Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers

Azzam Haidar
University of Tennessee, Knoxville
Knoxville, TN
haidar@icl.utk.edu

Panruo Wu
University of Tennessee, Knoxville
pwu11@icl.utk.edu

Stanimire Tomov
University of Tennessee, Knoxville
tomov@icl.utk.edu

Jack Dongarra
University of Tennessee, Knoxville
Oak Ridge National Laboratory, USA
University of Manchester, UK
dongarra@icl.utk.edu

ABSTRACT

The use of low-precision arithmetic in mixed-precision computing methods has been a powerful tool to accelerate numerous scientific computing applications. Artificial intelligence (AI) in particular has pushed this to current extremes, making use of half-precision floating-point arithmetic (FP16) in approaches based on neural networks. The appeal of FP16 is in the high performance that can be achieved using it on today's powerful manycore GPU accelerators, e.g., like the NVIDIA V100, that can provide 120 TeraFLOPS alone in FP16. We present an investigation showing that other HPC applications can harness this power too, and in particular, the general HPC problem of solving $Ax = b$, where A is a large dense matrix, and the solution is needed in FP32 or FP64 accuracy. Our approach is based on the mixed-precision iterative refinement technique – we generalize and extend prior advances into a framework, for which we develop architecture-specific algorithms and highly-tuned implementations that resolve the main computational challenges of efficiently parallelizing, scaling, and using FP16 arithmetic in the approach on high-end GPUs. Subsequently, we show for a first time how the use of FP16 arithmetic can significantly accelerate, as well as make more energy efficient, FP32 or FP64-precision $Ax = b$ solvers. Our results are reproducible and the developments will be made available through the MAGMA library. We quantify in practice the performance, and limitations of the approach.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Computing methodologies** → **Vector / streaming algorithms**;

KEYWORDS

Mixed-precision iterative refinement, half precision, HPC, GPGPU

ACM Reference Format:
Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. 2017. Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers. In *ScalA17: ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3148226.3148237

1 INTRODUCTION

Hardware trends in processor designs have been a driving force for the development of innovative high-performance numerical algorithms. Often, even well established algorithms must be redesigned to follow hardware trends and be efficient on new architectures. Notable examples are in the area of dense linear algebra, where many algorithms, operating with O(n^3) floating point arithmetic instructions (flops) on O(n^2) data, are expected to have performance close to the machine peak. These are the cases of redesigning LINPACK from using vector algorithms on the vector machines in the 70's to the popular LAPACK that uses blocked algorithms on cache-based processors. Other, more recent examples, include the redesign of LAPACK for multicore and heterogeneous manycore architectures, e.g., as evident in the MAGMA library [16].

The hardware trend that we target to harness in this paper, is that on modern architectures multiple floating-point arithmetic precisions are supported in hardware, and lower precisions are often much faster than the higher ones. For example, single precision 32-bit floating-point arithmetic (FP32) is usually twice as fast as double precision 64-bit floating-point arithmetic (FP64). Indeed, on most current multicore CPUs, high-end NVIDIA GPUs (e.g., P100), AMD GPUs (e.g., FirePro W9100), and Intel Xeon Phi, the single precision peak is twice the double precision peak. On some NVIDIA GPUs (e.g. the GeForce GTX Titan Black) the ratio of single vs double precision peak is $3\times$, but can go up to $32\times$ (e.g. on Titan X) depending on the ratio of 32-bit to 64-bit CUDA cores.

Recently, artificial intelligence (AI) neural network applications raised the need for FP16 arithmetic (see Figure 1), and some vendors started to accelerate it in hardware. An example is the NVIDIA P100 GPU that can reach 18.7 TeraFLOPS in FP16. Further hardware acceleration is provided in the up-coming V100 GPU that has special

Tensor Cores to achieve a peak of 120 TeraFLOPS vs 15 TeraFLOPS for FP32. Thus, getting aligned to use efficiently these hardware trends will be highly beneficial for high-performance linear algebra libraries.

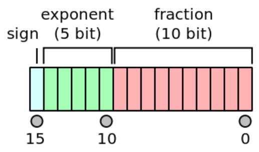


Figure 1: IEEE 754 FP16 format. This representation leads for example to 3.311 **decimal digits of accuracy and a maximum representable value of** $65,504$**.**

There are various ways to benefit from faster lower-precision arithmetic. In this paper we target the acceleration of $Ax = b$ solvers, where A is a dense matrix. Our approach is based on the mixed-precision iterative refinement technique. The main idea of this approach is to compute the LU factorization of A in low precision, and use that factorization as a preconditioner in an iterative refinement in higher-precision. The overall computation is fast, as the bulk of the flops, $O(n^3)$, are to factorize A in fast arithmetic. The few iterative steps, requiring $O(n^2)$ flops, refine the solution to high-precision accuracy. These methods have been extensively studied theoretically in the past, as discussed in Section 2. While there is certain understanding about the theory, a persistent challenge has been how to design and develop architecture-specific algorithms and highly-tuned implementations that resolve main computational issues, e.g., related to efficient parallelization, scaling, use of mixed-precision calculations, and tuning on new architectures. Indeed, much of the theoretical work has been restricted to Matlab or reference implementations, with numerical experiments on small problems, which is prone to overlook computational issues towards achieving the main goal of the technique – namely, acceleration over highly tuned working precision solvers. To address this on GPUs, we leverage building blocks from the MAGMA library that provides state-of-the-art high-performance algorithms like LU and other, including a set of highly tuned mixed-precision iterative algorithms for FP32-FP64 arithmetic [17].

