

Increasing Accuracy of Iterative Refinement in Limited Floating-Point Arithmetic on Half-Precision Accelerators

Piotr Luszczek
University of Tennessee

Ichitaro Yamazaki
*Sandia National Laboratories**

Jack Dongarra
*University of Tennessee
Oak Ridge National Laboratory
University of Manchester*

Abstract—The emergence of deep learning as a leading computational workload for machine learning tasks on large-scale cloud infrastructure installations has led to plethora of accelerator hardware releases. However, the reduced precision and range of the floating-point numbers on these new platforms makes it a non-trivial task to leverage these unprecedented advances in computational power for numerical linear algebra operations that come with a guarantee of robust error bounds. In order to address these concerns, we present a number of strategies that can be used to increase the accuracy of limited-precision iterative refinement. By limited precision, we mean 16-bit floating-point formats implemented in modern hardware accelerators and are not necessarily compliant with the IEEE half-precision specification. We include the explanation of a broader context and connections to established IEEE floating-point standards and existing high-performance computing (HPC) benchmarks. We also present a new formulation of LU factorization that we call signed square root LU which produces more numerically balanced L and U factors which directly address the problems of limited range of the low-precision storage formats. The experimental results indicate that it is possible to recover substantial amounts of the accuracy in the system solution that would otherwise be lost. Previously, this could only be achieved by using iterative refinement based on single-precision floating-point arithmetic. The discussion will also explore the numerical stability issues that are important for robust linear solvers on these new hardware platforms.

I. INTRODUCTION

Modern high-performance computing (HPC) hardware continues to experience an ongoing shift towards supporting a variety reduced-precision formats for representing floating-point numbers in order to offer a much increased performance rate. However, portability is often of little concern as the hardware tends to serve only a specific set of workloads that are of special interest to the particular vendor. The examples include Intel’s

Cascade Lake Vector Neural Network Instructions (VNNI) and the recently announced Xe platform for graphics cards, AMD’s Radeon Instinct cards (MI5, MI8, MI25, MI55, MI60) and NVIDIA’s compute cards from the Pascal, Volta, and Turing series. Finally, ARM included 16-bit floating point (FP16) in its NEON vector unit specification VFP 8.2-A. These accelerators follow two types of specifications for 16-bit floating-point format: IEEE-compliant FLOAT16 and extended-range BFLOAT16.

At the same time, a new breed of accelerators take the use of reduced precision to a new level as they target new machine learning workloads with little or no regard for the established floating-point standards. These accelerators are currently under information embargo due to competitive advantage reasons and therefore little is known about them. This new hardware includes Cloud AI 100 by Qualcomm, Dot-Product Engine by HPE, Eyeriss by MIT [1], TPU by Google [2], Deep Learning Boost by Intel, and Zion by Facebook.

However, our perspective is different as we try to use these new hardware chips to provide numerical accuracy that is comparable to standards-based floating-point computation while attaining predictable error bounds if such a guarantee is possible.

Along these lines, we propose novel schemes that result in mitigation strategies for a mixed-precision iterative refinement algorithm that allows the use of lowest-precision format and take advantage of its computational benefits. Depending on the platform, limited precision may be as high as $10\times$ faster than the double-precision peak performance. For example, on NVIDIA Volta, FLOAT64 tops out at about 6 teraFLOP/s while Tensor Core units in the same chip are capable of 120 teraFLOP/s.

The rest of the paper is organized as follows: Section II provides related work information in the area of limited-precision iterative refinement; Section III contains the details of the problem we aim to solve and Sections IV, V, and VI describe three mitigating strategies to deal with the issues described in the prior section; finally, Section VIII includes the experimental results combined with the discussion and Section IX concludes the paper and provides potential future research directions.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was partially supported by NSF Grant No. OAC 1740250 and CSR 151428.

