Harnessing a GPU's Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers

Azzam Haidar*, Stanimire Tomov*, Jack Dongarra*†‡ Nicholas J. Higham §
{haidar|tomov|dongarra}@icl.utk.edu,
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA
†Oak Ridge National Laboratory, USA
‡University of Manchester, UK
§School of Mathematics, University of Manchester, UK

***Abstract*—The use of low-precision arithmetic in computing has been a powerful tool for accelerating numerous scientific computing applications—and artificial intelligence in particular. Here, we present an investigation showing that other high-performance computing (HPC) applications can also harness this power. Specifically, we use the general HPC problem, $Ax = b$, where A is a large dense matrix, and a double precision (FP64) solution is needed for accuracy. Our approach is based on the mixed-precision (FP16→FP64) iterative refinement technique, and we generalize and extend prior advances into a framework, for which we develop architecture-specific algorithms and highly tuned implementations. These new methods show how using half-precision Tensor Cores (FP16-TC) for the arithmetic can provide up to 4× speedup. This is due to the performance boost that the `FP16-TC` provide and to the improved accuracy, which outperforms the classical FP16 because the GEMM accumulation occurs in FP32-bit arithmetic.**

***Index Terms*—FP16 Arithmetic, Half Precision, Mixed Precision Solvers, Iterative Refinement Computation, GPU Computing, Linear Algebra**

I. Introduction

To take advantage of new processor designs, algorithms must also be redesigned. This is especially true and challenging in the area of dense linear algebra, where many algorithms are expected to run close to the machine's peak performance. For example, LINPACK was redesigned to move away from using vector algorithms that were useful on the vector machines of the 1970s to the new Linear Algebra PACKage (LAPACK) that uses blocked algorithms on cache-based processors. LAPACK itself had to be redesigned for multi-core and heterogeneous many-core architectures, which resulted in the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [20].

To this end, this paper discusses the redesign of a mixed-precision iterative refinement technique to harness the fast `FP16-TC` arithmetic available in the latest NVIDIA GPUs. Modern architectures are trending toward multiple floating-point arithmetic precisions being supported in the hardware, and lower precisions are often much faster than higher precisions. For example, single-precision, 32-bit floating-point arithmetic (`FP32`) is usually twice as fast as double-precision, 64-bit floating-point arithmetic (`FP64`). Recently, various machine learning and artificial intelligence neural network applications increased the need for `FP16` arithmetic (see Figure 1), and vendors started to accelerate it in hardware. Currently, the the V100 TCs can accelerate `FP16` up to 85 teraFLOP/s—vs. 7 teraFLOP/s for `FP64` and 14 teraFLOP/s for `FP32` on a V100 through PCIe. Developing algorithms to use this hardware efficiently will be highly beneficial in HPC.

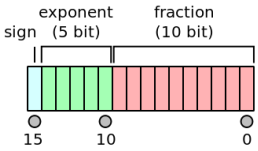


Fig. 1. IEEE 754 `FP16` format. This representation has 3.311 decimal digits of accuracy and a maximum representable value of 65,504.

The mixed-precision iterative refinement technique is used to accelerate $Ax = b$ solvers, where A is a dense matrix. The main idea is to compute the lower-upper (LU) factorization of A in low precision and use that factorization as a preconditioner in an iterative refinement in higher-precision. These methods have been studied in the past, as discussed in Section II. A persistent challenge has been to redesign the techniques for new architectures and to develop highly tuned implementations that resolve computational issues like inefficient parallelization, scaling, and use of mixed-precision calculations. A lot of the theoretical work with numerical experiments on small problems has been restricted on MATLAB or reference implementations, which are prone to overlook computational issues when achieving acceleration using highly tuned standard solvers. To address this problem on GPU Tensor Cores, we leverage building blocks from the MAGMA library, which provides state-of-the-art, high-performance algorithms like LU and others—including a set of highly tuned mixed-precision iterative algorithms for `FP32` →`FP64` arithmetic [21].

II. Related Work

Iterative refinement is a well-established technique that dates back to Wilkinson in the 1940s. The idea is to improve the solution of a linear system by solving the correction equation and adding the correction to the original (see Wilkinson [22], Moler [14], Stewart [19], and Demmel [6]). For more comprehensive treatment see Chapter 12 in Higham [11]). In iterative refinement, the three tasks (original solve/factorization, residual computation, and correction equation solve) can be done in the same precision (fixed precision) or in different precisions (mixed precision). Fixed precision iterative refinement was analyzed by Skeel [18] for an LU solver and extended by Higham [10] for a general solver. In the 2000s, motivated by processors equipped with `FP32` speed $2\times$ that of `FP64`, mixed precision iterative refinement—where the heavy LU factorization done in `FP32` and everything else done in `FP64`—was explored in [3], [12].

Replacing the direct triangular solve of the correction equation with an iterative method leads to "nesting" of two iterative methods, which are also called "inner–outer" iterations, and has been studied both theoretically and computationally [8], [15], [17], as well as used in mixed-precision computation scenarios [2]. Recently, Carson and Higham [4], [5] analyzed the convergence property of a three precision iterative refinement scheme (factorisation precision, working precision, residual precision) and concluded that if the condition of A is not too bad, $\kappa_\infty(A) < 10^4$, then using `FP16` for the $O(n^3)$ portion (e.g., the LU factorisation) and the (`FP32`, `FP64`) or the (`FP64`, `FP128`) as the (working, residual) precision for the $O(n^2)$ portion (refinement loop), one can expect to achieve forward and backward error on the order of 10^{-8} and 10^{-16} respectively. The same study also showed that when using the Generalized Minimal RESidual (GMRES) preconditioned by the `FP16` LU as the refinement procedure, the constraint on the condition number can be relaxed to be $\kappa_\infty(A) < 10^8$ when the (working, residual) precision is (`FP32`, `FP64`) and to 10^{12} when the (working, residual) precision is (`FP64`, `FP128`). The study provided error analysis of the iterative refinement algorithms, where the authors derived conditions for convergence and bounds for the attainable norm-wise forward error and norm-wise and component-wise backward errors using such techniques.

An investigation of similar iterative refinement methods on earlier generations of GPUs can be found in [9]. With the announcement of NVIDIA's V100 Tensor Cores that improve numerical precision and speed for `FP16`, it is our intention to comprehensively investigate how the V100 opens a new world of opportunities in matrix computations.

III. Contributions

The primary goal of this paper is to propose and implement a high-performance framework for the mixed-precision iterative refinement technique that makes use of GPU Tensor Core-accelerated `FP16-TC`. To this end, we will:

- introduce a new class of multi-precision dense matrix factorization algorithms; and here we mean that the factorization itself is implemented in multi-precision despite the fact the iterative refinement is mixed precision.
- develop a framework for exploiting GPU TCs in mixed-precision (`FP16-FP32`/`FP64`) iterative refinement solvers and describe the path to develop high-performance, Tensor Cores-enabled dense linear algebra building blocks kernels that can be used to exploit the FP16-TC in HPC applications;
- present a study on algorithmic variants of the IR techniques;
- illustrate that a number of problems can be accelerated up to $4\times$ through the mixed-precision solvers using fast `FP16-TC`, $3\times$ using the basic `FP16` mixed-precision solver, or $2\times$ using the `FP32` arithmetic;
- provide an analysis of the numerical behavior of the proposed mixed-precision, TC-accelerated solvers on different types of matrices; and
- quantify—in practice—the performance and limitations of this approach on V100 GPUs using TC.

