Investigating the Benefit of FP16-Enabled Mixed-Precision Solvers for Symmetric Positive Definite Matrices Using GPUs

Ahmad Abdelfattah[1(✉)], Stan Tomov[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, USA
{ahmad,tomov,dongarra}@icl.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
[3] University of Manchester, Manchester, UK

Abstract. Half-precision computation refers to performing floating-point operations in a 16-bit format. While half-precision has been driven largely by machine learning applications, recent algorithmic advances in numerical linear algebra have discovered beneficial use cases for half precision in accelerating the solution of linear systems of equations at higher precisions. In this paper, we present a high-performance, mixed-precision linear solver ($Ax = b$) for symmetric positive definite systems in double-precision using graphics processing units (GPUs). The solver is based on a mixed-precision Cholesky factorization that utilizes the high-performance tensor core units in CUDA-enabled GPUs. Since the Cholesky factors are affected by the low precision, an iterative refinement (IR) solver is required to recover the solution back to double-precision accuracy. Two different types of IR solvers are discussed on a wide range of test matrices. A preprocessing step is also developed, which scales and shifts the matrix, if necessary, in order to preserve its positive-definiteness in lower precisions. Our experiments on the V100 GPU show that performance speedups are up to 4.7× against a direct double-precision solver. However, matrix properties such as the condition number and the eigenvalue distribution can affect the convergence rate, which would consequently affect the overall performance.

Keywords: Mixed-precision solvers · Half-precision · GPU computing

1 Introduction

The solution of a dense linear system of equations ($Ax = b$) is a critical component in many scientific applications. The standard way of solving such systems includes two steps: a matrix factorization step and a triangular solve step. In this paper, we discuss the specific case where the matrix $A_{N \times N}$ is dense and symmetric positive definite (SPD). It is also assumed that A, b, and x are stored in 64-bit double precision format (FP64).

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12138, pp. 237–250, 2020.
https://doi.org/10.1007/978-3-030-50417-5_18

The standard LAPACK software [1] provides the `dposv` routine for solving $Ax = b$ for SPD systems in FP64. The routine starts with a *Cholesky factorization* (`dpotrf`) of A, such that $A = LL^T$, where L is a lower triangular matrix. The factors are used to find the solution x using two triangular solves with respect to b (`dpotrs`). Throughout the paper, we assume that b is an $N \times 1$ vector, and so the triangular solve step requires $\mathcal{O}(N^2)$ floating-point operations (FLOPs). In such a case, the Cholesky factorization dominates the execution time, since it requires $\mathcal{O}(N^3)$ FLOPs. Therefore, any performance improvements for solving $Ax = b$ usually focus on improving the factorization performance.

A full FP64 factorization extracts its high performance from a blocked implementation that traverses the matrix in panels of width nb (which is often called the blocking size). A blocked design enables high performance through the compute-bound Level 3 BLAS[1] routines. Sufficiently optimized routines such as matrix multiplication (`dgemm`) and symmetric rank-k updates (`dsyrk`) would guarantee a high performance Cholesky factorization that is close to the hardware peak performance. As an example, both cuSOLVER [14] (the vendor library) and the MAGMA library [4,11] reach an asymptotic performance of ≈6.3 teraFLOP/s on the V100 GPU for `dpotrf`. This is about 90% of the `dgemm` peak performance, meaning that there is little room for improving the performance of the factorization. Another direction to achieve more performance is to change the algorithmic steps for solving $Ax = b$. This is where *mixed-precision iterative refinement (MP-IR) algorithms* come into play. The basic idea of MP-IR solvers is to perform the Cholesky factorization using a "reduced precision." If FP32 is used for the factorization instead of FP64, a natural 2× improvement is expected. However, we cannot use the traditional triangular solves with the low-precision factors of A. In order to recover the solution back to FP64 accuracy, an extra algorithmic component is required: *iterative refinement (IR)*. It applies iterative corrections to an initial solution vector until it converges to FP64 accuracy. Early efforts to implement such algorithms in LAPACK were introduced by Langou et al. [12], and Baboulin et al. [5]. GPU-accelerated versions of the MP-IR solver also exist in the MAGMA library [4,11].