2 RELATED WORK

Iterative refinement is a classic 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 one (see Wilkinson [18], Moler [10] and Stewart [15]). As Demmel points out [5], the non-linearity of the round-off error makes the iterative refinement process equivalent to the Newton's method applied to the function $f(x) = b - Ax$. The choice of the stopping criterion in the iterative refinement process is critical. Formulas for the error computed at each step of the algorithm can be obtained, e.g., in [6, 11].

Replacing the direct solve of correction equation by an iterative method leads to *nesting* of two iterative methods. Variations of this type of nesting, also known in the literature as an *inner–outer* iteration, have been studied, both theoretically and computationally [7, 12, 14], as well as used in mixed-precision computation scenarios [1]. A recent study applied a version of these combinations to FP16 and studied it theoretically [2], confirming a result analogous to mixed FP32-FP64 iterative refinement that if the condition of A is not too bad ($\kappa(A) < 10^4$) then using FP16 for the $O(n^3)$ work (LU) and FP32/FP64 for the $O(n^2)$ work (refinement) one can expect to achieve forward and backward error of order $10^{-8}/10^{-16}$. The same work also proved that when using GMRES preconditioned by the FP16 LU as the refinement procedure, the constraint on the condition of matrix A can be relaxed to be $\kappa(A) < 10^8$. The study is theoretical with Matlab implementation and test results on small matrices. For some of them, even though there is convergence, the rate is slow, which would require detection and switch to faster algorithm. This is a typical case for inner-outer iterations where the exact speedup, or if any at all, depends on the convergence rate of GMRES and the iterative refinement, which remains to be investigated and thus subject to this study.

3 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 hardware accelerated FP16. Thus, we address computational challenges of efficiently parallelizing, scaling, and using FP16 arithmetic on today's highly parallel manycore GPUs to accelerate $Ax = b$ solvers. Namely, we:

- Introduce a framework for exploiting mixed-precision FP16-FP32/FP64 iterative refinement solvers and describe the path to draw high-performance GPU implementations, and predict the possible performance that can be achieved with these algorithms;
- Present a detailed analysis and study of the correlation between *FP16-FP32/FP64* arithmetic and *performance* for six different types of iterative refinement methods that are representative for a wide range of real scientific applications. This provides a clear understanding about the factors that contribute to performance;
- Provide analysis of the *numerical behavior* on different types of matrices and present a collection of lessons that allow researchers to understand and develop their own computational kernels in a simple and efficient fashion;
- Investigate and show how the hardware accelerated FP16 can be used to accelerate general $Ax = b$ solvers;
- Describe the analysis to design a performance model that provides an insight on the performance spectrum of iterative refinement solvers. The main advantage of this model is that it helps predicting more accurately the achievable performance;
- Quantify in practice the performance, and limitations of the approach on P100 GPUs.

The algorithms are described in detail to make them independently reproducible. Moreover, the codes, and driver routines that can be used as benchmarks, will be released and made available through the open-source MAGMA library.

4 FRAMEWORK

We develop a framework that unifies Iterative Refinement (IR) and preconditioned Krylov solvers from both a theoretical and practical point of view. We begin by examining two well developed strategies in solving linear systems – classic IR and Generalized Minimal RESidual (GMRES) with preconditioning. These two methods are typically applied to different matrices, the former to dense and the latter to sparse. Dense system is usually solved with a direct solver while iterative solvers are usually used for sparse systems. We will see that IR gives dense linear solver some iterative flavor as suggested in the name, and preconditioning gives sparse iterative solver some direct flavor. We will also see how these methods fit in our framework that has an even broader scope as it is rich in capabilities to generate more innovative, fast, and accurate solvers that adapt to different problem domains and the rapidly changing hardware landscapes.

4.1 Background

We begin by introducing two well known methods: Classic iterative refinement and preconditioned GMRES.

4.1.1 Classic Iterative Refinement. IR is a technique for improving a computed solution $\hat{x}$ to a linear system of equations $Ax = b$. One iteration of the refinement consists of three steps:

(1) Compute the residual $r = b - A\hat{x}$
(2) Solve the correction equation $Ad = r$ (typically using the LU factors produced by the initial solve)
(3) Correct the current solution $\hat{x} = \hat{x} + d$.

It can be seen that if the correction equation is solved exactly, one iteration of the refinement will produce the exact solution. In reality, due to various factors (round-off errors, inexact solvers) the correction equation can only be solved approximately, and the above iteration must be repeated until a satisfactory solution is obtained.

Traditionally, IR was used with Gaussian elimination with the residual computation (step 1) using higher precision than the working precision. This is called mixed-precision IR [4], in contrast to fixed-precision IR where all arithmetic is in the same precision. IR has been working well in this way: more precisely, if double precision is used in computing the residual (step 1) and all other arithmetic is in single precision, and the condition number of A is not too large, then IR converges to a solution correct to single precision (see Chapter 12 in [8]). IR can be used to improve the accuracy of an already stable algorithm, or it can be used to stabilize a less stable solver. In this paper, we are primarily interested in using IR to improve the solution of a low-precision (such as FP16 LU) solver.

