Implementation of Various 8-bit Data Types in ⟨T⟩LAPACK

Sudhanva Kulkarni and Dominic Lioce
Math 221 Class Project
University of California, Berkeley

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Implementation of Various 8-bit Data Types in ⟨T⟩LAPACK

***Abstract*—8-bit floating-point data formats are being explored as an option to increase the speed of machine learning computation. This work aimed to implement and test three 8-bit data types into the developmental numerical linear algebra package ⟨T⟩LAPACK. The data types varied the number of exponent and mantissa bits stored, greatly changing the range and precision of possible floating-point representations. The algorithms used for testing were LU decomposition (by Gaussian elimination with Partial Pivoting) and QR factorization (by Householder reflections), both available in ⟨T⟩LAPACK. The incorporation of stochastic floating-point rounding methods greatly improved accuracy for QR factorization, while LU decomposition was made slightly less accurate. To account for the small range of the P3109 floats, some basic scaling routines were implemented to squeeze all the matrix entries into this small range and prevent overflow and underflow. The use of 32-bit BLAS routines alos made it easy to escape overflow issues.**

I. Introduction

Recent advances in machine learning emphasize the importance of using large amounts of data for training models, or moving data quickly to provide on-the-fly calculation to make time-sensitive decisions [3]. Self-driving cars, for example, require quick-decision making based on inputs; incorrect decisions or indecisiveness are a safety hazard in many artificial intelligence systems. Additionally, data movement is the main bottleneck for decreasing computational speed: data movement between lower levels of memory is tens or thousands times slower (depending on the architecture) than doing floating point calculations [2]. Therefore, it is necessary to have fast calculations that sacrifice some precision in favor of faster data movement.

To address this issue, IEEE Working Group P3109 has been attempting to provide a standard for 8-bit data types and arithmetic with the goal of enhancing the speed of machine learning calculations [5]. In machine learning applications, precisions tend to be lower and accuracy is quantified in terms of correct predictions (inference accuracy) rather than error on a numerical method specifically. In order to make methods consistent between different hardware applications, standards must be developed.

Numerical linear algebra is fundamental to the functionality of machine learning [4]. One building block that serves as a standard for performing numerical linear algebra calculations has been Linear Algebra PACKage, or LAPACK (Basic Linear Algebra Subroutines, or BLAS, is also implemented in LAPACK). Templated LAPACK, or ⟨T⟩LAPACK, is being developed as a templated C++ implementation of LAPACK (originally written in Fortran 77). ⟨T⟩LAPACK, which is being developed under the Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC) Framework, has three main goals [6]:

1) Provide seamless access to the latest algorithms, numerics, and performance using familiar Sca/LAPACK interfaces.
2) Offer advanced capabilities through new interface extensions when necessary.
3) Serve as an efficient conduit for quickly integrating cutting-edge research discoveries into applications used by science and engineering communities relying on high-performance linear algebra libraries.

This work aimed to implement and test several 8-bit floating point formats in ⟨T⟩LAPACKwith the goals of informing the IEEE P3109 standards committee and creating additional accessibility for a widely used numerical linear algebra package that could be used with machine learning applications in the future. It also aimed to increase the accuracy of solutions using various methods.

II. Methods

A. 8-bit floats

8-bit floats operate the same way as regular floating point numbers. They are parametrized by 3 variables. The exponent e, the mantissa m and the sign s. There is also an additional bias that we apply

to the exponent. Any floating point number f is written as $(-1)^s \cdot 2^{e-bias} \cdot 1.m$. The P3109 committee proposes a a general 8-bit floating point format that is parametrized by a precision p. A P3109 float of precision p is assigned $8-p$ exponent bits, $p-1$ mantissa bits and 1 sign bit. Further, it is assigned a bias of $2^{8-p-1}-1$. Below, we list 3 different 8-bit floats of this type along with their minimum and maximum values. The P3109 format also has non-signaling NaNs and no -0 (NaN is given the position of -0) [1]. As in the case of IEEE-754 floats, the machine epsilon is given by 2^{1-p} Table III:

Type	Smallest Number	Largest Number
e4m3	3.051758E-05	49152
e5m2	4.6566129E-10	2147483648
e3m4	7.8125E-03	224

Table I: 8-bit data formats chosen for this study.