The algorithmic structure of MP-IR solvers did not change for almost a decade. This was true until half precision (16-bit floating-point format) was introduced into commercial HPC hardware (e.g., NVIDIA GPUs). The original motivation for FP16 computation was to accelerate machine learning applications rather than scientific HPC workloads. NVIDIA GPUs support the "binary16" format which is defined by the IEEE-754 standard [2]. Intel and Google support a different format called "bfloat16". Since our study targets GPUs, we focus on the binary16 format, which we also call half precision or simply FP16. NVIDIA's Volta and Turing architectures provide hardware accelerators, called Tensor Cores (TCs), for `gemm` in FP16. TCs can also perform a mixed-precision `gemm`, by accepting operands in FP16 while accumulating the result in FP32. TCs are theoretically 4× faster than using the regular FP16 peak performance on the Volta GPU. Applications that take advantage of TCs have access to up to 125

[1] Basic Linear Algebra Subroutines.

teraFLOP/s of performance. The vendor library cuBLAS [13] provides a number of matrix multiplication routines that can take advantage of TCs. Some other efforts introduced open-source routines that are competitive with cuBLAS [3].

Such a high performance of half-precision has drawn the attention of the HPC community to assess its benefit for scientific HPC workloads. Originally motivated by the analysis of Carson and Higham [6,7], the work done by Haidar et al. [9] introduced a mixed-precision solver that is different in several ways from the ones introduced in [12] and [5]. **First**, the new method uses three precisions (double, single, and half) to solve $Ax = b$ up to double-precision accuracy. **Second**, the new solver uses a *mixed-precision LU factorization*, where the dominant trailing matrix updates are performed using a mixed-precision `gemm`. **Third**, the new solver uses a new IR algorithm based on the GMRES method, instead of the classic IR solver that is based on triangular solves. The GMRES-based IR uses the original matrix A preconditioned by its low-precision factors, which yields a faster convergence and thus a higher performance.

In this paper, we design a similar mixed-precision solver for SPD matrices. Technically, the LU factorization supports such matrices, but (1) its operation count is much higher than a Cholesky factorization, and (2) SPD matrices don't need pivoting, which is a plus for performance. We show that the developed solver works well with problems whose condition number $\kappa_\infty(A)$ is up to $\mathcal{O}(10^9)$. We also implement an optional preprocessing step that includes scaling and diagonal shifts. The preprocessing step, which is based on [10], protects the matrix from losing its definiteness when FP16 is used in the factorization. Therefore, it helps solve a wider range of problems. Our experiments are conducted on a Tesla V100 GPU and span a wide range of dense SPD matrices with different condition numbers and eigenvalue distributions. We show how these two properties affect the convergence rate of GMRES-based IR, which in turn affects the performance. Our results show that the developed solution can be up to 4.7× faster than a direct full FP64 solver. This work is lined up for integration into the MAGMA library [4,11].

2 Background and Related Work

Classic MP-IR solvers for SPD systems used to perform the Cholesky factorization in single precision. The refinement phase iteratively updates the solution vector $\hat{x}$ until it is accurate enough. At each refinement iteration, three main steps are performed. **First**, the residual $r = b - Ax$ is computed in FP64. **Second**, we solve for the correction vector c, such that $Ac = r$. This step uses the low precision factors of A. **Finally**, the solution vector is updated $\hat{x}_{i+1} = \hat{x}_i + c$. Convergence is achieved when the residual is small enough.

A key factor for the high performance of MP-IR solvers is the number of iterations in the refinement stage. As mentioned before, a maximum of 2× speedup is expected from the factorization stage in FP32. This performance advantage can be completely gone if too many iterations are required for convergence. Typically, an MP-IR solver (FP32→FP64) requires 2–3 iterations for a well-conditioned

problem. This is considered a *best case scenario*, since the asymptotic speedup approaches 2×, meaning a minimal overhead by the IR stage. In most cases, an MP-IR solver is asymptotically 1.8× faster than a full FP64 solver.