The economics of IR using low-precision LU depends on how fast the low-precision initial solver can be and the relative cost of the refinement process and the number of iterations of refinement required. It is therefore a function of the executing hardware, the exact configuration of the process (different precisions and solver types), and also of the property of the matrix A (condition number). Note that in contrast, for a direct solver such as LU the speed does not depend on the spectral property of the matrix.

4.1.2 Preconditioned GMRES. Generalized Minimal RESidual (GMRES) is a popular Krylov space iterative solver for a general matrix, although it is most often used for large and sparse matrices. It was proposed by Y. Saad and M. Schultz [13] in 1986. The main idea is to find an approximate solution in the Krylov space at each iteration that has the minimal residual. The GMRES method works as a series of iterations where each iteration consists of the three steps (at iteration n):

(1) Compute q_n ($\perp$ to $q_1 \dots q_{n-1}$) using Arnoldi Iteration
(2) Solve y_n which minimizes $||r_n||$ using least square solver
(3) Compute $x_n = Q_n y_n$, where $Q_n = (q_1|\dots|q_n)$
(4) (Repeat if the residual is not small enough)

The Arnoldi Iteration (step 1) finds successively an orthogonal matrix Q_n and a Hessenberg matrix H_n such that $AQ_n = Q_{n+1}H_n$. It follows that

$$||Ax_n - b|| = ||H_n y_n - Q_{n+1}^T b|| \tag{1}$$

which can be solved rather cheaply exploiting the Hessenberg structure of H_n and also updating the QR factorization of H_{n+1} based on the QR factorization of H_n.

GMRES by itself might converge slowly on certain matrices with unfavorable spectral distribution or non-normal matrices. Preconditioning is thus an important technique that improves the property of the matrix A such that the GMRES can converge fast. Conceptually, the preconditioner is a matrix M^{-1} such that $M^{-1}A$ is close to identity. A crucial tradeoff in choosing a preconditioner is between the computation cost of calculating and applying M^{-1} and the closeness to identity. In practice, M^{-1} is never formed but whenever A is multiplied by a vector v (step 1,4) the preconditioner is applied as $M^{-1}Ax$. Note that *the application of the preconditioner is a linear solve for matrix M and right hand side vector Ax.*

4.2 The conceptual framework

By looking at the classic IR and preconditioned GMRES we can see the common structure: a linear solver inside a linear solver. In the classic IR, it is a correction equation inside, usually solved by LU triangular solve; in the preconditioned GMRES case, it is the application of preconditioner in every iteration. Our most interesting assumption is that there is a low precision (FP16) factorized $A \approx LU$ available and we would like to devise a linear solver that achieves at least single precision accuracy. If we follow the IR, we would use the low precision LU to get an initial solution, and use LU triangular solves to refine the solution iteratively. We categorize this method as IR (Iterative Refinement using triangular solve). On the other hand, we can use GMRES with low precision $M = LU \approx A$ as the preconditioner. We categorize this method as GM (GMRES using triangular solve preconditioner application). Furthermore we could see more innovative combinations of solver types and precisions within the “solver inside solver” framework. For example, in iterative refinement we could use GMRES to solve the correction equation instead of the triangular solve, resulting in a new type of solver IR-GM (Iterative Refinement using GMRES preconditioned by $M = LU$ to solve correction equation). This idea has been proposed and analyzed by Erin Carson and Nick Higham in [2, 3]. On the other hand, we can also use GMRES as a preconditioner inside an “outer” GMRES iteration, which would be categorized as GM-GM. This formulation can be further instantiated by using a different “outer” solver type, such as CG, and also for the “inner” solver type. We will however restrict our focus in the four general configurations: 1) IR; 2) IR-GM; 3) GM; 4) GM-GM. To make this concrete, we illustrate the four algorithms separately, as shown in algorithms 1-2.

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

Algorithm 1: IR: classic (mixed precision) Iterative Refinement using triangular solve. IR-GM: iterative refinement with GMRES to solve correction equation.

```
Data: An $n \times n$ matrix $A$, size $n$ vector $b$
Result: A solution vector $x_k$
(FP16) Solve $Ax_0 = b$ using $A \approx LU \rightarrow M$ factorization and
  triangular solve ;
Compute $r \leftarrow M^{-1}(Ax_0 - b), \beta = ||r_0||, q_1 \leftarrow r_0/\beta, b \leftarrow M^{-1}b$;
for $j = 1, \ldots, k$ do
    (FP64) Compute $h_{i,j} \leftarrow (M^{-1}Aq_j, q_i), i = 1, 2, \ldots, j$;
    (FP64) Compute $q_{j+1} \leftarrow M^{-1}Aq_j - \sum_{i=1}^{j} h_{i,j}q_i$,
      $h_{j+1,j} \leftarrow ||q_{j+1}||$, $q_{j+1} \leftarrow q_{j+1}/h_{j+1,j}$;
$x_k = x_0 + Q_k y_k$, where $y_k$ solves $\min ||\beta e_1 - H_k y_k||$;
```

Algorithm 2: GM: GMres iterations using triangular solve to apply FP16 LU preconditioner. M^{-1} means triangular solve. GM-GM: GMres iterations using GMres to solve with LU preconditioner. M^{-1} means GMRES solve with $M = LU$ as preconditioner.

4.3 Convergence and performance consideration

This subsection discusses the convergence condition, rate, and economics in performance of the four solvers for a variety of matrix types and sizes.