*This work was done while the author was at the University of Tennessee, USA. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract de-na0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

II. RELATED WORK

The iterative refinement algorithm has been known for many decades as an effective tool for increasing accuracy of a solution of a set of simultaneous linear equations [3], [4], [5]. It involves higher- and lower-precision floating-point arithmetic that are applied judiciously to preserve the quality of the solution while increasing overall performance. Higher-precision arithmetic is often possible through already available hardware units, though it is slower than the lower-precision. Absent appropriate hardware, higher precision may be realized through software-based techniques [6], [7]. The requirement for achieving improvement in the quality of the solution is to perform the accumulation of the residual $r \equiv Ax - b$ in higher-precision arithmetic than what is used for the L and U factors.

If it is possible to customize the working precision, as is the case on field-programmable gate arrays (FPGAs), then the refinement can be exploited by creating custom floating-point units [8]. Alternatively, it is possible to exploit the system matrix conditioning and use the iterative refinement as a method of accessing faster hardware capabilities [9]. We take this idea further—but working within the constraints of the modern hardware that limits the precision of the fastest format and thus creates greater pressure on the algorithmic development to counteract the numerical issues.

It was proposed to use Krylov subspace iteration based on GMRES and the integration into the residual refinement process was called GMRES-IR [10]. The impact of system matrix spectrum on the convergence and accuracy of such a GMRES iteration was studied by varying the spread and clustering of the singular values of the system matrix [11]. The representation of full-precision data in half precision poses additional issues, and a form of matrix scaling was proposed to address it [12].

The analogous effort in deep learning involves training the network in lower precision and performing inference in a higher one [13], [14]. The compute imbalance between training and inference is even higher than that of factorization and the subsequent iterative refinement. Another difference is that in the context of neural network training, lowering the precision may be incorporated into the model as a regularizer.

A matrix may offer an opportunity for a faster solver if the condition number is low enough and the speed of lower precision may be utilized by the available hardware. Taking advantage of the condition number of the L and U factors in the context of least squares problems may be performed as long as appropriate regularization is involved [15].

III. PROBLEM STATEMENT: ITERATIVE REFINEMENT IN LIMITED PRECISION

Figure 1 shows two most common 16-bit formats for storing and computing floating-point values on existing hardware. This excludes possibilities afforded by synthesizing custom floating-point designs on FPGAs.

Our goal is to solve a system of linear equations:

Format	sign	mantissa and exponent bits													
		10 + 5													
IEEE 754:	$\pm$														
unit round-off	$=$	5×10^{-4}													
		7 + 8													
BFLOAT16:	$\pm$														
unit round-off	$=$	4×10^{-3}													

Fig. 1. Half precision representations in industry standards and modern hardware. Yellow squares represent mantissa bit and navy blue squares represent exponent bit.

Algorithm 1: Mixed precision iterative refinement with the 16-bit factorization and 64-bit corrections.

1	$L^{(16)}, U^{(16)}, P \leftarrow \text{lu}(A^{(16)})$	$\mathcal{O}(n^3) \times 16$ bits
2	$v^{(16)} \leftarrow P \backslash b^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
3	$y^{(16)} \leftarrow L^{(16)} \backslash v^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
4	$x_0^{(16)} \leftarrow U^{(16)} \backslash y^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
5	for $k = 1, 2, \dots$ do	
6	$r_k^{(64)} \leftarrow b^{(64)} - A^{(64)} x_{k-1}^{(64)}$	$\mathcal{O}(n^2) \times \boxed{64}$ bits
7	$s_k^{(16)} \leftarrow P \backslash r_k^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
8	$t_k^{(16)} \leftarrow L^{(16)} \backslash s_k^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
9	$z_k^{(16)} \leftarrow U^{(16)} \backslash t_k^{(16)}$	$\mathcal{O}(n^2) \times 16$ bits
10	$x_k^{(64)} \leftarrow x_{k-1}^{(64)} + z_k^{(64)}$	$\mathcal{O}(n) \times \boxed{64}$ bits

$$Ax = b \quad (1)$$

where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$.