The different 8-bit data types were tested using both LU decomposition and QR factorization, both included as examples in ⟨T⟩LAPACK. The QR factorization uses Householder transformations, as would be common in most applications. The LU factorization utilizes GEPP to increase numerical stability [2]. The error metrics used for each of these methods can be seen in equations (1) and (2):

$$E_{LU} = \frac{\|LU - A\|_F}{\|A\|_F} \tag{1}$$

$$E_{QR} = \frac{\|QR - A\|_F}{\|A\|_F} \tag{2}$$

We also considered another error metric for QR, which was the Orthogonality of the Q matrix that was generated. The orthogonality is given below

$$\|QQ^T - I\|_{max} \tag{3}$$

Due to the lower precision and range of 8-bit data types, rounding methods, scaling methods and high precision accumulation were implemented to increase the accuracy of both LU and QR.

B. Accumulation in 32-bits

When working with such a narrow and course range of numbers, one often runs into overflow and underflow issues. One way we chose to combat this problem was the use of mixed precision accumulation. So, BLAS routines like GEMM and TRSM which look are carried out in a higher precision like 32 or 16 bits, and then the result is rounded back down to 8-bit precision. This procedure helped stabilize the routines significantly and is required for matrices of any significant size. For matrices of size $n > 10$, the error quickly spirals out of control for every format without the use of high precision accumulation. This procedure is easily implemented in C++ through the use of templated functions. It is also not too hard to implement in hardware as one can simply acccumulate all the 8-bit floats into a single 32-bit register. One can consider several methods for rounding back to 8-bit precision, and we discuss a small subset of that design space in the next section.

C. Rounding

Within each 8-bit data format, there are several different options for rounding. The IEEE standard 754 for binary floating-point arithmetic defines four rounding modes [7]:

- Round to nearest (round towards even in the event of a tie)
- Round toward 0
- Round toward $-\infty$
- Round toward ∞

The default setting is the first, which will round to the nearest number.

A common alternative for lower-precision floating-point formats is stochastic rounding, wherein a floating-point operation is rounded to the next larger or next smaller floating-point number with a probability that is 1 minus the relative distance of some number x to each of those numbers [8]. In equation form,

$$\text{fl}(x) \begin{cases} \lceil x \rceil \text{ with prob. } p = (x - \lfloor x \rfloor)/(\lceil x \rceil - \lfloor x \rfloor) \\ \lfloor x \rfloor \text{ with prob. } 1-p \end{cases} \tag{4}$$

In practice, this is implemented by simply adding random bits and truncating. Stochastic rounding has been shown to increase accuracy significantly

when working with lower precision [8]. A proof sketch for the other method can easily be seen by noting that for the truncated number to be greater than $\lceil x \rceil$, one would need to add $f(\lceil x \rceil - x)$ to x(where $f(a)$ means the fractional part of a). The range of numbers that can do this is the interval $[f(\lceil x \rceil) - f(x), f(\lceil x \rceil)]$, So the probability of generating a bitstring that lies between $[f(\lfloor x \rfloor), f(\lceil x \rceil)]$ and satisfies the above property is given by $\frac{f(\lceil x \rceil)-(f(\lceil x \rceil)-f(x))}{-f(\lfloor x \rfloor)+f(\lceil x \rceil)} = \frac{x-\lfloor x \rfloor}{f(\lceil x \rceil)-f(\lfloor x \rfloor)}$, which is exactly what we wanted.
Stochastic rounding is an attractive choice for many reasons. Unlike deterministic rounding, it eliminated the existence of an inherent bias during numerical computation. Further, as shown in [8], stochastic rounding allows us to convert several $O(n)$ bounds into $O(\sqrt{n})$ bounds (The Connolly paper shows it for dot products) which is characteristic of many Monte Carlo methods in scientific computing.