The developments will be released through an open-source library to make these experiments independently reproducible and to allow the scientific community build and study different type of research on the top of this work.

IV. Methods

We consider two methods to extract higher precision solutions from low-precision factorizations. The first method is the standard iterative refinement (`IR`) as implemented in LAPACK, and the second method is iterative refinement using GMRES for the correction equation that we denote by `IRGM`. The `IR` method is a well-established technique, while `IRGM` with GMRES is more recent and holds more promise for exploiting low precision in factorizations.

A. Background

Below we expound on the `IR` and `IR` with GMRES methods.

1) Iterative Refinement: The `IR` technique improves the accuracy of a computed solution, $\hat{x}$, for a linear system of equations, $Ax = b$. Here, we let the initial solution be $x_0 = 0$, and the iterative refinement is a series of iterations:

1) Residual computation: Compute the residual $r = b - Ax_i$.
2) Correction equation: Solve $Ac = r$ (e.g., using an initial LU factorization).
3) Solution update: Correct the current solution $x_{i+1} = x_i + c$.

If all three steps can be computed exactly, then the `IR` algorithm completes in one iteration. However, with the finite precision in floating point format and arithmetic, the above iterations must be repeated.

`IR` is usually carried out with Gaussian elimination with partial pivoting. In this context, the correction equation is solved by the Gaussian elimination's LU factors. We call the precision $\boldsymbol{u}$ (machine epsilon) in which steps 1 and 3 are carried out. Step 2 is performed in the precision u_f which

is the precision of the LU factorization, because it uses the "L" and "U" factors to solve the correction equation $Ac = r$ and then it cast the solution "c" to the working precision $\boldsymbol{u}$. If the Gaussian elimination is also computed in the precision $\boldsymbol{u}$, the method is called fixed-precision IR, otherwise if the Gaussian elimination is performed at lower precision $\boldsymbol{u}_f$, it is called mixed-precision IR. The fixed-precision IR can be used to improve the backward error of a Gaussian elimination without a strong stabilizing pivoting strategy [1], [7], [13], [18]. On the other hand, the mixed-precision IR also improves the forward error to the working precision—if the condition number of A is not too large: $\boldsymbol{u}_f \kappa_\infty(A) \leq 1$.

The economics of mixed-precision IR using low-precision LU, compared to directly solving with higher precision LU, depend on the relative speed of low-precision arithmetic and on the cost of the refinement process. It is therefore a function of the executing hardware, the IR configuration, and also of the property of the matrix A (condition number). In the specific case of the V100 GPU, the practical speed of FP16-TC is about 12× faster than FP64 for square Xgemm and is about 6× faster for the rank-k update Xgemm that is used by the LU factorization (see Figure 2a) which pushes the balance in favor of mixed-precision IR.

2) Iterative Refinement with Preconditioned GMRES: GMRES [16] is a popular Krylov subspace iterative solver for solving a general linear system of equations. The Stopping criterion of GMRES can be defined by the user to a certain value, and the obtained solution is guaranteed to be down to that value in the normwise backward error stability. For that, we decided to propose another variant of the iterative refinement method by replacing step "2" of the classical IR algorithm described above by a preconditioned GMRES to solve the correction system $Ac = r$. GMRES will be preconditioned by the low precision LU meaning. Our idea here is that, the GMRES solver will provide a better and more stable solution to $Ac = r$ than the basic triangular solve which is directly affected by the quality of the LU factors, in particular when the matrix A is ill-conditioned or in other term when the LU factorization is highly perturbed due to rounding error caused by the low precision. Using GMRES, we can still guarantee that the solution of the correction equation $Ac = r$ is down to the stopping criterion accuracy requested by the algorithm. The stopping criterion is chosen to be the same as the low precision arithmetic used during the factorization (e.g., we use 10^{-4} or 10^{-8} for when the LU is in FP16 or FP32 respectively). Since the purpose of this paper is rather about practical usage and possible performance gain than about error analysis, we guide the reader that [4], [5] provide a detailed error analysis study about the IR and IRGM techniques.

We describe both the IR and IRGM methods in Algorithm 1 in a unified framework, where the difference between them is in the correction equation solver (LU or preconditioned GMRES).

Data: An $n \times n$ matrix A, and size n vector b.
Result: A solution vector $x^{(i)}$ approximating x in $Ax = b$, and an LU factorization of $A = LU$.
(FP16) Solve $Ax^{(1)} = b$ using FP16 LU factorization and triangular solve;
$i \leftarrow 1$;
repeat
  (FP64) Compute residual $r^{(i)} \leftarrow Ax^{(i)} - b$;
  (Low Precision) Solve $Ac = r^{(i)}$ using
    IR: triangular solve using the LU factors, or
    IRGM: GMRES preconditioned by $M = LU$;
  (FP64) Update $x^{(i+1)} = x^{(i)} - c$;
  $i \leftarrow i + 1$;
until $x^{(i)}$ *is accurate enough*;

Algorithm 1: IR: mixed-precision iterative refinement using triangular solve. IRGM: mixed-precision iterative refinement with GMRES to solve correction equation.

B. Algorithmic Advancements

A recent study by Carson and Higham [4] provides an analysis of using three precisions for the IR iterations: FP32 by default, FP16 for the LU factorization, and FP64 for the residual computation (Table I). For a matrix with a condition number of $\kappa_\infty(A) < 10^4$, the IR converges to FP32 accuracy. Furthermore, if we replace the correction step (e.g., the direct LU triangular solve of step 2 of the IR algorithm in Section IV-A1) by the GMRES is preconditioned by the low-precision LU, the restriction on the condition number can be relaxed to $\kappa_\infty(A) < 10^8$. However, it remains unclear how fast GMRES converges. Inspired by the analysis of the aforementioned study and the performance potential of FP16 on the NVIDIA V100 GPUs, we developed a practical implementation of a similar IR variant with GMRES denoted by IRGM. The implementation is different from [4] in that:

- 1) we only use two precisions—FP16 for the LU factorization $O(n^3)$ work and FP64 for the high precision for everything else; This decision is driven by the work done by [3], [12] which studied and showed the feasibility of the two precisions iterative schema and by the complexity required to implement three precision algorithm in particular the quad precision.
- 2) we proposed a multi-precision factorization algorithms. Our LU algorithm uses both FP16 and FP32 precision during its progress. We only use FP16 matrix multiplication (hgemm) in the LU factorization, while everything else is in FP32—effectively making our LU multi-precision. This decision is driven by the idea to compute the numerical sensitive portion of the algorithm in higher precision to avoid underflow and overflow that can easily occur when operating in full FP16.

Type	Range	Unit Roundoff Error
FP16	$10^{\pm 5}$	5×10^{-4}
FP32	$10^{\pm 38}$	6×10^{-8}
FP64	$10^{\pm 308}$	1×10^{-16}

TABLE I
IEEE PRECISIONS AND NUMERICAL PROPERTIES.