Algorithm 1 shows an iterative refinement adopted from the 32/64 bit formulation [16], [17], [18] to the 16/64 bit scenario. The algorithm solves the system from Eq. (1) by introducing representation of matrices and vectors in fixed-precision arithmetic:

$$A^{(64)} x^{(64)} = b^{(64)} \quad (2)$$

The algorithm lowers the precision from 64 to 32 in order to perform all the $\mathcal{O}(n^3)$ operations at the speed of low-precision hardware. There are three consequences from the numerical stability perspective:

- 1) Small residual vectors cannot be represented due to limited representation range of the lower precision format: $\|r^{(64)}\| < \text{FP}_{\min}$.
- 2) Residual vectors are not accurately represented in lower precision due to large unit round-off: $U^{(16)} > 5 \times 10^{-4}$.
- 3) The factorization suffers from a loss of accuracy due to limited storage bits for the L and U factors: $\|L^{(16)} \times U^{(16)} - P \times A^{(64)}\| \geq \|L^{(32)} \times U^{(32)} - P \times A^{(64)}\|$ and this is further exacerbated by the matrix's condition number $\kappa(A)$.

Note that, in the context of least squares minimization, it might be more preferable to numerically find $\min \|b - Lx\|_2^2$ rather than the original $\min \|b - Ax\|_2^2$ which is much more

computationally expensive. This is especially true if the L factor is better conditioned than the original system matrix A [19]. The condition number of the system matrix A depends on the originating application; but in a benchmarking context, $\kappa(A)$ can be a controlled quantity. A known result in that regard states that $\kappa_2(A) \approx \sqrt{n}$ if the entries of A are normal i.i.d. symmetrically around zero: $x_{ij} \in \mathcal{N}(0, 1)$ [20]. This can be guaranteed in practical situations through the correct choice of the pseudo random number generator (PRNG) for the system matrix A . In fact, the High Performance LINPACK (HPL) benchmark uses a uniform distribution around zero: $\mathcal{U}(-\frac{1}{2}, \frac{1}{2})$ [21] which is sufficiently well conditioned in practice in order to bound the condition number at a reasonable level. And at the same time, matrices generated in such a way would still cause an excessive pivot growth, especially when partial pivoting is not used.

IV. COPING WITH LIMITED RANGE: PROMOTION TO SINGLE PRECISION

One of the main reasons why the classic mixed-precision iterative refinement does not experience problems with limited range was due to the fact that single precision format devotes 8 bits to the exponent. The number of exponent bits is much more limited in half-precision format from IEEE (BFLOAT16, of course, does not have this problem as it shares its exponent format with FLOAT32). In order to deal with this limitation, we propose to promote the computation of the solves during the refinement to single-precision. This could be achieved by making another copy of the L and U factors in single precision, but this would increase the storage complexity by $\mathcal{O}(n^2)$. Instead, we simply perform the promotion on-the-fly during the back and forward solves. This induces additional computational cost of conversion during each iteration—but this is completely masked by the memory-access overhead since the solves are bandwidth-limited.

V. COPING WITH LIMITED RANGE: ADAPTIVE RESIDUAL SCALING

To cope with the limited representation range, we propose to scale the residual vectors as their norm becomes small so they can be represented in the lower-precision format while the scaling factor is represented in higher precision. In this strategy, we focus on lines 8 and 9 of Algorithm 1, and propose to replace these lines with the following three steps that scale the residual vector before casting it in lower precision (the scaling factor is chosen to be the maximum value with respect to the magnitude):

- 1) $t_k^{(16)} \leftarrow L^{(16)} \setminus \left(s_k^{(16)} / \max |r_k^{(64)}| \right)$
- 2) $z_k^{(16)} \leftarrow U^{(16)} \setminus t_k^{(16)}$
- 3) $x_k^{(64)} \leftarrow x_{k-1}^{(64)} + z_k^{(64)} \times \max |r_k^{(64)}|$

