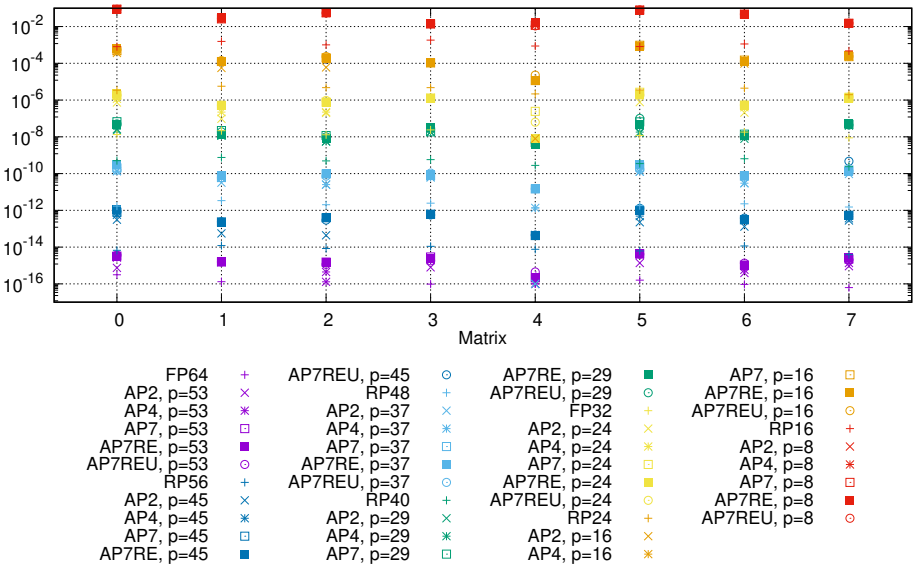

Fig. 4. Normwise backward error computed from the FP128 uniform precision SpMV

3 Evaluation

We perform the performance evaluation of our methods on one node of the Jean Zay supercomputer, equipped with two Intel Cascade Lake 6248 processors with 20 cores running at 2.5 GHz each, for a total of 40 cores. The experimental code was compiled using GCC 8.5.0 with `-O3 -march=native -fopenmp -lgomp` (1 thread/core). It was executed with `numactl -interleave=all`.

We collected eight matrices from the SuiteSparse Matrix Collection [11], listed in Table 4 (each matrix has size of $n \times n$ with nnz nonzero elements). The matrices are ordered by nnz in descending order. We do not exploit the potential symmetry of the matrices: symmetric matrices are expanded to unsymmetric ones before the execution. The vector x is set to $e = [1, \ldots, 1]^T$.

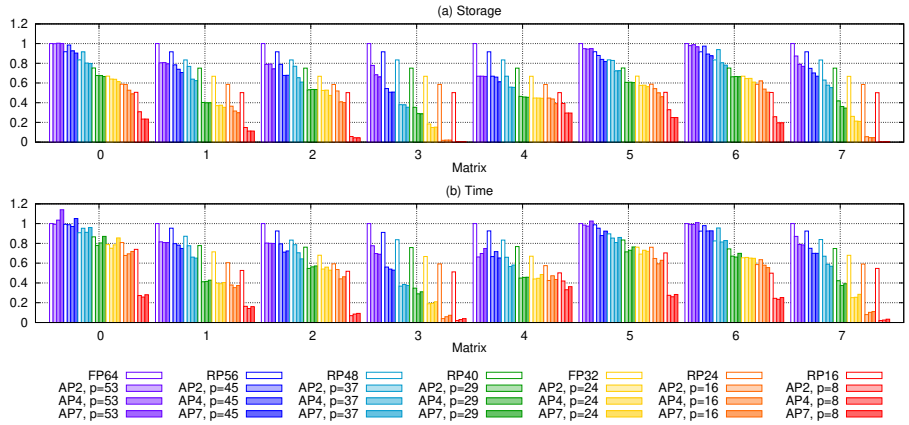

Fig. 5. Storage and time gains achieved by adaptive precision variants over uniform precision ones (normalized by the FP64 cost)

We have chosen to store the input and output vectors x and y in FP64 for all our experiments. We observed that this does not negatively affect the performance compared with storing them in FP32 and can even improve it in some cases due to the appearance of denormalized numbers. To further minimize the risk of incurring underflow, overflow, and subnormality, we also scale the matrix by its norm so that all elements are bounded by 1.

For each result, we report the shortest execution time out of 15 executions (the SpMV is executed five times in a single program, and the program is executed three times.).

To evaluate the potential offered by the introduction of RPFP formats in the adaptive precision SpMV, we compare the following configurations:

- **FPxx**: Uniform precision SpMV with FPxx (xx=32 or 64).
- **RPxx**: Uniform precision SpMV with RPxx (xx=16, 24, 40, 48, or 56).
- **AP2**: Adaptive precision SpMV with two precision levels: the natively supported IEEE FP64 and FP32, as well as dropping.
- **AP4**: Adaptive precision SpMV with four precision levels: FP64, RP48, FP32, and RP16, as well as dropping.
- **AP7**: Adaptive precision SpMV with all seven precision levels: FP64, RP56, RP48, RP40, FP32, RP24, and RP16, as well as dropping.