These different rounding schemes were applied in ⟨T⟩LAPACK in three ways:

- Fully Deterministic Rounding: Deterministic 32-bit BLAS operations, deterministic round down back to 8-bit, deterministic operations in 8-bit.
- Hybrid Rounding: Deterministic 32-bit BLAS operations, stochastic round down to 8-bit, stochastic operations in 8-bit.
- Fully Stochastic Rounding: Stochastic 32-bit BLAS, stochastic round to 8-bit and stochastic operations in 8-bit.

As shown in [12] and [11] , one can derive bounds for a mixed precision, Householder QR as $O(n(\epsilon_1 + m\epsilon_2))$ where the ϵ_i are the machine epsilons for the courser and finer precisions respecyively. If we change any of the precisions to be stochastic, the corresponding n or m gets replaced by a $\sqrt{n}$ or $\sqrt{m}$ (shown in). This error bound is the basis for picking which rounding method works best. In particular, it lets use easily eliminate Fully Stochastic Rounding as the machine epsilon for 32-bit floats is very small compared to that of the 8-bit floats. So the reduction from m to $\sqrt{m}$ doesn't mean much as the error is largely dominated by the $n\epsilon_1$ term. This bound also agrees with experiment as the error ends up being identical for for the second and third rounding modes.

D. Scaling

We also investigated various scaling algorithms to accommodate for the small range of the 8-bit types. The main scaling algorithms used are as follows:

- Unit scaling, seeking unit variance for all entries within the matrix [9]
- Symmetry-preserving row and column equilibration scaling, a two-sided diagonal scaling algorithm [10]
- Norm-Global scaling by a constant factor of $\sqrt{fl_{max} \cdot \epsilon / \|A\|_1}$ to avoid overflow/underflow
- Column-wise scaling by a factor of $\sqrt{fl_{max} \cdot \epsilon / \|A_i\|_1}$

The second scaling algorithm listed above has the algorithmic interpretation seen in Figure 1 [10]:

Algorithm 2.5 (symmetry-preserving row and column equilibration). Given $A \in \mathbb{R}^{n\times n}$, which is assumed to have no zero row or column, this algorithm computes nonsingular diagonal matrices R and S such that $B = RAS$ has the property that $\max_k |b_{ik}| = \max_k |b_{ki}| = 1$ for all i and $R = S$ if $A = A^T$. 'tol' is a convergence tolerance.

$R = I$, $S = I$
repeat
  for $i = 1\colon n$ **do**
    $r_i = \|A(i,:)\|_\infty^{-1/2}$
    $s_i = \|A(:,i)\|_\infty^{-1/2}$
  end for
  $A = \mathrm{diag}(r) A \,\mathrm{diag}(s)$
  $R = \mathrm{diag}(r) R$
  $S = S\,\mathrm{diag}(s)$
until $\max_i |r_i - 1| \le \mathrm{tol}$ and $\max_i |s_i - 1| \le \mathrm{tol}$

Figure 1: Symmetry-preserving row and column equilibration scaling, a two-sided diagonal scaling algorithm [10].

Unit scaling was designed to be used for training Neural Networks in low precisions. The main idea behind it is to ensure that all operations leave the matrix with unit variance. An example of this would be globally scaling the product of two matrices A and B by $\sqrt{n}$.
Symmetric row Column equilibration essentially calculates two diagonal matrices R and S and runs LU decomposition on RAS rather than A(note that this cannot be used for QR as it would completely change the Q). This squeezes the entries of A into the $[-1, 1]$ range and lets us perform LU fairly reliably even on matrices that have entries of the order 10^7 or 10^{-7}. As shown in [10], this method

worked relatively well for half precisions and so we expect it to work well for the 8-bit format too.
Norm-Global scaling is exactly as described above. It works relatively well for matrices where entries are of about the same magnitude, but may suffer from underflow issues in case our matrix has rows and columns of different magnitudes. As will be seen later, the orthogonality of Q also takes a hit in such a situation. Such matrices can show up in machine learning where different features can have varying magnitudes. A simple example can be seen in the Boston Housing dataset.
To address this problem, we introduce a column-wise norm-scaling so that we can write $A = QRD$ rather than $A = QR$.