C. Tensor Cores in the V100 and their Use in Half-Precision LU

Driven primarily by the need for training and inferencing in deep learning, the latest offering from NVIDIA—the Tesla V100 GPU based on the Volta architecture—provides specific programmable matrix multiply–accumulate units that are said to deliver a theoretical peak performance of 110 teraFLOP/s in FP16-TC. The V100 has 8 Tensor Cores per streaming processor for a total of 640 Tensor Cores. A Tensor Core can compute $D = A * B + C$ per clock, where all matrices are 4×4 in size (i.e., 64 floating point FMA mixed-precision operations per clock). The A, B must be in FP16, but the C, D can be in either FP16 or FP32. The multiplication occurs in FP16 and is accumulated in FP32 with other products. In our experiment, FP16-TC denotes the use of the Tensor Core FP16 routine. We expect significant improvement in numerical behavior over the basic FP16 arithmetic, where all calculations are rounded and accumulated in FP16. The Tensor Core caters primarily to deep learning applications, where lower precision is tolerable, and is not straightforward for use in more numerically demanding applications (e.g., numerical simulations and linear solvers in particular). Since TC provide a large speedup for matrix-multiplication, it can be expected to accelerate FP16 dense matrix factorizations, which are rich in matrix multiplication operations. The challenge lies in how to use the low-precision factorization results to achieve FP32/FP64 level accuracy effectively and efficiently.

D. Convergence Consideration

This subsection discusses the convergence rate of the IR and IRGM methods. Unfortunately, while some convergence conditions are known for classical IR—linearly convergent except for the hard case, where the LU factorization is not stable and the IR diverge—the convergence rate of IRGM is difficult to predict due to the GMRES behavior.

The latest analysis for mixed precision iterative refinement can be found in [4], [5]. It yields the sufficient condition for convergence $\kappa_\infty(A) < 10^4$ for IR. It would provide the sufficient condition for convergence $\kappa_\infty(A) < 10^8$ for IRGM. If quadruple precision were used in computing the residuals it can be even relaxed to $\kappa_\infty(A) < 10^{12}$. For GMRES-based solvers, the convergence rate of GMRES is complicated to analyze, and the low accuracy FP16 preconditioner means we have little knowledge about the preconditioned matrix. In general, for a normal matrix, A, the GMRES converges slower as the condition number of A increases. For a non-normal matrix, the convergence rate cannot be predicted by condition number alone. In practice, the convergence rate depends on the matrix type, the condition number, and the matrix size. Therefore, in the next section we decided to study our two proposed algorithms by covering matrices with different spectrum, sizes, and types.

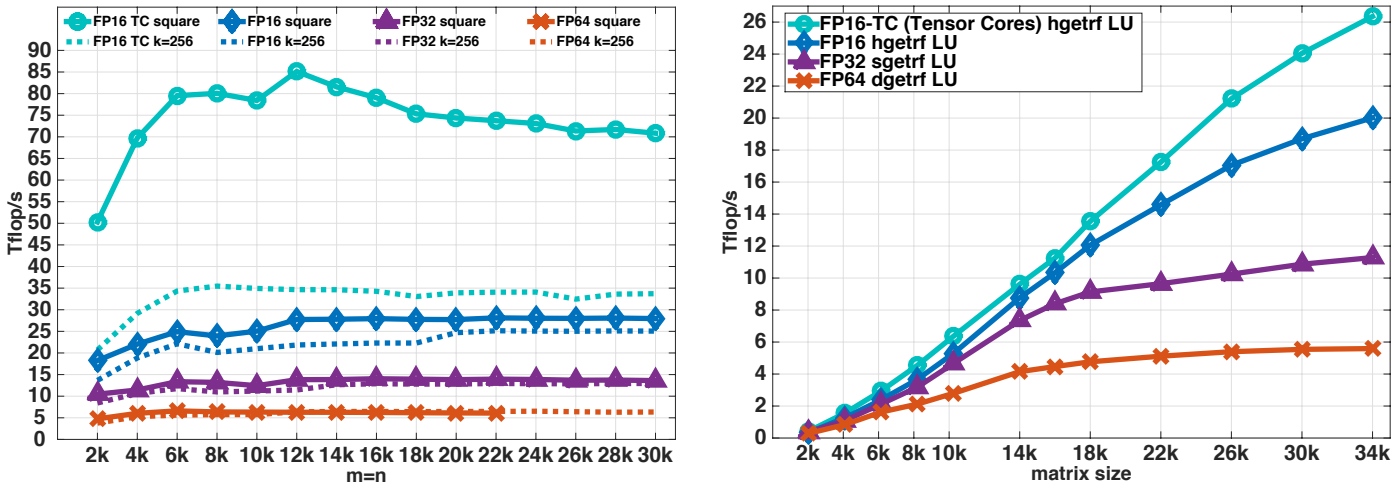
V. ANALYSIS AND EXPECTED PERFORMANCE

The primary motivation for using FP16 arithmetic is its unprecedented high performance compared to higher precisions. This performance is quantified for the V100 GPU in Figure 2. The PCIe V100 has a practical peak of 6.8 teraFLOP/s in FP64, 14 teraFLOP/s in FP32, 28 teraFLOP/s in FP16, and an incredible 85 teraFLOP/s in FP16-TC (Tensor Cores). The performance of the LU factorization relies mostly on the performance of the Schur update (or rank-k Xgemm update), which is a tall-skinny matrix multiplication that occurs during each step of the LU algorithm. For that, to understand/model how the LU factorization could benefit from the FP16 arithmetic, we first study the performance of the Xgemm routine. This is shown in Figure 2a for the four available precisions (FP64, FP32, FP16, and FP16-TC). We consider the FP16-TC as a separate precision, since it consists of a mixed-precision Xgemm, where the multiplication is performed in FP16, while the accumulation is in FP32. Thus, FP16-TC is more accurate than a homogeneous FP16 computation. We also note that, in addition to being more accurate, the FP16-TC is also much faster due to the use of Tensor Cores.

As shown in Figure 2a, the FP16-TC hgemm-TC operating on square matrices is about 12× faster than its FP64 dgemm counterpart. Furthermore, and as expected, the FP16 hgemm is 2× faster than the FP32 sgemm and about 4× faster than the FP64 dgemm. Figure 2a also depicts the performance of the Xgemm for a rank-k update (dashed lines). This is the type of operation needed by the LU and thus it give us an indicator of the performance ceiling/bound of any LU implementation. The rank-k update hgemm-TC is slower than the square hgemm-TC but still carries an attractive speedup compared to the dgemm. The rank-k hgemm-TC achieves about 35 teraFLOP/s; compare that to about 25 teraFLOP/s for the rank-k hgemm, 13 teraFLOP/s for the rank-k sgemm, and around 6 teraFLOP/s for the rank-k dgemm.

We also developed a multi-precision LU factorization in FP16, where we performed the numerically sensitive portion of the code in FP32 and only the GEMM's are in FP16 or FP16-TC. The idea here is to provide a better stable LU factorization than the fully FP16 implementation without loss of performance. Figure 2b shows the performance for the four precisions. As expected, our LU implementation follows roughly the same trend as the Xgemm kernel for large n which prove that our implementation is very well optimized and is able to attain the theoretical upper bound. Our hgetrf-TC and hgetrf solvers achieve a speedup from 3× to 4× over the dgetrf and a 2× speedup over sgetrf.