In the first step, scaling the k -th vector $\|s_k\| / \max |r_k|$ brings the entries of the residual closer to unity where they can be represented most accurately in lower precision. Note that the scaling has to take place in higher precision where the greater range can capture the exponent correctly.

```
#include <mma.h>
using namespace nvccuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::
        col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::
        row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float>
        c_frag;

    // Initialize the output to zero
    wmma::fill_fragment(c_frag, 0.0f);

    // Load the inputs
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);

    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Store the output
    wmma::store_matrix_sync(c, c_frag, 16, wmma::
        mem_row_major);
}
```

Fig. 2. Sample C++ code targeting Tensor Core units with WMMA primitives available in CUDA software stack targeting compute capability 7 hardware including NVIDIA Volta and Turing.

VI. COPING WITH LIMITED ACCURACY: PRECISION PARTITIONING

NVIDIA CUDA 9 and its hardware’s compute capability 7 introduced warp-level operations that leverage Tensor Cores. Their main purpose is to perform warp-synchronous matrix multiply-accumulate (WMMA) of either the out-of-place form $D \leftarrow A \times B + C$ or the in-place form $C \leftarrow A \times B + C$ where $A, B, C, D \in \mathbb{R}^{4 \times 4}$ and the precision of the output matrices could be either 16- or 32-bits. We are interested in leveraging this capability to capture additional precision bits during the refinement.

Figure 2 shows a sample code for in-place matrix multiplication executed with a call to `wmma::mma_sync`. Our observation is that we can pack four independent vectors into `b_frag` and a single instruction would then perform multiplication by 4 independent right-hand side vectors. We propose the following partitioning of a 64-bit vector into 4 16-bit vectors:

$$X^{(64)} \equiv \left[x_{1 \div 16}^{(64)} | x_{17 \div 32}^{(64)} | x_{33 \div 48}^{(64)} | x_{49 \div 64}^{(64)} \right] \quad (3)$$

where $x_{1 \div 16}$ represents bits from 1 to 16 of the mantissa of $x^{(64)}$.

Note that, at the implementation level, this partitioning could be achieved with a pair of the standard C library functions `frexp()` and `ldexp()`. Subsequently, we can reconstruct the original 64-bit representation with a dot-product:

$$x^{(64)} = X^{(64)} \times [10^0, 10^{-16}, 10^{-32}, 10^{-48}]^t \quad (4)$$

Note that this is a floating-point equivalent of fixed-point storage of multi-byte integers whereby the least-significant

byte stores the lowest 8 bits of the integer. The second to the least-significant byte stores the range of bits $9 \div 16$, and so on.

Due to the way the NVIDIA hardware is organized, computing on four vectors of our proposed partitioning has the same computational load as it would have had for a single vector. At the same time, global GPU memory loads use 128-byte transactions and so, with suitable storage, it might in fact have the same bandwidth cost for either one or four vectors provided careful coding is used. Exploiting these hardware structures is a strong motivating factor for adopting our proposed adaptive residual scaling approach.

By comparison, Google’s Tensor Processing Unit (TPU) handles calculations in MXUs (matrix units) that are packaged in one-per-core arrangement (the cores themselves are, confusingly, also called Tensor Cores). The MXU accepts its input in 32-bit format and processes them internally in BFLOAT16. The unit of operation is a matrix-matrix multiply with a single instruction handling 128-by-128 matrices. In order to take advantage of such large dimensions, it is necessary to perform packing of right-hand side entries and replication of matrix entries so that each dot-product instruction has meaningful data in the entire MXU.

VII. COPING WITH LIMITED RANGE: NUMERICALLY BALANCED FACTORS WITH SIGNED SQUARE LU

In this strategy, we focus on line 1 of Algorithm 1 and introduce a new factorization variant of LU that is more suitable for floating-point storage format with limited range.