4.3.1 Convergence. The robustness of the solvers depends on the convergence condition, and the performance of the solvers on the convergence rate. Unfortunately while some convergence conditions are known for iterative refinement and preconditioned results, the convergence rate is hard to predict.

For the mixed precision iterative refinement the latest analysis can be found in [2, 3]. Assuming that LU factorization is done in FP16, the working precision is FP32, and the residual is computed in FP64, the IR is going to converge if the condition number of A is relatively small: $\kappa_2(A) < 10^4$. Using GMRES to solve the correction equation (IR-GM), the condition on A can be relaxed: $\kappa_2(A) < 10^8$. For GMRES-based solvers, in theory GMRES will converge in n steps, preferably much less than n, if no rounding errors are present. However it is very hard to predict how fast the convergence will be now that we have a preconditioner that is FP16 accurate $M = LU \approx A$. For normal matrix A the iterations needed to converge in GMRES increase as the condition number of A increases; for non-normal matrix the convergence rate cannot be entirely predicted by condition number only. In practice, the convergence and convergence rate depend on the matrix type, condition number, and matrix size. We will present empirical investigations in the next section covering the four algorithms on matrices with different spectrum, sizes, and types.

4.3.2 Performance. Our main motivation in utilizing FP16 is its high performance vs. performance in higher precisions, e.g., illustrated for the NVIDIA P100 GPU in figure 2. The P100 has peak FP16 performance of 18.7 TFLOPS and indeed figure 2a shows the FP16 matrix-matrix multiplication routine (`hgemm`) achieves close to 15 TFLOPS on a tall-skinny matrix multiplication which constitutes the main computation load in LU factorization (`Xgetrf`). As expected, the achieved performance of FP16 `hgemm` is about 2× that of FP32 `sgemm` and 4× of FP64 `dgemm` on P100. Further, from figure 2b we see that FP16 LU factorization `Xgetrf` follows roughly the same trend as `Xgemm` when the matrix is large; when the matrix size is small the performance advantage of FP16 `hgetrf` vs. FP32 `sgetrf` or FP64 `dgetrf` is not so pronounced. This is partly due to an implementation choice we made in `hgetrf` where the matrix multiplication is done in FP16 but the panel factorization is done in FP32 as optimized FP16 panel factorization is not readily available. All the precision conversions are accounted for in the figure. The situation for smaller matrix can be improved by implementing a high performance panel factorization in FP16 which will be left as future work.

4.4 Performance model

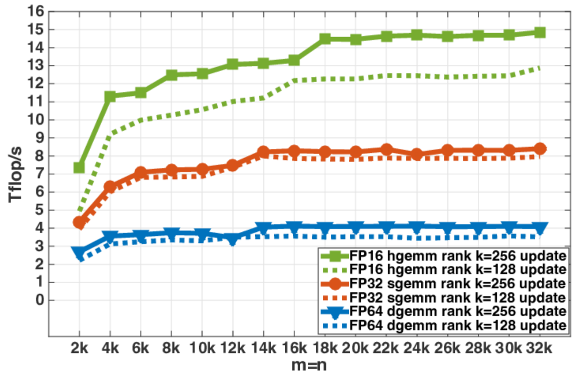
This section is dedicated to the theoretical performance analysis of the mixed precision (MP) algorithms for linear solvers. A detailed study of linear solver based on LU factorization algorithms is used for the sake of illustration. The idea is to provide a model that allows us to understand and define when iterative refinement techniques can be used in a beneficial fashion. 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 LU factorization requires $\frac{2n^3}{3}$ operations and $O(n^3)$ of them are compute intensive. Thus, it should behave like a compute intensive routine. We mention that the LU factorization can reach about 85% of the `Xgemm` kernel which is the most compute intensive kernel. The iterative refinement solvers consist of an LU factorization in particular precision $\varepsilon_{FPXX} < \varepsilon_{FP64}$, and then, iterative refinement based on 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:

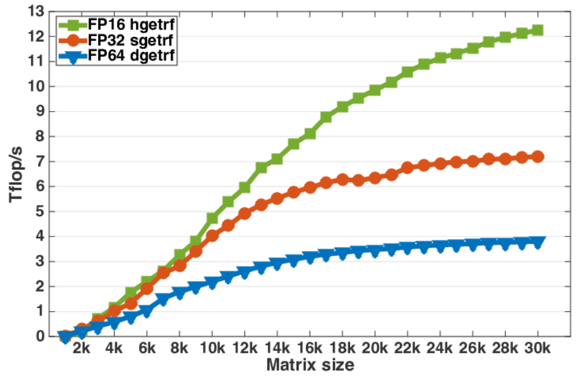
$$\text{time for MP } = \frac{2n^3}{3P_{Xgetrf}} + k\left(\frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{Xtrsv}}\right) \quad (2)$$

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

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.



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



(b) Performance of the Xgetrf routine.

Figure 2: Performance of the three arithmetic precisions obtained on a Nvidia P100 GPU.