This section is dedicated to the theoretical performance analysis of the mixed precision (MP) algorithms for linear solvers. The idea is to understand and predict when iterative refinement techniques can be used in a beneficial fashion.



(a) Performance of the rank-k update (`Xgemm` function) used in `Xgetrf`.

(b) Performance of the `Xgetrf` routine.

Fig. 2. Performance of the three arithmetic precisions obtained on a Nvidia V100 GPU.

From a performance point of view, an algorithm is beneficial when it reaches the solution in a time faster than the reference one (which is the FP64 `dgesv` routine in our case). The iterative refinement solvers consist of an LU factorization in low precision $\varepsilon_{FPXX} < \varepsilon_{FP64}$, and then, iterative loop based on either classical IR or GMRES (as described above) can be used to improve the solution to ε_{FP64} which is comparable to the reference LU_{FP64} arithmetic. Thus, let us define:

$$\text{time for FP64} = \frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dtrsv}} \quad (1)$$

$$\text{time for MP} = \frac{2n^3}{3P_{Xgetrf}} + k\left(\frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{Xtrsv}} + \xi\right) \quad (2)$$

where P denotes the performance of the corresponding routine, and k denotes the number of iterations required by the MP solver to achieve the double precision solutions and where ξ refer to the other negligible amount of work required by the iterative refinement such as norm computation, residual calculation, pivoting, synchronizations. From practical and experiment results, we found that ξ is negligible compared to the cost of the `dgemv` and the `Xtrsv`.

Based on the LU performance results provided in Figure 2b and on the benchmark of the `dgemv` and `Xtrsv` routine, we illustrate in Figure 3 the expected speedup of our three MP routines (e.g., `dhgesv-TC`, `dhgesv`, `dsgesv`) as function of the number of iterations k, where we can see how the performance vary with the increases of k. Usually, a small number of iterations is advantageous and can bring the highest performance while a large number of iterations is going to affect the performance.

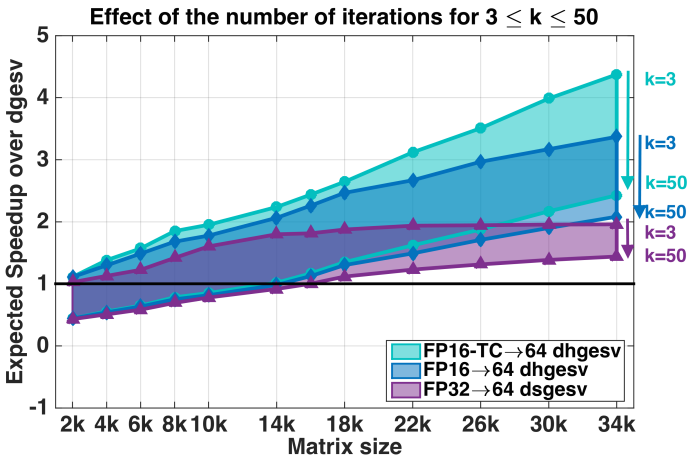


Fig. 3. Illustration of the speedup of the three MP routines (e.g., `dhgesv-TC`, `dhgesv`, `dsgesv`) over the `dgesv` routine as function of the number of iterations and the matrix size.

VI. Numerical Behavior Discussion

Our experiments were performed on a system with two 10-core Intel(R) Xeon(R) E5-2650 v3 CPUs (20 cores total) running at 2.30 GHz and one NVIDIA V100 PCIe GPU. To study the proposed methods and to highlight their practical use, we performed a large set of experiments on 21 types of matrices, with each type featuring different properties, that represent a wide range of problems. We found that we could classify the 21 types of matrices using 6 representatives cases. We first studied the numerical behavior of our iterative refinement algorithms (e.g., `dsgesv`, `dhgesv`, and `dhgesv-TC`) each using either the `IR` or the `IRGM` solver, and showed the convergence history of each technique for the different types of matrices.

This study aims to provide an analysis of each method's sensitivity relative to the matrix type as well as provide insight on the performance expected from the iterative refinement methods. For example, if an iterative refinement method requires a large number of iterations to achieve the `FP64` solution accuracy for a certain matrix type, then we could

expect that its performance compared to the standard dgesv will degrade or be even slower. We note that the number of iterations that we report is the total number of iterations—including the inner GMRES iterations in the case of the IRGM solver. This means that, in both cases, the number of iterations is precise indicator of the time spent in the refinement loop.

Table II describes the different matrix types and sizes used in our experiments.

Figures 4 and 5 show the convergence history of the six proposed solvers (the three precision implementations each using either IR or IRGM). They are labeled as FPXX→FP64 YY, where "XX" corresponds to the algorithm used for the LU factorization (FP16-TC, FP16, or FP32), and "YY" represents the iterative refinement solver (IR or IRGM) used to attain FP64 solution accuracy. In Figure 4a, we display the most numerically favorable type of matrix to solve—the diagonal dominant matrix. Here, we can see that all six variants converge in 3–5 iterations. For this type of matrix, since the number of iterations is small, we can expect a large speedup over the FP64 routine.

We believe that the FP32 routine will achieve a 2× speedup and that both of the FP16 routines will achieve about 3×–4× speedup while delivering a solution at FP64 accuracy. More details about the performance are provided in the next section. Figure 4b represents a matrix type that has positive eigenvalues and singular values for which the logarithms are uniformly distributed between 1 and $\frac{1}{cond}$. This is slightly more difficult than the diagonal dominant type. We observe that the convergence of the FP32 remains stable at 3 iterations, while the FP16 slightly increases to about 7–8 iterations. Interestingly, the FP16-TC converges faster than the FP16 and slightly slower than the FP32. This is because the accumulation in the FP16-TC rank-k update is in FP32 arithmetic and thus produces a better result than the FP16. This behavior can be seen on all graphs in Figures 4 and 5.

Figure 4c shows a more difficult type of matrix with clustered singular values and a positive eigenvalue. The FP32 method using either IR or IRGM converges as expected in 3 iterations. The FP16 IR variant converges very slowly and needs more than 400 iterations to drive the solution to 10^{-9} accuracy. However, the IRGM solver (FP16 IRGM) achieves the FP64 solution accuracy in about 14 iterations. This reveals the sensitivity of the FP16 IR variant and highlights the importance of using the GMRES solver inside the iterative refinement process. We note that GMRES delivers a more stable solution to $Ac = r$ inside the refinement process, which allows the IRGM method to converge faster than IR. We also note that the diamond marker in the blue dashed line—the curve that represents FP16 IRGM—shows the number of outer iterations (refinements) in the IRGM solver. We can see that the number of outer iterations is about 4, which also proves the theory that the solution of $Ac = r$ delivered by GMRES is good enough to make the refinement loop converge in 4 iterations. More details about using GMRES inside the refinement process can be found in [4]. The FP16-TC variants using either IR or IRGM converge in 4 iterations. This underlines the importance of the FP32 accumulation done in the hgemm-TC routine used in the FP16-TC variant. The FP16-TC variant works well for this type of matrix, and we can expect about a 4× speedup.