The traditional form of LU factorization with partial pivoting takes the form:

$$PA = LU \quad (5)$$

where $A, L, U \in \mathbb{R}^{n \times n}$ and $P \in \{0, 1\}^{n \times n}$ with L lower-triangular, U upper-triangular, and P a permutation matrix, respectively. This method of approaching the linear system solve given by Eq. 1 follows the commonly used decompositional approach to numerical matrix computations [22], [23]. The partial pivoting in the LU factorization, represented by the P matrix, maintains L well conditioned (to the extent possible) and transfers the unbound pivot growth into U . In fact, the majority of implementations commonly push this one step farther and produce matrix U that could be singular if A is singular in exact arithmetic, i.e., $\kappa(A) = \infty$, or only singular in the working precision wp : $\kappa(A^{(wp)}) = +\text{Inf}$. This approach works well in a practical context because it clearly informs the user about numerical issues with the input matrix A . But in cases when A is well conditioned, in particular when $\kappa(A^{(wp)}) \ll 1/u^{(wp)}$ (for FP64, $u \approx 10^{-16}$), U might still receive excessively large entries while L would be computed with unitary diagonal and subdiagonal entries smaller than 1 in magnitude. This imbalance has to be fixed for lower-precision factorization, and we propose to amend the standard formulation with an algorithm that is less prone to floating-point overflow.

There are already two factorizations that treat the diagonal more flexibly than LU: one is LDU and the other is LDLT. The

Algorithm 2: Vector-oriented formulation of the signed squared root LU for a square matrix A of dimension n and produces lower- and upper-triangular matrices L and U and a permutation matrix P .

```

1  $P \leftarrow \mathbb{I}_{n \times n}$ 
2 for  $i = 1 \dots n - 1$  do
3    $\text{pivot} \leftarrow \arg \max_{1 \leq k \leq n} |A_{i,k}|$       {find a pivot}
4    $A_{\text{pivot}, \dots} \leftrightarrow A_{i, \dots}$       {swap current with the pivot row}

5    $P_{\text{pivot}, \dots} \leftrightarrow P_{i, \dots}$       {record the pivot}
6    $L_{i+1 \dots n, i} \leftarrow A_{i+1 \dots n, i} / \sqrt[n]{|A_{i,i}|}$       {scale  $L$ }
7    $L_{i,i} \leftarrow \sqrt[n]{|A_{i,i}|}$       {record  $L$  scaling}
8    $U_{i,i+1 \dots n} \leftarrow A_{i,i+1 \dots n} / \sqrt{|A_{i,i}|}$       {scale  $U$ }
9    $U_{i,i} \leftarrow \sqrt{|A_{i,i}|}$       {record  $U$  scaling}
10   $A_{i+1 \dots n, i+1 \dots n} \leftarrow A_{i+1 \dots n, i} \times A_{i,i+1 \dots n}$  {Schur's complement}

```

former is for non-symmetric matrices while the latter is for the symmetric ones. We cannot adopt these directly because they separate the diagonal without applying it to the trailing matrix. Thus, they cannot reap the benefits of scaling the matrix entries and avoiding overflow in lower-precision arithmetic. Because we require the scaling by the diagonal, we need a nearly symmetric decomposition of the diagonal entry. We do this by proposing a signed square root operation:

$$\sqrt[n]{x} \stackrel{\text{def}}{=} \text{sign}(x) \sqrt{|x|} \quad x \in \mathbb{R} \quad (6)$$

This allows us to represent any real number¹ as a product of two numbers of equal magnitude and possibly different signs: $\mathbb{R} \ni x = \sqrt[n]{x} \sqrt{|x|}$. We will use this property to scale both lower and upper matrix entries symmetrically up to a sign. Algorithm 2 shows our proposed signed square root LU algorithm for square matrices. Deriving its generalizations for rectangular matrices is trivial. The algorithm also admits a block outer product and recursive formulations. Just like the classic LU factorization, the new algorithm can be implemented in an in-situ fashion and with extra $\mathcal{O}(n)$ storage to represent the permutation matrix that is fully compatible with the classic LU.