From theoretical and practical point of view, the performance of the half and the single precision is about 4× and 2× the one of double precision, respectively. This means that $P_{hgetrf} = 2P_{sgetrf} = 4P_{dgetrf}$ and that $P_{sgetrf} = 2P_{dgetrf}$. The dgemv performance on a P100 GPU is about 130 Gflop/s, while both dtrsv and strsv, which are also memory bound kernels, achieve around 120 and 140 Gflop/s, respectively. Thus, for simplicity, we can consider that $P_{dgemv} = P_{dtrsv} = P_{strsv}$. Also, when the LU factorization routine (e.g., Xgetrf) is well implemented and optimized, its performance is considered to be about 85% of the Xgemm routine. On most of the new architectures (multicore, GPUs, KNL) the performance of the Xgemm routine is close to the peak of the machine and is about 30× higher than both the Xgemv and the Xtrsv routines that are both memory bound kernels, meaning that $P_{Xgetrf} \approx 30P_{Xgemv}$ (see figure 2a for the Xgemm performance). Thus, from a theoretical point of view we can compute the maximum speedup that a method will bring by:

for the FP32→FP64:

$$S_{\text{FP32}\rightarrow\text{FP64}} = \frac{\text{time FP64}}{\text{time FP32}\rightarrow\text{ FP64}} = \frac{\frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dgemv}}}{\frac{n^3}{3P_{dgetrf}} + k(\frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{dgemv}})}$$

$$= \frac{\frac{2n^3}{3*30P_{dgemv}} + \frac{2n^2}{P_{dgemv}}}{\frac{n^3}{3*30P_{dgemv}} + \frac{4kn^2}{P_{dgemv}}} = \frac{2n+180}{n+360k} \tag{4}$$

$$S_{\text{FP16}\rightarrow\text{FP64}} = \frac{\frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dtrsv}}}{\frac{2n^3}{3P_{hgetrf}} + k(\frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{strsv}})} = \frac{2n+360}{n+720k} \tag{5}$$

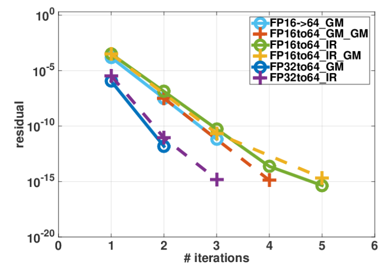
From a practical point of view, the cost of the refinement iteration is slightly larger than one Xgemv and one Xtrsv. It involves pivoting, residual check, synchronizations and orthogonalization (for GMRES). Thus, to have a more realistic upper bound speedup, the factor of the number of iterations should be multiplied by 1.4 (value obtained from experiments).

$$S_{\text{FP32}\rightarrow\text{FP64}} = \frac{2n+180}{n+500k} \qquad S_{\text{FP16}\rightarrow\text{FP64}} = \frac{2n+360}{n+1080k} \tag{6}$$

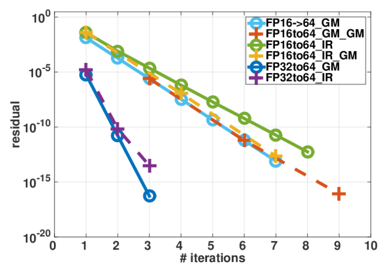
5 NUMERICAL BEHAVIOR DISCUSSION

All our experiments are performed on a computer equipped with two Intel(R) Xeon(R) E5-2650 v3 @ 2.30GHz CPUs and one NVIDIA P100 PCIe GPU accelerator. We developed LU factorization, solve, and iterative refinement routines to support FP16 and FP32. The GMRES routines were adapted from R. Li and Y. Saad [9] to work with MAGMA [16].

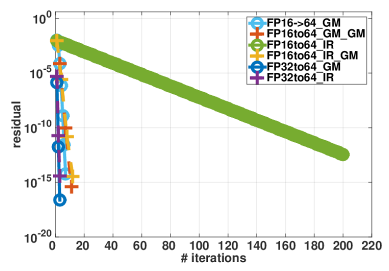
Figure 3 illustrates the convergence history of our four proposed algorithms using the FP16 LU factorization in addition to two similar variants with FP32 LU instead of FP16 LU. They are labeled as FP32→64 IR and FP32→64 GM. Figure 3a represents the most numerically favorable type of matrix to solve—the diagonal dominant matrix. We can see that every method converges within 5 iterations, and all the FP16 based methods perform similarly. Not surprisingly, FP32 based method converges in fewer iterations because of its higher precision. Figure 3b represents a less favorable matrix type that has positive eigenvalues and singular values whose logarithms are uniformly distributed. Among FP16 methods, GM and IR_GM converge fastest. Figure 3c shows a more difficult matrix for FP16 methods. This type of matrix has positive eigenvalues and all singular values being 1, except one, being 0.01. Among FP16 methods the classic IR converges very slowly, while GM converges fastest in 7 iterations. Again, FP32 based methods have no problem converging. Figure 3d shows the convergence for a matrix that has the same singular value distribution but not necessarily positive eigenvalues. Now all FP16 methods converge in around 30 iterations, and still GM converges fastest—in 24 iterations. Figure 3e is another easy type of matrix to solve for FP16 methods, and again GM converges fastest (in 6 iterations). Figure 3f is the most difficult matrix where the fastest FP16 method – the GM – converges in around 200 iterations while the worst IR_GM converges in 900 iterations. For all the matrices we considered here, FP16→64 GM is the most robust one and fastest in convergence among the FP16 based methods. *This observation suggests the surprising effectiveness of a traditionally sparse iterative solver GM, which might be robust enough for dense linear systems and also economical coupled with proportionally fast FP16 arithmetic.*



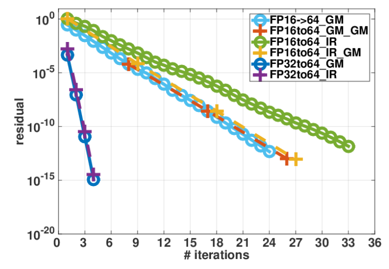
(a) matrix with diagonal dominant.