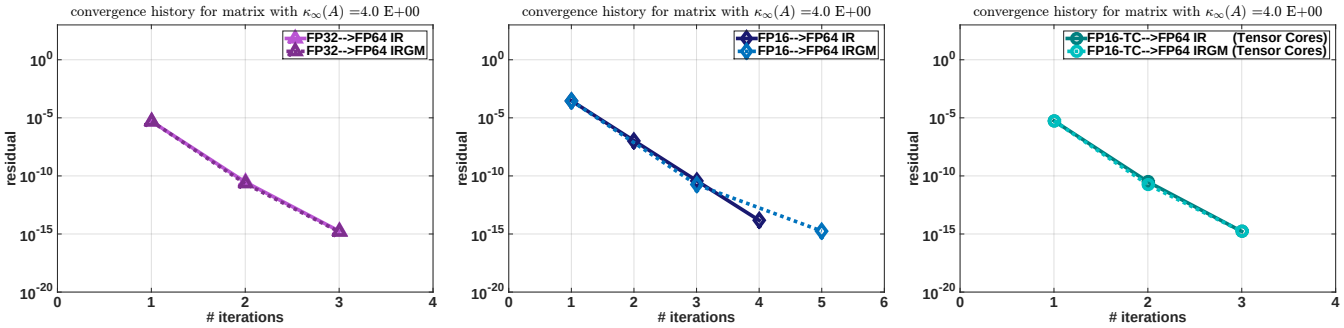
Figure 4d shows results on matrices with the same singular value distribution as in Figure 4c but with complex eigenvalues and of different positive and negative sign. When comparing the two figures, we notice the effect of having positive eigenvalues. We also note that the FP32 routine is not affected and converges in 4 iterations. The convergence of the FP16-TC routine decreases slightly, requiring about 13 iterations. The FP16 IR variant diverges, which highlights the problem with using trsv in the classical iterative refinement to compute the correction, as it becomes unstable when the LU factorization is performed in FP16 and cannot bring the solution down to an acceptable accuracy. While using GMRES, the FP16 IRGM can bring the solution down to the FP64 accuracy in about 63 iterations.

Figure 5a shows results on matrices where all methods behave well. Convergence was achieved in 3 iterations for FP32, 4 iterations for FP16-TC, and 7 iterations for FP16. Figure 5b has the same singular value distribution as Figure 5a but not necessarily positive eigenvalues. This type is the most difficult, and the FP16 variants using either IR or IRGM do not converge. Note that the FP16 IRGM can converge when allowing more than 2,000 iterations, but for our experiment we limited the max iterations to 400, since we have seen a large performance drop when iterations are around 200—where the iterative refinement becomes a drawback. The FP32 variants are not influenced by the matrix type and always converge in about 3–4 iterations. Surprisingly, the FP16-TC behaves very well and converges in about 18 iterations, while the FP16 did not converge.

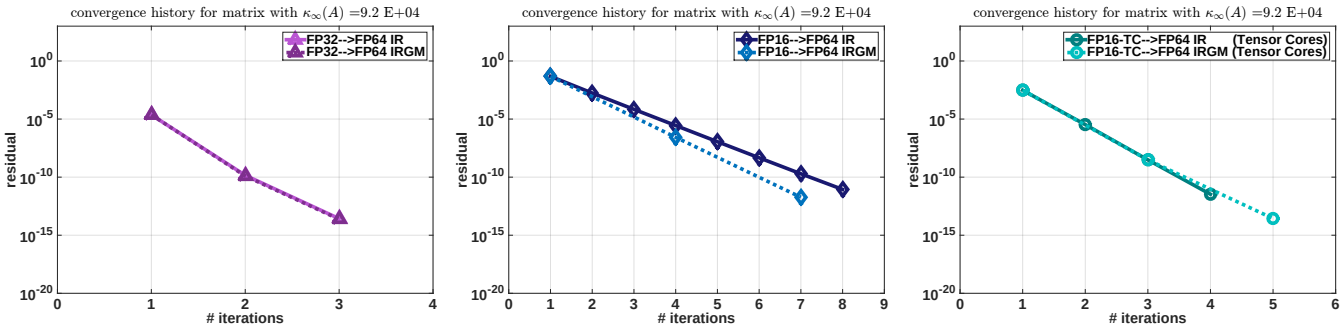
Lesson: For all the matrices considered, the FP16-TC variant is the most robust and fastest in convergence among the FP16 arithmetic-based methods. The FP32 refinement variants emphasize a consistent behavior regardless of the matrix types. This observation suggests the surprising effectiveness of the FP16 arithmetic, which might be robust enough to be used in HPC dense linear system solvers.

VII. Experimental Results Discussion

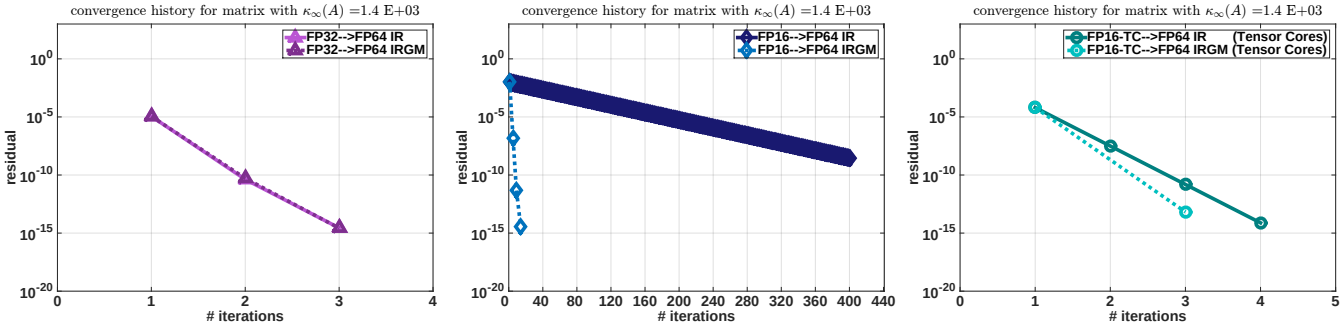
This section presents the performance results of our three iterative refinement methods—dhgesv-TC, dhgesv, and dsgesv—using either IR or IRGM, compared to the reference dgesv solver. We also depict the number of iterations required by each method to reach FP64 accuracy. The teraFLOP/s are computed based on the same formula ($P = \frac{2n^3}{3 \text{ time}}$), which means performance reflects the time to solution (e.g., if a method has 2× higher performance, it is 2× faster). The performance results are presented in Figures 6 and 7 for the six representative types of matrices studied in Section VI. In each figure, there are four performance curves that refer to the reference dgesv and to the three iterative refinement algorithms, dhgesv-TC, dhgesv, and dsgesv.



