Note on changing precision and scaling*

Zhengbo Zhou†

Contents

1 hpz19

[8] Higham, Pranesh & Zounon, Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems, SIAM Journal on Scientific Computing 41, A2536–A2551, 2019

Let α be a number stored in half precision. Then, $|\alpha| \in [6 \times 10^{-8}, 7 \times 10^{4}]$. Since it is so limited, therefore a straightforward rounding of single and double precision data into half precision can leads to overflow, underflow or subnormal numbers, all of which are undesirable.

Higham, Pranesh and Zounon [8] developed an algorithm for converting a matrix from single or double precision to half precision. We states its outline:

(1) perform two-sided diagonal scaling in order to equilibrate the matrix (ensure every row and column has ∞-norm 1);

*Date: March 26, 2025, Manchester, UK

†Department of Mathematics, University of Manchester, Manchester M13 9PL, United Kingdom. zhengbo.zhou@student.manchester.ac.uk

(2) multiply by a scalar to bring the largest element within a factor $\theta \leq 1$ of the overflow level;

(3) round to half precision.

The usage of half precision has the drawback that the elements of the matrix A may overflow or underflow when rounded to half precision. To see why, consider Table 1.

TABLE 1. *Parameters for* `bfloat16`, `fp16`, `fp32`, `fp64` *and* `fp128` *arithmetic, to three significant figures: unit roundoff* u, *smallest positive (subnormal) number* $x_{\min}^s$, *smallest normalized positive number* $x_{\min}$, *and largest finite number* $x_{\max}$.

	u	$x_{\min}^s$	$x_{\min}$	$x_{\max}$
`bfloat16`	3.91×10^{-3}	9.18×10^{-41}	1.18×10^{-38}	3.39×10^{38}
`fp16`	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
`fp32`	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
`fp64`	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}
`fp128`	9.63×10^{-35}	6.48×10^{-4966}	3.36×10^{-4932}	1.19×10^{4932}

When rounded to `fp16`, any double precion number with magnitude on the interval $[6.6 \times 10^4, 1.8 \times 10^{308}]$ will overflow, and any double precision number with magnitude on the interval $[2.2 \times 10^{-308}, 5.9 \times 10^{-8}]$ will underflow. Overflow is *unrecoverable*, because common factorizations cannot produce useful results for a matrix with infinities amongst its entries. Underflow during the rounding could cause a serious *loss of information*; moreover, the rounded matrix could have a zero row or column and hence be structurally singular.

The work on iterative refinement [3, 5, 6] all discussed the possibility of overflow, underflow and potential instability of using `fp16`.

1.1 Traditional treatments

Let $A^{(h)}$ be the matrix that rounded to half precision. Traditional treatment for *overflow* maps the overflowed numbers to the nearest largest representable number, $\pm x_{\max}$. Algorithm 1 is a litte more general. This algorithm with $\theta = 1$ is the approach used in [3, 5, 6].

Algorithm 1 Round then replace infinities. Mapping any elements of modulus greater than $\theta x_{\max}$ to $\pm \theta x_{\max}$.

Input: $A \in \mathbb{R}^{n \times n}$; $\theta \in (0, 1]$.
Output: The rounded `fp16` matrix $A^{(h)} \in \mathbb{R}^{n \times n}$.
$A^{(h)} = \mathrm{fl}_h(A)$
For every i, j such that $|a_{ij}| \geq \theta x_{\max}$, set $a_{ij}^{(h)} = \mathrm{sign}(a_{ij}) \theta x_{\max}$.

Another approach is to scale the matrix before rounding. Algorithm 2 ensures the largest entry in magnitude is $\theta x_{\max}$. In particular, the gap between $a_{\max}$ and $a_{\min}$ where $a_{\min} = \min_{i,j} |a_{i,j}|$ is retained after scaling and rounding if underflow does not occur.

Algorithm 2 Scale by scaler then round. Scaling all elements to avoid overflow.

Input: $A \in \mathbb{R}^{n\times n}$; $\theta \in (0,1]$.
Output: The rounded `fp16` matrix $A^{(h)} \in \mathbb{R}^{n\times n}$.

$a_{\max} = \max_{i,j}|a_{ij}|$
$\mu = \theta x_{\max}/a_{\max}$
$A^{(h)} = \mathrm{fl}_h(\mu A)$

1.2 Diagonal scaling then round

Algorithms 1 and 2 have the following drawbacks.