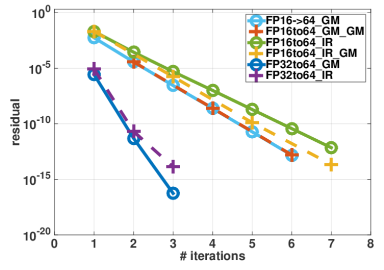
(b) matrix with positive λ and where σ_i is random number between $\frac{1}{cond}$, and 1 such that their logarithms are uniformly distributed.



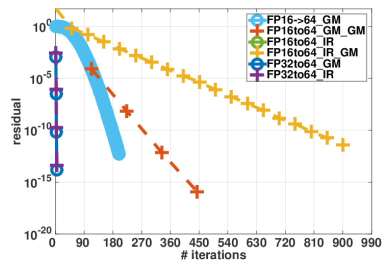
(c) matrix with positive λ and with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$)



(d) matrix with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$)



(e) matrix with positive eigenvalues and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.



(f) matrix with arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

Figure 3: Convergence history of the 6 proposed refinement algorithms for different type of matrices all of size 10240 and generated with condition number equal to 10^2.

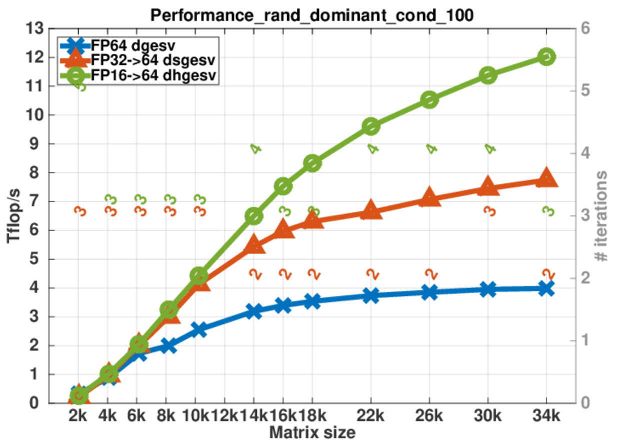
6 EXPERIMENTAL RESULTS DISCUSSION

This section presents the performance results obtained by 1) our two proposed iterative refinement methods, dhgesv (FP16) and dsgesv (FP32), using the GMRES method, and 2) the reference FP64 dgesv solver. We note that when the IR method converges, its behavior is similar to GM within 2-3% +/- difference in performance. The IR-GM method was slightly 5-10% slower. Figure 4 illustrates the performance results for the three routines for different types of matrices, as well as, we depict the number of iterations required by each of the two refinement methods (dsgesv and dhgesv) in order to achieve the FP64 solution. In each figure, there are three performance curves that refer to: 1) the reference dgesv routine that uses the double precision (FP64) arithmetic in all its computational steps; 2) the dsgesv routine which performs the LU factorization in single precision (FP32) and uses the preconditioned FGMRES (GM method) to achieve the FP64 solution; and 3) the dhgesv routine that performs the LU factorization in half precision (FP32) and uses the preconditioned FGMRES (GM method) to achieve the FP64 solution.

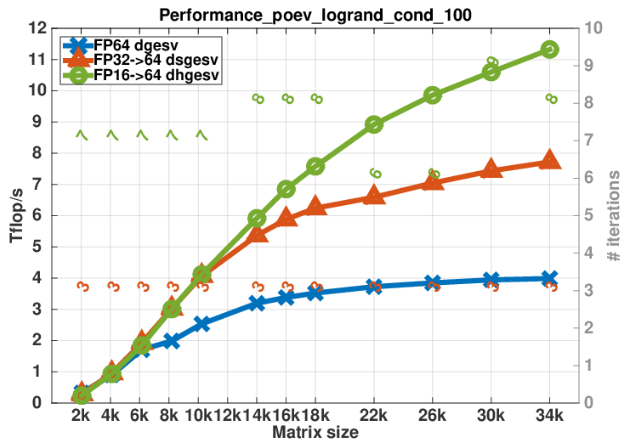
In Figure 4a, the matrix is diagonally dominant and thus both dsgesv and dhgesv converge very fast within small number of iterations. Thus, based on Equation (6) from Section 4.4, one can expect very good speedup over the reference dgesv routine. The performance illustrated in Figure 4a runs alongside with the expectation model – the dhgesv (FP16→FP64) routine depicted in green color is about 3× faster than the reference dgesv depicted in blue. Also by looking at the dsgesv (FP32→FP64) depicted in red color, we can see that it is about 1.8× faster than dgesv. In other term, this means that for diagonal dominant matrices, using the refinement techniques provide a large gain, 1.8× to 3× speedup.

Similarly to Figure 4a, Figure 4b shows the performance and the number of iterations of our methods for matrices with positive eigenvalue and logarithmic uniform distribution of their singular values. As shown in the figure, the number of iterations does not increase with the matrix size for both methods and is slightly higher than the one with diagonal dominant matrices for the dhgesv routine, while it remain around 3 iterations for the dsgesv routine. Thus one can expect that the performance gain will be roughly similar to diagonal dominant. The dhgesv (FP16→FP64) routine achieves the same solution as the dgesv FP64 but is about 2.6× faster, and the dsgesv (FP32→FP64) keeps the same speedup ratio of being 1.7× faster.