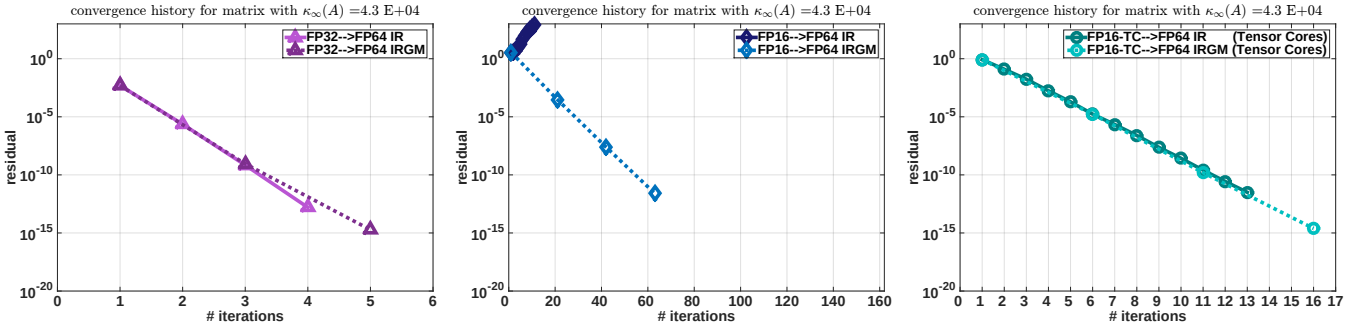
(a) Matrix of type 1: diagonal dominant.



(b) Matrix of type 2: positive λ, where σ_i is a random number between $\frac{1}{cond}$, and 1 such that their logarithms are uniformly distributed.

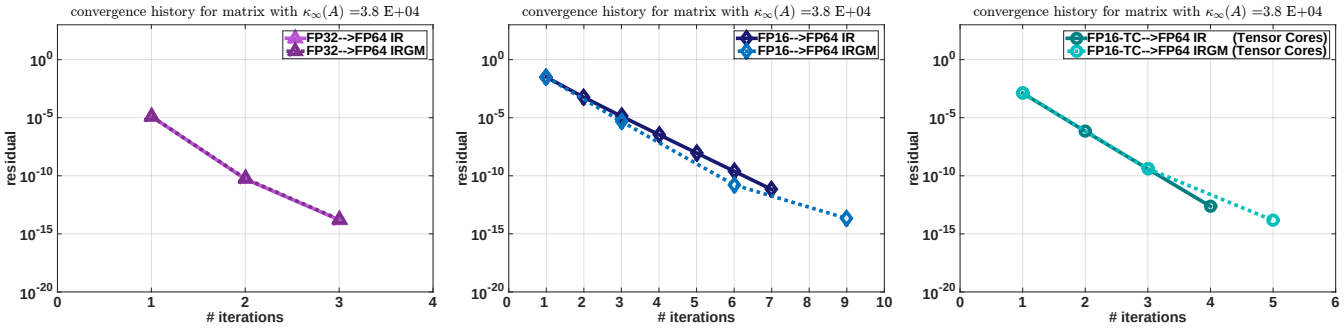


(c) Matrix of type 3: positive λ with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$).

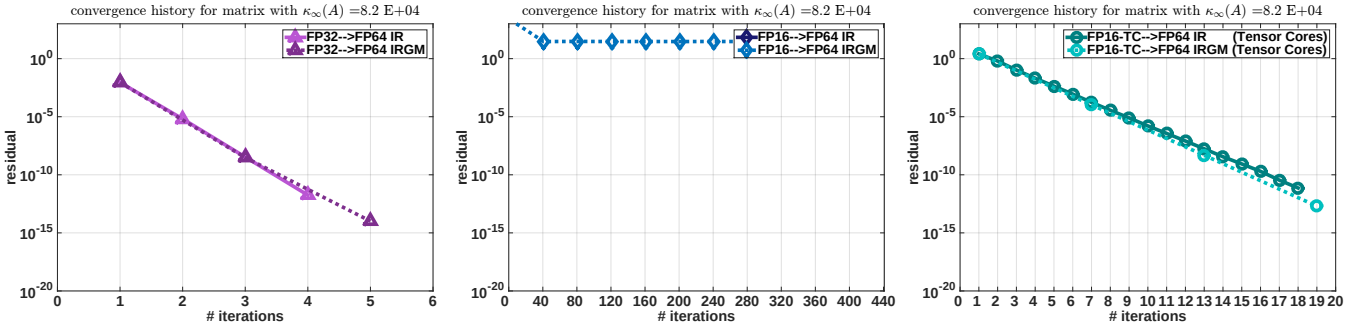


(d) Matrix of type 4: clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$).

Fig. 4. Convergence history of the two proposed iterative refinement algorithms (`IR` and `IRGM`) for the three precisions studied (`FP16-TC`, `FP16`, and `FP32`) and for different types of matrices—all of size 22,000 $\times$ 22,000, and having $\kappa_\infty(A)$ varying between 10^1 and 10^5.

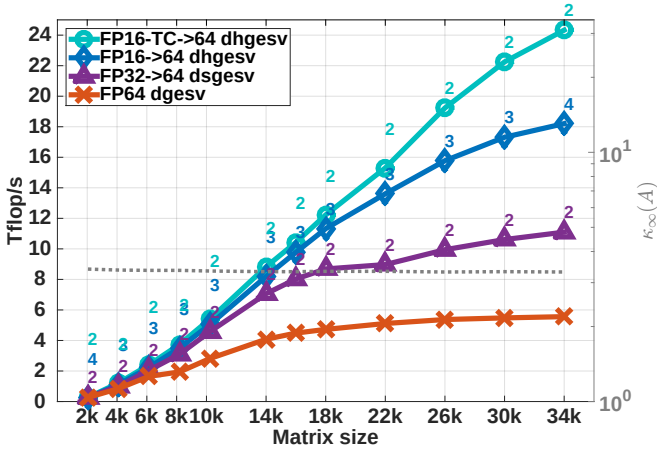


(a) Matrix of type 5: positive eigenvalues and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

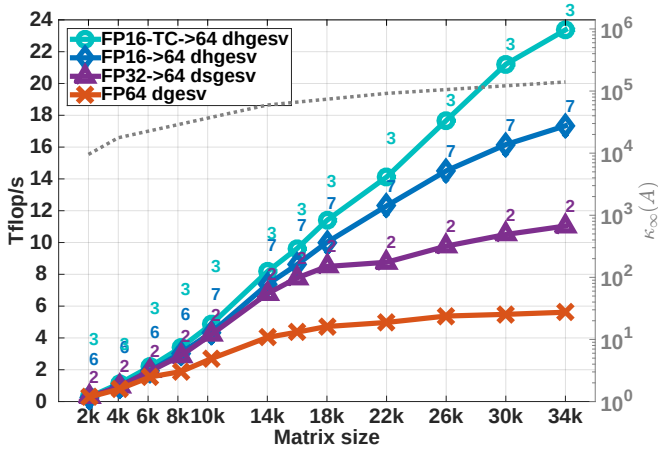


(b) Matrix of type 6: arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

Fig. 5. Convergence history of the two proposed iterative refinement algorithms (`IR` and `IRGM`) for the three precisions studied (`FP16-TC`, `FP16`, and `FP32`) and for different types of matrices—all of size 22,000 × 22,000, and having $\kappa_\infty(A)$ varying between 10^1 and 10^5.



(a) Matrix of type 1: diagonal dominant.