1. Algorithm 1 makes a potentially *large perturbation* for every element that overflows, so it can make a large change to the matrix.

2. When $a_{\max} > x_{\max}$, Algorithm 2 reduces every element in magnitude, so it increases the risk of underflow.

To address these issues, we consider the following class of algorithms: carry out two-sided diagonal scaling prior to converting to `fp16`, which replace A by RAS,

$$R = \mathrm{diag}(r_i), \quad S = \mathrm{diag}(s_i), \qquad r_i, s_i > 0, \quad i = 1:n.$$

There is *no* clear conclusion on when or how one should scale; see [2] for a experimental study. In any case, the focus of scaling algorithms in [8] is different from that in previous studies, where the aim of scaling has been to reduce the condition number or to speed up the converegnce. We scale in order to *help squeeze a single or double precision matrix into half precision*.

Our proto-algorithm for two-sided scaling is given in

Algorithm 3 Two-sided diagonal scaling then round.

Input: $A \in \mathbb{R}^{n\times n}$; $\theta \in (0,1]$.
Output: The rounded `fp16` matrix $A^{(h)} \in \mathbb{R}^{n\times n}$.

1: Apply any two-sided diagonal scaling algorithm to A, to obtain diagonal matrices R and S.
2: Let $\beta = \max_{i,j} |(RAS)_{ij}|$
3: $\mu = \theta x_{\max}/\beta$
4: $A^{(h)} = \mathrm{fl}_h(\mu(RAS))$

Lines 2–4 are essentially applying Algorithm 2 to RAS.

1.3 Determine diagonal scaling matrices

We now consider two different algorithms for determining R and S; both algorithms are carried out at working precision. We first consider row and column equilibration.

This scaling ensures that every row and column has maximum element in modulus equal to 1. The LAPACK routine `xyyEQU` carry out this form of scaling [1]. Notice that, Algorithm 3 applies row scaling before the column scaling. If the column scaling is

Algorithm 4 Row and column equilibration.

Input: $A \in \mathbb{R}^{n\times n}$ that has no zero row or column.
Output: Nonsingular diagonal matrices R and S such that $B = RAS$ has the property that $\max_k |b_{ik}| = \max_k |b_{kj}| = 1$ for all $i, j \in [1, n]$.

1: **for** $i = 1 : n$ **do** $r_i = \|A(i,:)\|_\infty^{-1}$ **end for**
2: $R = \mathrm{diag}(r)$
3: $\widetilde{A} = RA$
4: **for** $j = 1 : n$ **do** $s_i = \|\widetilde{A}(:,j)\|_\infty^{-1}$ **end for**
5: $S = \mathrm{diag}(s)$

applied first, a different result may be obtained. The result have the same characteristic scaling property, but the conditioning may be very different.

If the input matrix is symmetric then Algorithm 3 will generally destroy symmetry. Algorithm 5 preserves the symmetry [10].

Algorithm 5 Symmetry-preserving row and column equilibration

Input: $A \in \mathbb{R}^{n\times n}$; tol > 0 is a convergence tolerance.
Output: Nonsingular diagonal matrices R and S such that $B = RAS$ has the property that $\max_k |b_{ik}| = \max_k |b_{kj}| = 1$ for all $i, j \in [1, n]$. $R = S$ if A is symmetric.

1: $R = I$, $S = I$.
2: **while** $\max_i |r_i - 1| \le$ tol and $\max_i |s_i - 1| \le$ tol **do**
3: **for** $i = 1 : n$ **do**
4: $r_i = \|A(i,:)\|_\infty^{-1/2}$
5: $s_i = \|A(:,i)\|_\infty^{-1/2}$
6: **end for**
7: $A \leftarrow \mathrm{diag}(r) A \,\mathrm{diag}(s)$
8: $R \leftarrow \mathrm{diag}(r) R$
9: $S \leftarrow S \,\mathrm{diag}(s)$
10: **end while**

The algorithm 5 is iterative and scales simultaneously at both sides. It has the properties that

1. if $A = A^T$, then $R = S$;
2. then algorithm is permutation invariant: if it produces the scaling RAS for A, then it produces the scaling $P_1 R P_1^T (P_1 A P_2) P_2^T S P_2$ for $P_1 A P_2$;
3. it is linearly convergent with asymptotic convergence rate $1/2$.