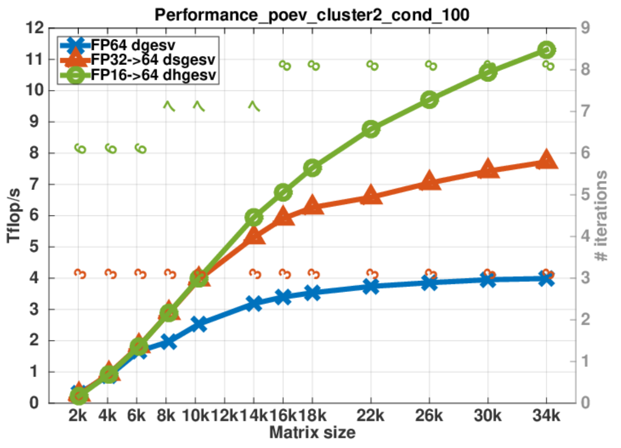
Figure 4c presents our study for matrices with clustered singular values and positive eigenvalues. The behavior observed here is similar to the previous one where both dhgesv and dsgesv are



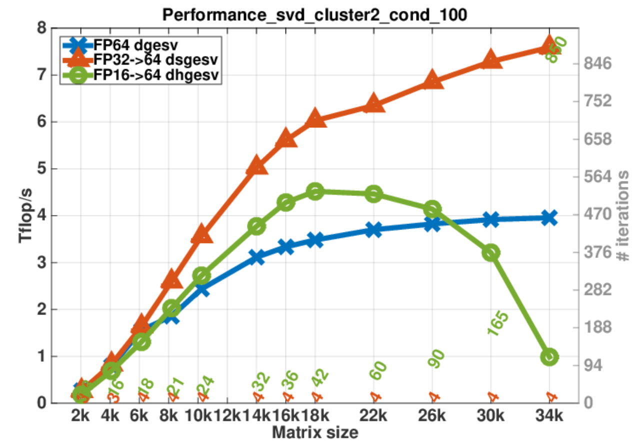
(a) matrix with diagonal dominant.



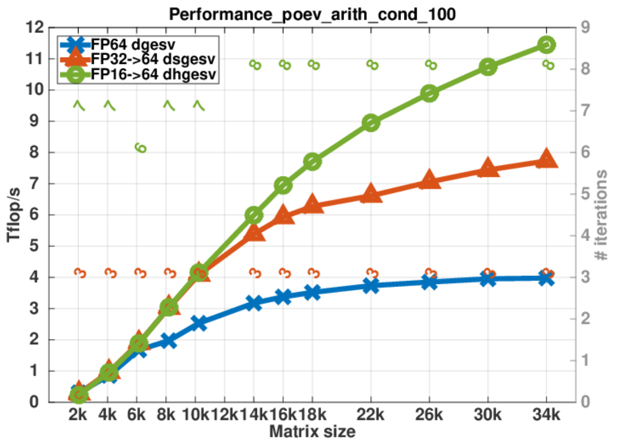
(b) matrix with positive λ and where σ_i is random number between $\frac{1}{cond}$, 1 such that their logarithms are uniformly distributed.

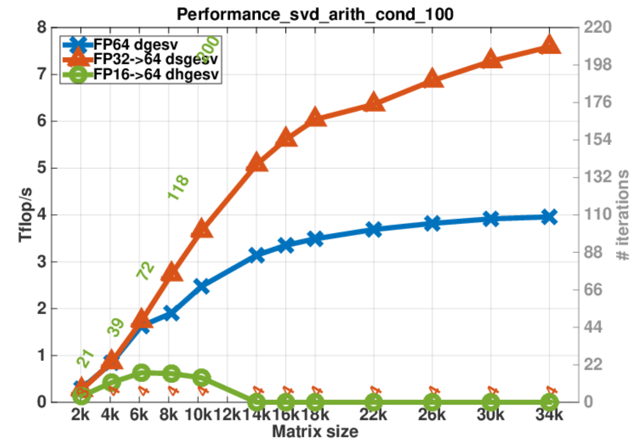


(c) matrix with positive λ and with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$)



(d) matrix with clustered singular values, σ_i=(1, $\cdots$, 1, $\frac{1}{cond}$)





(e) matrix with positive eigenvalues and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

(f) matrix with arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$.

Figure 4: Performance in Tflop/s for the three linear solver (dgesv,dsgesv,dhgesv) and the required number of iterations to achieve FP64 arithmetic solution using either the dsgesv (orange numbers) or the dhgesv (green numbers) solver, for different matrices size and different type of matrices. Note that $cond = 10^2$ for these experiments.

advantageous and can provide 1.7× to 2.6× speedup. In contrast to Figure 4c, Figure 4d shows the performance and the number of iterations for matrices that have similar clustered singular value distribution but when the eigenvalues are with imaginary part. The observation that we made here is interesting. The behavior of the dsgesv (FP32→FP64) remains the same as in the previous experiments, meaning that the number of iterations is not strongly dependent on the matrix size or on the matrix type. The dsgesv requires about 3 iterations and thus we can always see a factor of 1.7× speedup. For the dhgesv, the convergence is matrix type and also matrix size dependent. The number of iterations increases dramatically with the matrix size and is larger than what is observed in Figure 4c. In this case, the rounding error of the FP16 computation for the LU factorization, and possibly the range of the representative numbers for FP16 arithmetic, disturb the system and the convergence rate decreases dramatically, and thus affecting the performance. In this case we can see that the half precision dhgesv routine is not beneficial at all and can be even slower than the FP64 one. For such type of matrices, the best is to use the dsgesv (FP32→FP64) routine.