VIII. EXPERIMENTAL RESULTS

First, we present two reference sets of results to gauge the behavior of our proposed improvements. We measure the accuracy of the factorization by computing the residual norm: $\|r\|_\infty \equiv \|Ax - b\|_\infty$ while the number of digits can be estimated with $\log_{10} \|r\|_\infty$. Figure 3 shows a heat map of the number of digits achieved after 20 iterations of the refinement when both the factorization and storage of the factors uses 32-bit floating-point arithmetic. This is an established method [16], [17], [18] adopted by LAPACK. Bright red/orange colors in the figure indicate close to 15 accurate digits in the solution

¹Complex numbers naturally admit two square roots that are unambiguously defined and may be used for complex-domain LU factorization.

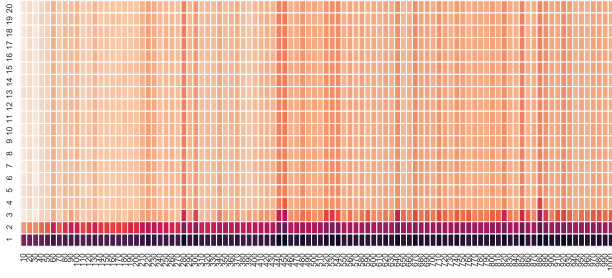


Fig. 3. Heat map of the number of digits of accuracy when performing iterative refinement using single-double precisions. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

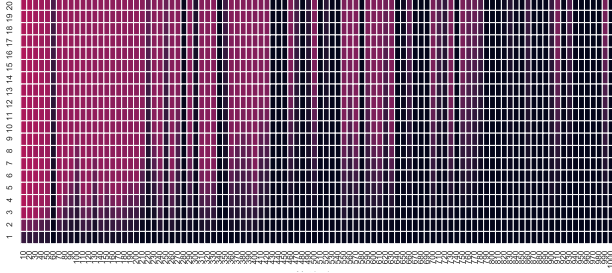


Fig. 4. Heat map of the number of digits of accuracy when performing iterative refinement using half-double precisions. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

while black indicates few digits: 2 or less. For matrices up to about 400, the third iteration delivers 15 digits of accuracy—a full precision result for 64-bit format even though factorization only used 32 bits. For matrices beyond the size of 400, a fourth iteration might be needed to get the full set of digits of accuracy. In Figure 4, we repeat the same experiment but with the factors and refinement limited to FP16. Very few digits are recovered for matrices under 200 and almost no digits beyond that. We proceed to the tests that seek to improve the accuracy by using the methods we proposed earlier. It is worth noting that the result from Figure 4 does not use any precision-extending modes that are available in both Tensor Core from NVIDIA or MXU from Google: the inputs, the outputs, and the intermediate computation uses 16-bit IEEE format.

We start with the strategy of promoting the entries of the L and U factors to single precision to temporarily extend the range of the computed values. We also use single precision to accumulate the results. Figure 5 shows the resulting heat map of accuracy. The number of recovered digits does not go as high as for the experiments with single-double iterative refinement presented in Figure 3. However, there are many matrices for which the method recovers many digits in the solution—much more, in fact, than when using half-precision alone.

We proceed by abandoning the use of single-precision altogether and we only use adaptive scaling to control the range of the entries in the solution. Figure 6 shows the heat

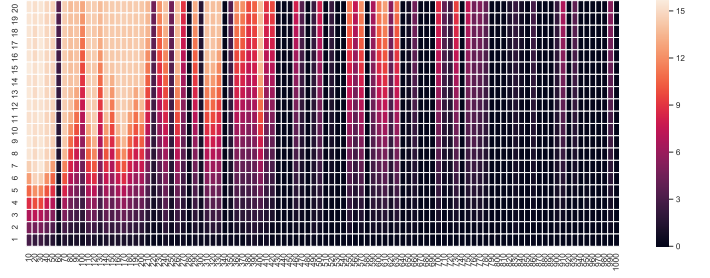


Fig. 5. Heat map of the number of digits of accuracy when performing iterative refinement using half-double precisions with single-precision promotion for the solves. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