We note that β in line 2 of Algorithm 3 is equal to 1 for Algorithms 4 and 5.

Other two-sided diagonal scaling algorithms exists, including Hungarian scaling [9] and other algorithms discussed by Larsson [11]. [8] mentioned they does not find any clear benefit of doing other algorithms.

1.4 Discussion

How do Algorithm 1, Algorithm 2, and Algorithm 3 with Algorithm 4 or Algorithm 5 compare for converting a matrix to `fp16`? Depending on the usage of the `fp16` matrix $A^{(h)}$, several possible criteria may be of interest.

1. As few elements as possible should underflow or become nonzero but unnormalized.
2. Key properties of the original matrix, such as singular values or condition number, should be preserved as much as possible.

The important insight in [8] is that after compute an LU factorization of $A^{(h)}$, they refine their solution in context of $Ax = b$ instead of $A^{(h)}y = b^{(h)}$. This can be achieved using a carefully constructed preconditioner. In fact, not only the scaled and unscaled algorithms are mathematically equivalent. There are even a numerical equivalence.

In conclusion, [8] developed a new conversion algorithm that employs two-sided diagonal scaling along with a potential scalar multiplication that moves the elements of largest magnitude close to the overflow threshold.

It seems that they are not mentioning the underflow and subnormal numbers generation.

Their code is available here https://github.com/SrikaraPranesh/fp16Scaling.

2 hipr21

[7] Higham & Pranesh, Exploiting Lower Precision Arithmetic in Solving Symmetric Positive Definite Linear Systems and Least Squares Problems, SIAM Journal on Scientific Computing 43, A258–A277, 2021

In section 3.1, they discuss about scale the matrix A to $H = D^{-1}AD^{-1}$, $D = \text{diag}(a_{ii}^{1/2})$. The matrix D wil be kept at working precision u. The aim is to reduce the dynamic range in order to avoid overflow and reduce the chance of underflow in conversion to lower precision.

> This is needed for `fp16`, but it is usually not necessary for `bfloat16` and single precision. ... We note that we could choose D to have diagonal elements that are powers of 2, to avoid rounding errors, but in our experience doing so brings no practical benefits.

This is same as the scaling in Algorithm 5. H is positive definite (Sylvester law of inertia) with diagonal entries 1, and $|h_{ij}| < \sqrt{h_{ii}h_{jj}} = 1$. Hence, the row and column equilibration is succeed with the largest elements all located along the diagonal. In fact, R and S converges to D if A is symmetric positive definite.

3 okca22

[12] Oktay & Carson, Multistage mixed precision iterative refinement, Numerical Linear Algebra with Applications 29, 2022

They use the conversion algorithm 3. They first attempt an LU factorization without scaling. They then test whether the result L and U factors contain `Inf` or `NaN`; if so, they retry the LU factorization using the two-sided scaling algorithm. Also, after compute the approximation solution x_0 for the GMRES-IR, if x_0 contains `Inf` or `NaN`, then simply use a zero vector as the input to the GMRES-IR.

In addition, they also incorporate scaling in each refinement step. After computing the residual r_i, they scale the result to obtain $r_i \leftarrow r_i/\|r_i\|_\infty$. This scaling is then undone when they update the approximate solution, via $x_{i+1} = x_i + \|r_i\|_\infty d_{i+1}$. As long as $1/\|A\|_\infty$ does not overflow and $\|A^{-1}\|_\infty$ does not overflow, this scaling avoids the largest element of d_{i+1} overflowing or underflowing.

4 sctu24

[14] Scott & Tøuma, Avoiding Breakdown in Incomplete Factorizations in Low Precision Arithmetic, ACM Transactions on Mathematical Software 50, 1-25, 2024

Uses 2-norm scaling, i.e. entries in column j of A are normalized by the 2-norm of column j [13, Sec. 3.2.3].

5 sctu25

[15] Scott & Tůma, Developing robust incomplete Cholesky factorizations in half precision arithmetic, Numerical Algorithms 2025

In practice, numbers with small absolute values are often flushed to zero since the factorization is approximated anyway. This work also uses the 2-norm scaling which is the same as in [14].

6 hbtd20

[4] Haidar, Bayraktar, Tomov, Dongarra & Higham, Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences 476, 2020.

Similarly, it can only *avoid overflow and reduce underflow*. They suggested using Algorithm 3, which can reduce the possibility of underflow by balancing.