In addition, to evaluate the potential offered by the reduced-exponent formats, we also compare the above variants with the following configurations.

- **AP7RE**: Adaptive precision SpMV with seven precision levels: FP64, RPRE48, RPRE40, RPRE32, FP32, RPRE16, and RPRE8, as well as dropping.
- **AP7REU**: Adaptive precision SpMV with seven precision levels: FP64, RPREU48, RPREU40, RPREU32, FP32, RPREU16, and RPREU8, as well as dropping.

Moreover, we test the adaptive precision variants with various accuracy targets $\epsilon = 2^{-p}$, and compare them with the uniform precision variant of corresponding accuracy (for example, for $p = 29$ we compare with the RP40 variant).

We begin by checking the correctness of our code. Figure 4 presents the backward errors achieved by each configuration. The figure confirms that all adaptive precision variants achieve the prescribed accuracy of order ϵ.

3.1 Performance of adaptive precision SpMV with RPFP

Figure 5 presents the storage and time costs of each variant, normalized by the FP64 cost. The time cost closely follows the storage cost as expected. The first observation is that the adaptive precision variants are usually faster than the uniform precision variant at comparable accuracy, with very large speedups in some cases (up to 96%) that are explained by huge storage reductions (up to 99%) thanks to the use of dropping. The AP2 variant, which only uses natively supported precisions, performs well but we can see that the AP4 and AP7 variants can in many cases further improve the performance by exploiting custom precision formats. For example, the maximum storage gain of AP4 over AP2 is 24% and the maximum speedup is 21%. As for the difference between AP4 and AP7, it is less significant, and AP4 can be more efficient in some cases because of the increased weight of the indices storage. Still, using seven rather than just four precisions can bring an improvement up to 11% in the storage and a maximum speedup of 9%.

3.2 Performance of adaptive precision SpMV with RPRE and RPREU

In order to provide performance evaluations of the RPRE and RPREU variants, we have chosen not to use the RPRE24 format but to use FP32 instead, which we have observed to be more efficient since the latter is a natively supported format. We have also experimented using both FP32 and RPRE24 and obtained similar, although slightly less efficient results than when only using FP32 (which can be explained by the short length of the intervals associated with these two formats).

Figure 6 compares the storage and time costs of the AP7RE and AP7REU variants with those of the previously analyzed AP7 and uniform precision variants. The figure shows that the use of reduced-exponent formats can improve the performance in some cases and on the contrary degrade it in other cases. Nevertheless, these variants can achieve up to 16% storage reduction and a maximum speedup of 13%. The heavier decoding of these reduced-exponent formats presumably explains why their use does not always improve the performance.

Finally, we plot in Figure 7 the distribution of the precision formats used for each nonzero element of the matrix. The figure illustrates that having a greater number of reduced-precision formats (AP7 instead of AP4 or AP2) allows for a finer tuning of the precisions assigned to each element. Moreover, the figure also shows that using reduced-exponent formats (AP7RE or AP7REU) allows for

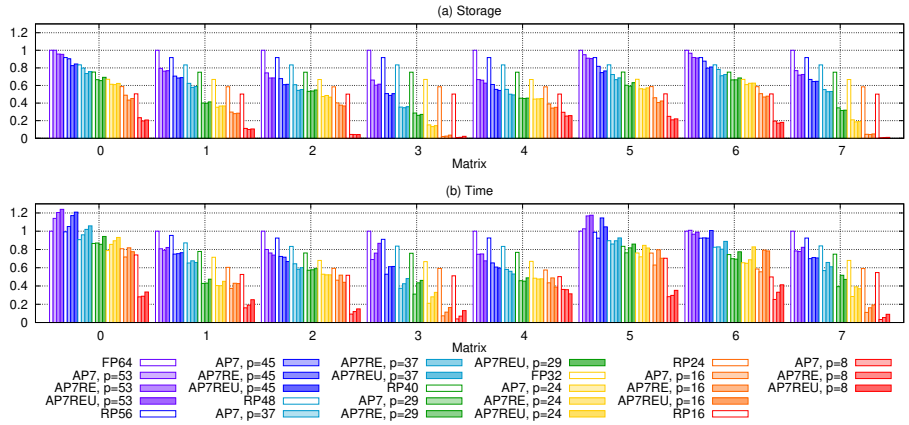

Fig. 6. Storage and time gains achieved by AP7RE and AP7REU variants over the uniform precision and AP7 ones (normalized by the FP64 cost)

reusing the exponent (and possibly sign) bits for the significand, which increases the proportion of elements stored in reduced precisions.