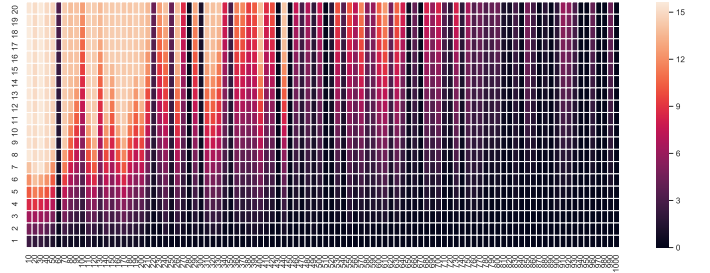


Fig. 6. Heat map of the number of digits of accuracy when performing iterative refinement using half-double precisions with adaptive scaling. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

map of the accurate digits in the solution. We observe that even though the single precision is gone, we manage to attain similar accuracy levels for the same set of matrices.

Next, we tested the mixed-precision iterative refinement when storing the solution vectors in a partitioned form and the heat of observed accuracy is presented in Figure 7. The results are in line with the prior two methods and we show again that it is possible to maintain high accuracy of the solution without the use of single precision as long as matrix conditioning in half precision is conducive to obtaining convergent iteration.

Finally, we present the accuracy results from running our new LU variant that deals with the limited range during the factorization by using the newly proposed signed square root

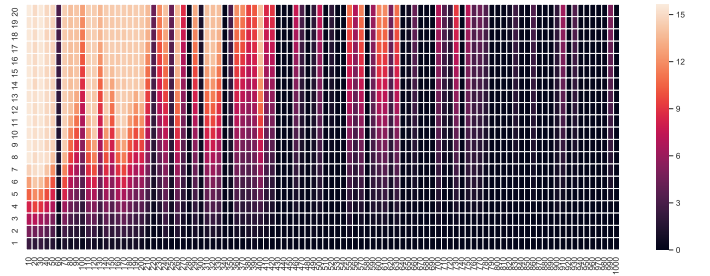


Fig. 7. Heat map of the number of digits of accuracy when performing iterative refinement using half-double precisions with partitioning of the mantissa bits. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

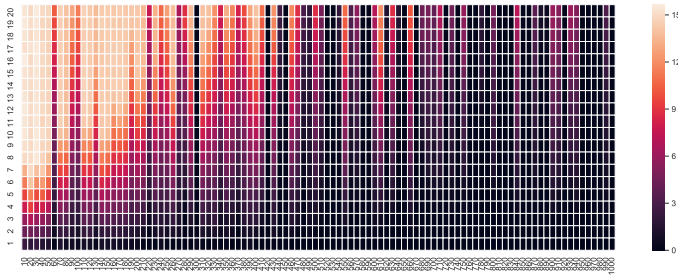


Fig. 8. Heat map of the number of digits of accuracy when performing iterative refinement using half-double precisions with signed square root LU factors. The horizontal axis represents the matrix size and the vertical axis shows the estimate of number digits of achieved accuracy: $\log_{10} \|Ax - b\|_{\infty}$.

operation. Figure 8 shows the heat map of accuracy across the same matrix dimensions as was presented before. The results are more accurate with more digits in the solution recovered during the iteration for a number of matrix sizes.

IX. CONCLUSIONS AND FUTURE WORK

We presented multiple methods that allowed us to recover some of the accuracy that would have been lost when using half-precision arithmetic in the mixed-precision refinement algorithm. We show how a number of range- and accuracy-enhancing strategies combined with the new variant of LU factorization, based on the newly proposed signed square root operation, greatly improve the correct digits in the linear system solve when compared with the original mixed-precision refinement that uses half precision for LU factorization.

In the future, we would like to look into more exotic accelerator hardware and see how our proposed methods extend the usability of these platforms to numerical linear algebra. Further modifications of the original mixed-precision iteration refinement algorithm are also possible and will be the subject of future research.