Figure 4e corresponds to the results obtained for matrices with positive eigenvalues and arithmetic distribution of its singular values. The dsgesv behavior stays the same as the one shown in the previous graph. It requires about 3 iterations and thus brings a factor of 1.7× speedup over the dgesv. Also, we can notice that the dsgesv routine acts similarly to all matrices with positive eigenvalues and it converges in about 7-8 iterations, making it an attractive routine to use in such cases, offering a factor of 2.7× speedup.

Figure 4f represents the arithmetic singular values distribution without the constraint that the eigenvalues are positive, thus, the eigenvalues are in the complex plane. Here we can notice that the dsgesv still converges with about 3-4 iterations for any matrix size, leading to the observed 1.7× speedup while the dhgesv fails to converge within 300 iterations for most of the large matrices, making it useless.

7 CONCLUSIONS AND FUTURE DIRECTIONS

We developed a framework for accelerating the general $Ax = b$ solver using hardware-accelerated FP16 arithmetic on GPUs. The approach, based on the mixed-precision iterative refinement technique, was extended from prior developments for FP16 arithmetic and we developed architecture-specific algorithms and highly-tuned implementations that resolve the main computational challenges of efficiently parallelizing, scaling, and using FP16 arithmetic in the approach on high-end GPUs. We quantified in practice the performance and the limitations of the approach. In particular, we can conclude that the dsgesv (FP32→FP64) routine can be used for all matrix types and for any sizes, and it provides a speedup of about 1.7×, which also means about the same factor gain in energy. The dhgesv (FP16→FP64) is advantageous when the matrix has nice properties, such as diagonal dominance, or when eigenvalues are always positive. In this later case we can expect a factor of 2.5 to 2.7× speedup, and thus also energy saving. Based on our analysis, we think that the (FP16→FP64) will be always beneficial in the case of the Cholesky factorization which, will be considered in future work.

Other future work includes releasing the software through MAGMA and tuning it for the up-coming V100 GPU to use the Tensor Cores acceleration. Of interest is also to investigate the energy-efficiency of the approach compared to working precision implementations and other architectures.

ACKNOWLEDGMENTS

We gratefully acknowledge the advice and discussions with Prof. Nick Higham that improved the manuscript. 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. The work was also partially supported by Nvidia and NSF grant No. 1514406.

REFERENCES

[1] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. 2009. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications* 180, 12 (2009), 2526–2533.
[2] Erin Carson and Nicholas J Higham. 2017. Accelerating the solution of linear systems by iterative refinement in three precisions. *MIMS EPrint 2017.24, University of Manchester* (2017).
[3] Erin Carson and Nicholas J Higham. 2017. A New Analysis of Iterative Refinement and its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *MIMS EPrint 2017.12, University of Manchester* (2017).
[4] James Demmel, Yozo Hida, William Kahan, Xiaoye S Li, Sonil Mukherjee, and E Jason Riedy. 2006. Error bounds from extra-precise iterative refinement. *ACM Transactions on Mathematical Software (TOMS)* 32, 2 (2006), 325–351.
[5] J. W. Demmel. 1997. *Applied Numerical Linear Algebra*. SIAM.
[6] J. W. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. 2006. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.* 32, 2 (2006), 325–351.
[7] Gene H. Golub and Qiang Ye. 2000. Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iteration. *SIAM Journal on Scientific Computing* 21, 4 (2000), 1305–1320.
[8] Nicholas J Higham. 2002. *Accuracy and stability of numerical algorithms* (second ed.). SIAM.
[9] Ruipeng Li and Yousef Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63, 2 (2013), 443–466.
[10] C. B. Moler. 1967. Iterative Refinement in Floating Point. *J. ACM* 14, 2 (1967), 316–321.
[11] W. Oettli and W. Prager. 1964. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numer. Math.* 6 (1964), 405–409.
[12] Y. Saad. 1991. *A flexible inner-outer preconditioned GMRES algorithm*. Technical Report 91-279. Department of Computer Science and Egineering, University of Minnesota, Minneapolis, Minnesota.
[13] Youcef Saad and Martin H Schultz. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing* 7, 3 (1986), 856–869.
[14] Valeria Simoncini and Daniel B. Szyld. 2002. Flexible Inner-Outer Krylov Subspace Methods. *SIAM J. Numer. Anal.* 40, 6 (2002), 2219–2239. https://doi.org/10.1137/S0036142902401074
[15] G. W. Stewart. 1973. *Introduction to Matrix Computations*. Academic Press.
[16] S. Tomov, J. Dongarra, and M. Baboulin. 2010. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parellel Comput. Syst. Appl.* 36, 5-6 (2010), 232–240. DOI: 10.1016/j.parco.2009.12.005.
[17] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. 2010. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proc. of the IEEE IPDPS'10*. Atlanta, GA, 1–8.
[18] J. H. Wilkinson. 1963. *Rounding Errors in Algebraic Processes*. Prentice-Hall.