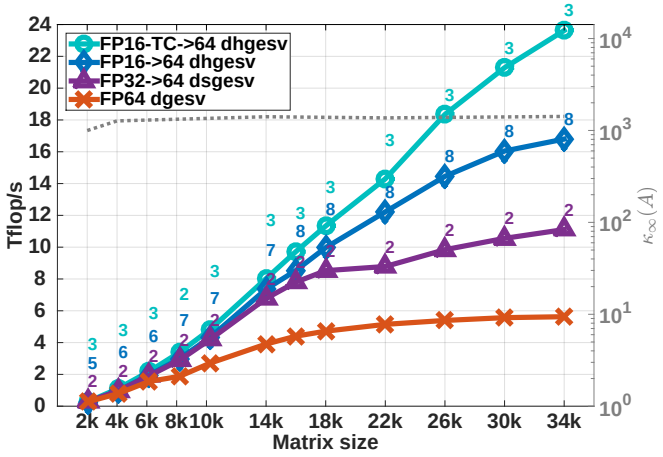


(b) Matrix of type 2: positive λ, where σ_i is random number between $\frac{1}{cond}$, and 1 is such that their logarithms are uniformly distributed.

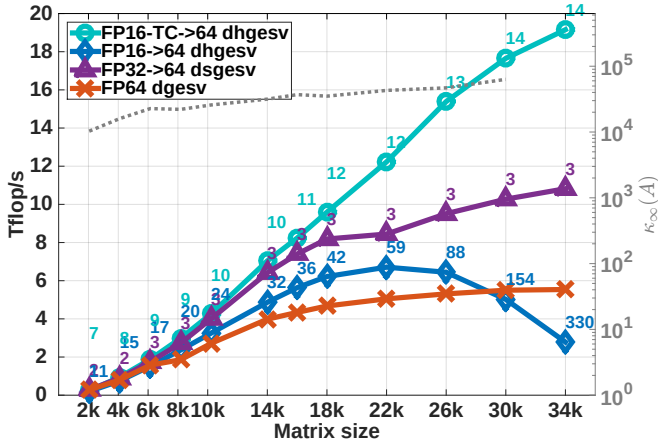
Fig. 6. Performance in teraFLOP/s of the four proposed linear solvers (one standard solver and three iterative refinement solvers, respectively) for different matrix sizes and different matrix types: 1) the `FP64` standard `dgesv` solver (orange color with "×"), 2) the `FP32` solver `dsgesv` (purple color with "△"), 3) the `FP16` solver `dhgesv` (blue color with "◇"), and 4) the `FP16-TC` solver `dhgesv-TC` (cyan color with "○"). For the iterative refinement solvers, we also depict the required number of iterations to achieve the `FP64` arithmetic solution, and we note that "nc" mean no convergence after 200 iterations. Note that the right "y" axis shows the condition number, $\kappa_\infty(A)$.

Type	Description		
1	-	Random numbers with diagonal modified to be dominant	
2	Positive eigenvalue λ	Random σ in $[\frac{1}{cond}, 1]$ such that their logarithms are uniformly	
3	Positive eigenvalue λ	Clustered σ	$\sigma = [1, \cdots, 1, \frac{1}{cond}]$;
4	-	Clustered σ	$\sigma = [1, \cdots, 1, \frac{1}{cond}]$;
5	Positive eigenvalue λ	Arithmetically distributed σ	$\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond}), i = 1..n$, $\frac{\sigma_{i+1}}{\sigma_i}$ is constant
6	-	Arithmetically distributed σ	$\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond}), i = 1..n$, $\frac{\sigma_{i+1}}{\sigma_i}$ is constant

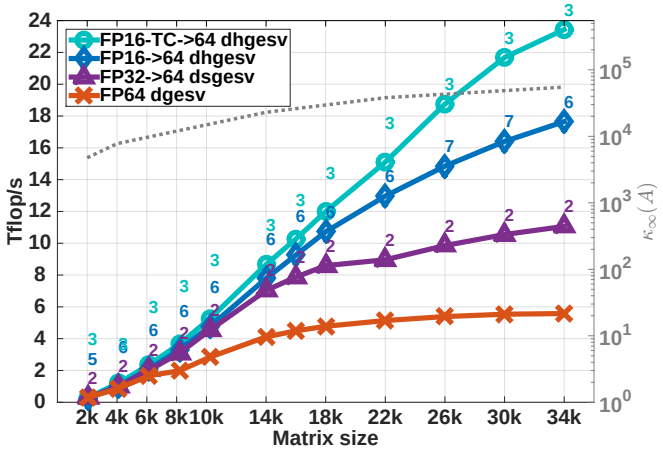
TABLE II
DESCRIPTION OF THE TEST MATRICES, WHERE COND IS THE CONDITION NUMBER OF A.



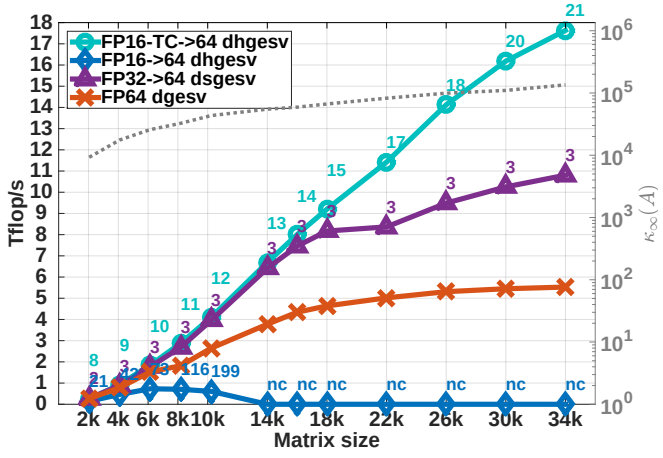
(a) Matrix of type 3: positive λ with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$).



(b) Matrix of type 4: clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$).



(c) Matrix of type 5: positive eigenvalues and arithmetic distribution of its singular values, $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.



(d) Matrix of type 6: arithmetic distribution of its singular values, $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

Fig. 7. Performance in teraFLOP/s of the four proposed linear solvers (one standard solver and three iterative refinement solvers, respectively) for different matrix sizes and matrix types: 1) the `FP64` standard `dgesv` solver (orange color with "×"), 2) the `FP32` solver `dsgesv` (purple color with "△"), 3) the `FP16` solver `dhgesv` (blue color with "◇"), and 4) the `FP16-TC` solver `dhgesv-TC` (cyan color with "○"). For the iterative refinement solvers, we also depict the required number of iterations to achieve the `FP64` arithmetic solution, and we note that "nc" mean no convergence after 200 iterations. Note that the right "y" axis shows the condition number, $\kappa_\infty(A)$.

In Figure 6a, the matrix is diagonally dominant, and—as shown in Section VI—all variants require three to five iterations to converge. Thus, one can expect that the low precision iterative refinement algorithms will bring a large speedup compared to dgesv. Since the number of iterations is small, we imagine that the speedup ratio will be similar to the one observed in Figure 2b for the LU factorization. We confirm our expectation by looking at Figure 6a. The FP16-TC dhgesv-TC routine delivers a solution 4× faster than its FP64 dgesv counterpart. Similarly, the dhgesv variant shows a ≈3× speedup over the dgesv, and the dsgesv variant shows a ≈1.8× speedup over the dgesv. This is example illustrates the importance of using the low FP16-TC precision in HPC.