Using half precision in legacy MP-IR algorithms was mostly unsuccessful. Performing the factorization in FP16 further worsens the quality of the factors of A, which leads to a longer convergence or even a divergence. For SPD matrices, an FP16 factorization can fail due to the loss of definiteness during the conversion to FP16. While countermeasures have been proposed by Higham et al. [10], a more practical approach for high performance is possible. Similar to [9], we adopt a mixed-precision Cholesky factorization, in which the rank-k updates are performed using a mixed-precision `gemm` (FP16→FP32), while all other steps are performed in FP32. The quality of the Cholesky factors would be better than a full FP16 factorization. We also apply a slightly modified version of the preprocessing proposed by Higham et al. [10] in order to support matrices with higher condition numbers and avoid the loss of definiteness, overflow, and possibly underflow.

Now, considering the IR step, the low quality of the produced factors leads to the likely failure of the classic IR algorithm (e.g., following the classic mixed-precision solvers' convergence theory [12]). In fact, classic IR would only work for matrices with relatively small condition numbers, as we show later in Sect. 7. An alternative approach, which further improves the numerical stability and convergence of the overall solver, is to solve the correction equation ($Ac = r$) using an iterative method, such as GMRES [16]. The solver thus uses two nested refinement loops, which are also often referred to as "inner-outer" iterative solvers [15,17]. We call the new IR algorithm IRGMRES. The recent work by Carson and Higham [6,7] analyzes this type of solvers when three precisions are used (e.g., {FP16, FP32, FP64} or {FP16, FP64, FP128} for {factorization, working precision, residual precision}, respectively). They prove that, if a preconditioned GMRES is used to solve the correction equation, then forward and backward errors in the order of $10^{-8}/10^{-16}$ are achievable if the condition number of A satisfies $\kappa_\infty(A) < 10^8/10^{12}$, respectively. The work in [9] implements a simplified version of GMRES with just two precisions, typically using the working precision as the residual precision. By preconditioning GMRES using the low-precision factors of A, FP64 accuracy can be achieved for matrices with condition numbers up to 10^5. Our study expands upon this work for SPD matrices using a mixed-precision Cholesky factorization. Successful convergence is achieved for condition numbers up to 10^9. In addition, we study the behavior of both IR and IRGMRES for a wide range of SPD matrices, and show how the condition number and the eigenvalue distribution affect the convergence of the IRGMRES solver. Finally, we show that the modified version of the preprocessing steps proposed in [10] enable our solver to support harder problems that were not solvable otherwise (i.e., without preprocessing).

3 System Setup

All the experiments reported in this paper are conducted on a system with two Intel Broadwell CPUs (Intel Xeon CPU E5-2698 v4 @ 2.20 GHz), with 20 cores per CPU. The system has 512 GB of memory. The GPU is a Tesla V100-SXM2, with 80 multiprocessors clocked at 1.53 GHz. Our solver is developed as part of the MAGMA library, which is compiled using CUDA-10.1 and MKL-2018.0.1 for the CPU workloads. The number of MKL threads is set to 40 throughout all the experiments.

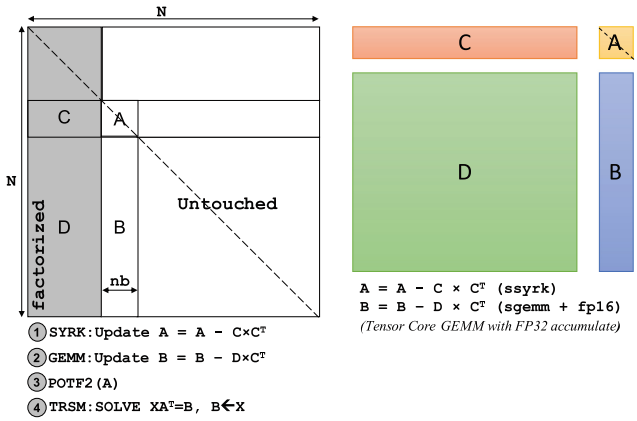


Fig. 1. Steps of a single iteration in the left-looking Cholesky factorization, as well as the mixed-precision update (`syrk` + `gemm`).