4 Conclusion

We have demonstrated the potential of using custom floating-point formats for accelerating the adaptive precision sparse matrix–vector product algorithm of [6]. We have shown that using up to seven different reduced-precision formats from [8] can lead to speedups of up to 96%. Moreover, we have developed new reduced-exponent formats that can improve performance even more with further speedups of up to 13%. Since the adaptive precision algorithm allows for rigorously controlling the loss of accuracy, all these performance gains are achieved at an accuracy comparable with that of obtained with uniform precision algorithms.

A promising perspective for further improvements is to use different sparse matrix formats with a reduced relative weight of the indices, which would significantly increase the potential of adaptive precision. This is in particular the case of diagonal or block sparse formats.

Acknowledgments

This research was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #20KK0259, the InterFLOP (ANR-20-CE46-0009) and MixHPC (ANR-23-CE46-0005-01) projects of the French National Agency for Research (ANR) and the interdisciplinary CNRS project CASSIDI.
This work was granted access to the HPC resources of IDRIS under the allocation 2023-AD010614925 made by GENCI.

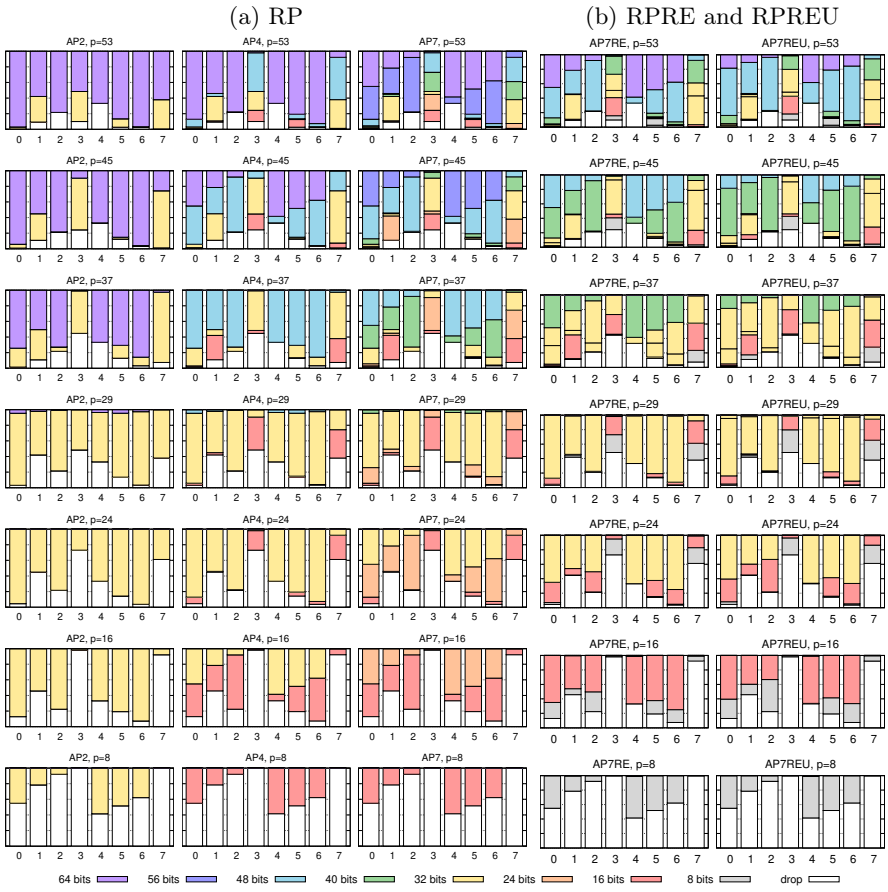

Fig. 7. Distribution of the precision formats used for each nonzero element. Each bar on the x-axis corresponds to a different matrix and the y-axis indicates the percentage of nonzero elements stored in each format.

References

1. N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, pp. 347–414, May 2022.
2. P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary, "Mixed precision low rank approximations and their application to block low rank lu factorization," *IMA J. Numer. Anal.*, 2022.
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.*, vol. 31, no. 6, p. e4460, 2019.
4. 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. Softw.*, vol. 47, no. 2, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3441850
5. S. Abdulah, Q. Cao, Y. Pei, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief, and Y. Sun, "Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with PaRSEC," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 964–976, Apr. 2022.
6. S. Graillat, F. Jézéquel, T. Mary, and R. Molina, "Adaptive precision sparse matrix–vector product and its application to krylov solvers," *SIAM J. Sci. Comput.*, vol. 46, no. 1, pp. C30–C56, 2024.
7. K. Ahmad, H. Sundar, and M. Hall, "Data-driven mixed precision sparse matrix vector multiplication for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371275
8. D. Mukunoki and T. Imamura, "Reduced-Precision Floating-Point Formats on GPUs for High Performance and Energy Efficient Computation," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 144–145.
9. T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí, "Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision blas," *Software: Practice and Experience*, 2021.
10. D. Mukunoki, M. Kawai, and T. Imamura, "Sparse Matrix-Vector Multiplication with Reduced-Precision Memory Accessor," in *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2023, pp. 608–615.
11. T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011.