Figure 6b shows the performance of our methods for matrices with positive eigenvalue and logarithmic uniform distribution of their singular values. As shown in the figure, and similar to the previous graph in Figure 6a, the number of iterations remains constant when the matrix size increases for all algorithms. This type of matrix is marginally more difficult for the FP16 variant than the diagonally dominant matrix type—about 7 iterations vs. 3 iterations (Figure 6a). The FP16-TC and FP32 variants require two to three iterations for both examples. Thus, one can expect the performance gain to be roughly similar to the diagonal dominant example. The dhgesv-TC (FP16 →FP64) variant results in a speedup of 4× over dgesv. The dhgesv (FP16 →FP64) routine achieves the same solution as the dgesv and is about 3× faster, while the dsgesv (FP32 →FP64) is about 1.7× faster.

Figure 7a supports our findings that low precision techniques can be used to speedup a linear solver by a large factor. The performance results depicted here are similar to the previous two examples, where dhgesv-TC, dhgesv, and dsgesv outperform dgesv and provide around 4×, 2.6× and 1.7× speedups, respectively. In contrast to Figure 7a, Figure 7b shows the performance and the number of iterations for matrices that have similar clustered singular value distribution, but their eigenvalues are not necessarily positive and can be even complex. The observation made here is interesting. The behavior of the dsgesv (FP32 →FP64) variant remains the same as in the previous experiments, requiring two to three iterations independent of the matrix size or matrix type. Thus, we will always see a 1.7× speedup. For the dhgesv-TC, the number of iterations increases compared to the previous examples (10–14 iterations vs. 3 iterations). We also see that the iteration count increases slightly with matrix size. Thus, one can expect the performance of the dhgesv-TC here to be slightly lower than in the previous example (while still being about 3× faster).

For the dhgesv, the number of iterations increases dramatically with the matrix size and is larger than what was depicted in Figure 7a. In this case, the rounding error of the FP16 method—and possibly the range of the representative numbers for FP16 arithmetic—disturb the LU factorization. As a result, the convergence rate decreases dramatically, which then affects the performance. In this case, we can see that dhgesv is not beneficial at all and can be slower than FP64. For such matrix types, the best practice is to use either the Tensor Core version of the dhgesv-TC, which provides a 3× speedup, or to use dsgesv, which yields a 1.7× speedup.

Figure 7c shows results for matrices with positive eigenvalues and an arithmetic distribution of their singular values. The dsgesv behavior stays the same as the one shown in the previous graph and requires about 2 iterations, resulting in a 1.7× speedup over dgesv. We also note that the dhgesv-TC routine acts similarly to dsgesv and converges in about 3 iterations, thereby making it an attractive routine to use in such cases, where it offers a speedup of around 4×. The dhgesv behavior is comparable to the other problem types with positive eigenvalues, requiring about 7 iterations, regardless of the matrix size. Thus, we obtain a speedup of around 2.6×. The results in Figure 7d are similar to those in Figure 7c but without the constraint of positive eigenvalues. Here, we note that the dsgesv still converges in about 3–4 iterations for any matrix size, leading to the observed 1.7× speedup, while the dhgesv fails to converge within 400 iterations for most of the large matrices, thereby making dhgesv useless in this case and validating our discussion of Figure 5b in Section VI. The attractive revelation is the FP16-TC implementation. The V100 GPU's FP16-TC feature does accumulation in FP32 arithmetic and is able to fix the issue of the FP16; in doing so, it converges in about 10–20 iterations, netting a speedup of around 3×.

Lesson: The speedups presented in Figures 6 and 7 confirm our numerical analysis from Section VI, where we noted that iterative refinement algorithms are advantageous and exhibit very good speedups. The dhgesv-TC (FP16-TC →FP64) routine can be used for all matrix types and for any matrix size to provide speedups of about 3×–4×, and the dsgesv (FP32 →FP64) routine can be used for all matrix types and for any matrix size to provide speedups of about 1.7×. The number of iterations is constant for all matrix types and matrix sizes for dsgesv. It is also constant for dhgesv-TC for matrices with positive eigenvalues and only slightly increases for other cases. The dhgesv (FP16 →FP64) is acceptable and provides a speedup of around 2.6× when the matrix has good properties (e.g., diagonal dominance) or when eigenvalues are always positive.

VIII. Conclusions and Future Directions

We developed a framework of algorithms and their high-performance implementations for exploiting GPU Tensor Cores in mixed-precision FP16–FP32/FP64 iterative refinement solvers. We demonstrated for the first time how to use the Tensor Core to provide an additional FP16-TC performance boost to solvers in high FP64 accuracy. We provided results and analysis for a number of algorithms on different types of matrices. Specifically, we showed practical cases where even a highly optimized FP64-precision solver, running at

6 teraFLOP/s, can be accelerated up to 4×. The new developments introduced a new class of mixed-precision dense matrix factorization algorithms that can be used as building blocks in other mixed-precision algorithms.

The developments opened up opportunities for few future work directions, including further optimizations, development of a full set of mixed-precision factorization routines, and release as open-source software. Also of interest is investigating the energy efficiency of the approach compared to working precision implementations and other architectures. One can expect that a 4X speedup will at least bring 4X energy improvement. In our experiments, we observed about 5X energy efficiency improvement but we omit the details of this improvement as we think it do not perfectly fit with the subject of this paper and we will be studying it in more details in future work.

REFERENCES

[1] M. Arioli, J. W. Demmel, and I. S. Duff. Solving Sparse Linear Systems with Sparse Backward Error. *SIAM Journal on Matrix Analysis and Applications*, 10(2):165–190, 1989.

[2] 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*, 180(12):2526–2533, 2009.

[3] 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. *Int. J. High Perform. Comput. Appl.*, 21(4):457–466, Nov. 2007.

[4] 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 Journal on Scientific Computing*, 39(6):A2834–A2856, 2017.

[5] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.*, 40(2):A817–A847, 2018.

[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[7] J. Dongarra, V. Eijkhout, and P. Łuszczek. Recursive Approach in Sparse Matrix LU Factorization. *Scientific Programming*, 9(1):51–60, 2001.

[8] G. H. Golub and Q. Ye. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing*, 21(4):1305–1320, 2000.

[9] A. Haidar, P. Wu, S. Tomov, and J. Dongarra. Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers. In *SC16 ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Denver, CO, 11/2017 2017. ACM, ACM.

[10] N. J. Higham. Iterative refinement enhances the stability ofqr factorization methods for solving linear equations. *BIT Numerical Mathematics*, 31(3):447–468, Sep 1991.

[11] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2 edition, 2002.

[12] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. J. Dongarraa. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[13] X. S. Li and J. W. Demmel. Making Sparse Gaussian Elimination Scalable by Static Piv oting 1 Introduction 2 New algorithm and stability. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, number c, pages 1–17, 1998.

[14] C. B. Moler. Iterative refinement in floating point. *J. ACM*, 14(2):316–321, 1967.

[15] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. Technical Report 91-279, Department of CSE, University of Minnesota, Minneapolis, Minnesota, 1991.

[16] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. and Stat. Comput.*, 7(3):856–869, 1986.

[17] V. Simoncini and D. B. Szyld. Flexible inner-outer Krylov subspace methods. *SIAM J. Numer. Anal.*, 40(6):2219–2239, 2002.

[18] R. D. Skeel. Iterative Refinement Implies Numerical Stability for Gaussian Elimination. *Mathematics of Computation*, 35(151):817–832, 1980.

[19] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[20] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.

[21] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010.

[22] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, 1963.