ACKNOWLEDGMENTS

This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. It was also partially supported by the National Science Foundation through OAC-1740250.

REFERENCES

- [1] Y. Chen, J. S. Emer, and V. Sze, “Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks,” *CoRR*, vol. abs/1807.07928, 2018. [Online]. Available: <http://arxiv.org/abs/1807.07928>
- [2] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: <http://arxiv.org/abs/1704.04760>
- [3] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Princeton, NJ, USA: Prentice-Hall, 1963.
- [4] —, *The Algebraic Eigenvalue Problem*. London, UK: Oxford University Press, 1965.
- [5] G. Peters and J. H. Wilkinson, “On the stability of Gauss-Jordan elimination with pivoting,” *Communications of the ACM*, vol. 18, pp. 20–24, 1975.
- [6] J. R. Hauser, “Berkeley SoftFloat,” 2018, <http://www.jhauser.us/arithmetic/SoftFloat.html>.
- [7] —, “Handling floating-point exceptions in numeric programs,” *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 2, pp. 139–174, March 1996.
- [8] J. Lee, G. D. Peterson, R. J. Harrison, and R. J. Hinde, “Mixed precision dense linear system solvers for high performance reconfigurable computing,” ser. 2009 Symposium on Application Accelerators in High-Performance Computing (SAAHPC’09), University of Illinois at Urbana-Champaign, USA, 2009.
- [9] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, “Mixed precision iterative refinement techniques for the solution of dense linear systems,” Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester, Tech. Rep. MIMS EPrint: 2007.124, 2007, iISSN 1749-9097, Reports available from: <http://www.manchester.ac.uk/mims/eprints>.
- [10] E. Carson and N. J. Higham, “A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems,” *SIAM J. Sci. Comput.*, vol. 39, no. 6, pp. A2834–A2856, 2017.
- [11] A. Haidar, S. Tomov, J. Dongarra, and N. 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*, ser. SC ’18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3291656.3291719>
- [12] N. J. Higham, S. Pranesh, and M. Zounon, “Squeezing a matrix into half precision, with an application to solving linear systems,” 2018, mIMS Preprint.
- [13] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” *CoRR*, vol. abs/1502.02551, 2015, accessed: 2018-08-01. [Online]. Available: <http://arxiv.org/abs/1502.02551>
- [14] —, “Deep learning with limited numerical precision,” in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 2015, pp. 1737–1746, accessed: 2018-08-01. [Online]. Available: <http://proceedings.mlr.press/v37/gupta15.html>
- [15] G. W. Howell and M. Baboulin, “LU preconditioning for overdetermined sparse least squares problems,” in *Proceedings of Parallel Processing and Applied Mathematics (PPAM) 2015*, Lublin, Poland, 2015, to appear.
- [16] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, “Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems),” in *ACM/IEEE SC 2006 Conference (SC’06)*, Nov. 2006, p. 50.
- [17] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, “Accelerating scientific computations with mixed precision algorithms,” *Computer Physics Communications*, vol. 180, pp. 2526–2533, 2009.
- [18] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, *High Performance Computing and Grids in Action*. IOS Press, Amsterdam, Nov. 2007, ch. Exploiting Mixed Precision Floating Point Hardware in Scientific Computations.
- [19] G. Peters and J. H. Wilkinson, “The least-squares problem and pseudo-inverses,” *Comput. J.*, vol. 13, pp. 309–316, 1970.
- [20] A. Edelman, “Eigenvalues and condition numbers of random matrices,” Ph.D. dissertation, Massachusetts Institute of Technology, May 1989.
- [21] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: Past, present, and future,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, August 10 2003, doi: 10.1002/cpe.728.
- [22] G. W. Stewart, *Matrix Algorithms: Volume 1: Basic Decompositions*. SIAM, 1998.
- [23] —, *Matrix Algorithms: Volume II: Eigensystems*. SIAM, 2001.