7 What does LAPACK do?

```
safmin = dlamch( 'Safe minimum' )
eps = dlamch( 'Precision' )
smlnum = safmin / eps
bignum = one / smlnum
rmin = sqrt( smlnum )
```

```
rmax = sqrt( bignum )
anrm = dlansy( 'M', uplo, n, a, lda, work )
iscale = 0
IF( anrm.GT.zero .AND. anrm.LT.rmin ) THEN
iscale = 1
sigma = rmin / anrm
ELSE IF( anrm.GT.rmax ) THEN
iscale = 1
sigma = rmax / anrm
END IF
IF( iscale.EQ.1 )
    CALL dlascl( uplo, 0, 0, one, sigma, n, n, a, lda, info )
```

This part of code is from LAPACK routine `dsyev`. For all symmetric solvers in LAPACK, they will perform similar scaling.

$$\texttt{smlnum} = \frac{x_{\min}}{2u}, \quad \texttt{bignum} = \frac{1}{\texttt{smlnum}}$$

Here, $2u$ is the machine precision (mathematically). Divided by $2u$ ensures the division by tiny number gives meaning digits. Taking the *square roots* ensures the column norms or any kind that requires the square of an entry does not overflow.

When the largest element is below `smlnum`, it will lift all the elements up until the largest element meet `smlnum`. On the other hands, when the largest element is above `bignum`, it will lower all the elements down until the largest element meet `bignum`. Therefore, underflow may still exists.

Remark 7.1. It is interesting that no one cares about underflow too much. It seems the most important thing is not to overflow.

Remark 7.2. For `dgesvj` (one-sided Jacobi), it scales the column norm such that no overflow will appear. It has two vectors relate to the singular values, the square roots of the column norms and their scaling factors.

References

[1] Edward Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Third edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. xxi+429 pp. ISBN 0-89871-447-8 (paperback). (Cited on p. 3)

[2] Joseph M. Elble and Nikolaos V. Sahinidis. Scaling linear optimization problems prior to application of the simplex method. *Computational Optimization and Applications*, 52(2): 345–371, 2011. (Cited on p. 3)

[3] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. *The design of fast and energy-efficient linear solvers: on the potential of half-precision arithmetic and iterative refinement techniques*, chapter Part I, pages 586–600. Springer International Publishing AG, 2018. ISBN 9783319936987. (Cited on p. 2)

[4] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 476(2243), 2020. (Cited on p. 6)

[5] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC18 (Dallas, TX), Piscataway, NJ, USA, 2018, pages 47:1–47:11. IEEE. (Cited on p. 2)

[6] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, ACM Press, November 2017, pages 1–8. (Cited on p. 2)

[7] Nicholas J. Higham and Srikara Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM Journal on Scientific Computing*, 43(1):A258–A277, 2021. (Cited on p. 5)

[8] Nicholas J. Higham, Srikara Pranesh, and Mawussi Zounon. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM Journal on Scientific Computing*, 41(4):A2536–A2551, 2019. (Cited on pp. 1, 3, 4, 5)

[9] James Hook, Jennifer Pestana, Françoise Tisseur, and Jonathan Hogg. Max-balanced Hungarian scalings. *SIAM Journal on Matrix Analysis and Applications*, 40(1):320–346, 2019. (Cited on p. 4)

[10] Philip A. Knight, Daniel Ruiz, and Bora Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications*, 35(3):931–955, 2014. (Cited on p. 4)

[11] Torbjörn Larsson. On scaling linear programs—some experimental results. *Optimization*, 27(4):355–373, 1993. (Cited on p. 4)

[12] Eda Oktay and Erin Carson. Multistage mixed precision iterative refinement. *Numerical Linear Algebra with Applications*, 29(4), 2022. (Cited on p. 5)

[13] Jennifer Scott and Miroslav Tůma. `HSL_MI28`: An efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Transactions on Mathematical Software*, 40 (4):1–19, 2014. (Cited on p. 6)

[14] Jennifer Scott and Miroslav Tůma. Avoiding breakdown in incomplete factorizations in low precision arithmetic. *ACM Transactions on Mathematical Software*, 50(2):1–25, 2024. (Cited on p. 6)

[15] Jennifer Scott and Miroslav Tůma. Developing robust incomplete Cholesky factorizations in half precision arithmetic. *Numerical Algorithms*, 2025. (Cited on p. 6)