III. Results

Unit scaling did not provide any noticeable advantages over the norm-global scaling. Symm-Equilibration largely expanded the range of LU and brought down the error by a small amount. We also observed that the vanilla algorithm had a hard time converging and would often run indefinitely. To work around this, we added a maximum number of iterations as well so that the algorithm terminates quickly. LU can be carried out far beyond the range of acceptable numbers with an appropriate hit to the accuracy.

Figures 2 through 5 for LU and QR used only global norm scaling in order to provide an accurate comparison between the different rounding schemes. The matrices generated for these tests had random entries $A(i,j) \sim \text{Uniform}(-1,1)$.

For QR, we compare the global scaling with column-wise scaling by looking at the the average orthogonality of Q for random matrices of size 10,25,50 and 100. Specifically, we tested with matrices that were constructed so that every odd column had elements from $Uniform(-1000, 1000)$ and every even column had $Uniform(-1, 1)$. We chose this metric as it was easier to detect underflow damage here compared to the standard error formula. The matrices were constructed as above since we wanted to confirm the usefulness of column scaling in relatively extreme situations/ The results for e4m3 and e5m2 are presented in Tables 2 and 3

As seen by the tables, column scaling doesn't seem to provide too much of an advantage when the

Figure 2: Accuracy of LU decomposition as a function of matrix size n: deterministic rounding.

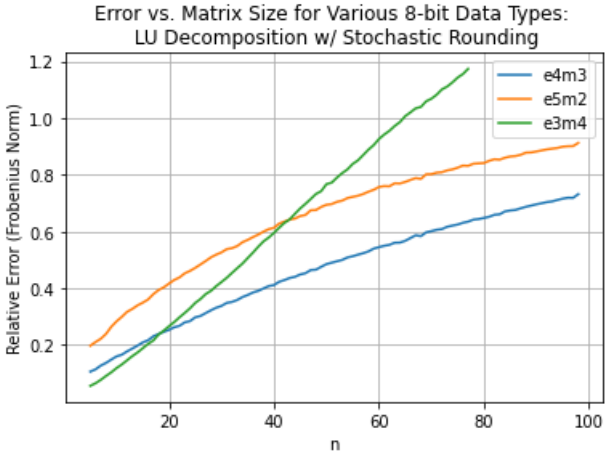

Figure 3: Accuracy of LU decomposition as a function of matrix size n: stochastic rounding.

difference in magnitudes is of order 10^3. However, the error becomes pronounced for larger difference in magnitude (order 10^7 or 10^8) where for column scaling, we see errors of 1.385 and 1.14 (e4m3, e5m2), and for global scaling, we see errors of 3.3375 and 2.875 (e4m3, e5m2). The error is also more pronounced for larger n. For $n = 200$ and a difference in magnitudes of

IV. Conclusions and Future Work

The implementation of three selected 8-bit floating-point formats into ⟨T⟩LAPACK was successful. The rounding method that utilizes deterministic floating-point operations in 32-bit before stochastically rounding down to 8-bit and doing 8-bit

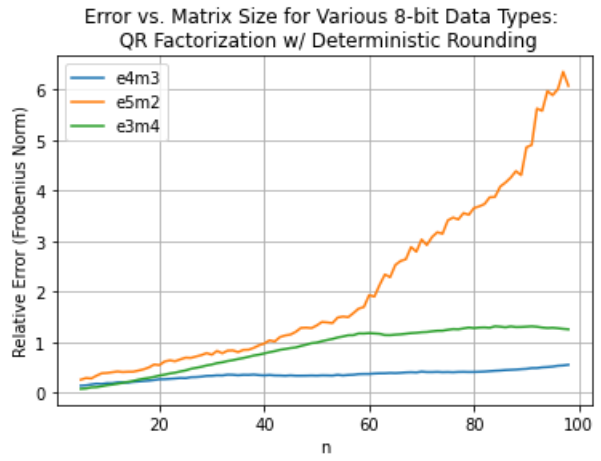

Figure 4: Accuracy of QR factorization as a function of matrix size n: deterministic rounding.

Figure 5: Accuracy of QR factorization as a function of matrix size n: stochastic rounding.

floating-point operations stochastically improves the accuracy of QR factorization for all three tested 8-bit implementations significantly. The long-term behavior of said QR factorization sees the error scale with $n^{1/2}$, which is favorable to the unpredictable trend when using a purely deterministic rounding scheme. Stochastic rounding appears to interfere with LU decomposition though, even disallowing the LU decomposition of *e3m4* type for $n > 73$. Implementation of more 8-bit types could also be attempted, though data types with lower range may not function well for large matrices when paired with LU, and data types with lower precision may not produce as accurate of results in general. Other linear algebra routines (Cholesky, SVD, etc.) could also be explored using 8-bit data types. The results from LU or QR can also be paired up with relavent iterative refinement schemes such as GMRES-IR (as in [10]) to improve the accuracy of the solutions. The ultimate goal of this endeavour would be to eventually try to effectively solve problems like rank-deficient least squares so as to contribute to the ongoing effort of speeding up machine learning algorithms.

n	**e4m3**	**e5m2**
10	0.45	0.89
25	0.435	1.44
50	0.72	2.0
75	0.67625	1.67
100	0.8033	1.7925

Table II: average orthogonality with column-wise scaling

n	**e4m3**	**e5m2**
10	0.468125	0.97
25	0.54	1.46
50	0.70875	2.075
75	0.80875	1.555
100	0.87625	1.8175

Table III: average orthogonality with global scaling

REFERENCES

[1] IEEE Working group P3109 Interim Report on 8-bit Binary Floating-Point Formats

[2] J. Demmel, Numerical Linear Algebra. Berkeley, CA: Center for Pure and Applied Mathematics, University of California, 1993.

[3] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," 2017 IEEE Custom Integrated Circuits Conference (CICC), 2017. doi:10.1109/cicc.2017.7993626

[4] P. Caceres, Introduction to Linear Algebra for Applied Machine Learning with Python, https://pabloinsente.github.io/intro-linear-algebra (accessed Dec. 5, 2023).

[5] Institute of Electrical and Electronics Engineers, "IEEE Working Group P3109 Interim Report on 8-bit Floating-point Formats." New York, New York, Nov. 24, 2023

[6] "Collaborative Research: Frameworks: Basic Algebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC)," NSF AWARD SEARCH: Award # 2004850, https://www.nsf.gov/awardsearch/showAward?AWD_ID=2004850 (accessed Oct. 9, 2023).

[7] "IEEE SA - IEEE standard for floating-point arithmetic," IEEE Standards Association, https://standards.ieee.org/ieee/754/6210/ (accessed Dec. 5, 2023).

[8] M. P. Connolly, N. J. Higham, and T. Mary, “Stochastic rounding and its probabilistic backward error analysis,” SIAM Journal on Scientific Computing, vol. 43, no. 1, 2021. doi:10.1137/20m1334796

[9] C. Blake, D. Orr, and C. Luschi, “Unit scaling: Out-of-the-box low-precision training,” arXiv.org, https://arxiv.org/abs/2303.11257 (accessed Dec. 5, 2023).

[10] N. J. Highham, S. Pranesh, and M. Zounon, “Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems”, 2018

[11] M. P. Connolly and N. J. Higham, “Probabilistic Rounding Error Analysis of Householder QR Factorization”, 2022

[12] L. Minah Yang, Alison Fox and Geoffrey Sanders. ”Rounding error analysis of mixed precision block Householder QR algorithms”. SIAM J Scie. Comput. 43(3), A1723-A1753, 2021