4 Mixed-Precision Cholesky Factorization

The first step in our solver is to obtain the Cholesky factorization ($A = LL^T$). This step is expected to be much faster than a factorization in FP64 or FP32. The performance advantage obtained in this step serves as an upper bound for the speedup achieved by the whole solver. As mentioned before, we use an FP32 factorization that uses mixed-precision updates. Figure 1 shows the steps of the mixed-precision factorization. Both the `potf2` and `trsm` steps are performed in FP32. We adopt the left-looking variant of the factorization, since it relies on `gemm` as the dominant operation in the update. The factorization is designed similarly to other factorizations in MAGMA. The panel step is performed on the CPU. This "hybrid execution" has the advantage of hiding the panel task on the CPU while the GPU is performing the update [18].

The `sgemm` updates are replaced by a call to a cuBLAS routine that performs an implicit FP32→FP16 conversion of the multiplicands, while accumulating the result in FP32. A tuning experiment was conducted to find the best blocking size

nb for the mixed-precision factorization. The details of the experiment are omitted for lack of space, but its final outcome suggests that setting $nb = 512$ achieves the best performance for the mixed-precision factorization. Figure 2 shows the performance of the mixed-precision Cholesky factorization (`spotrf_fp16`). The figure shows significant speedups against full-precision factorizations. In fact, the asymptotic speedup approaches 3× against single precision, and 6× compared to double precision. As mentioned before, we expect the IR phase to consume some of these performance gains.

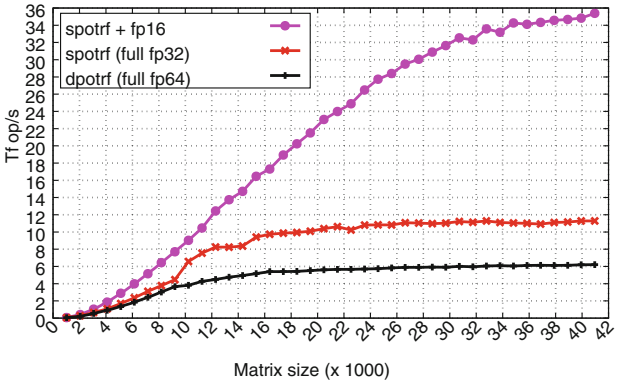
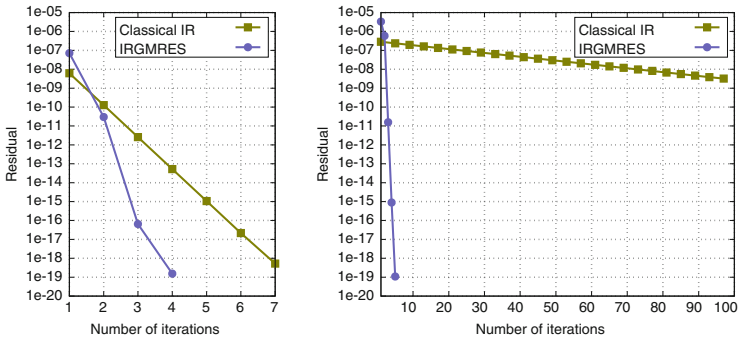


Fig. 2. Performance of the mixed-precision Cholesky factorization (`spotrf_fp16`) against full-precision factorizations in FP32 (`spotrf`) and FP64 (`dpotrf`). Results are shown on a Tesla V100-SXM2 GPU, and two 20-core Intel Broadwell CPUs.

5 GMRES-Based Iterative Refinement

The main difference between classic IR and GMRES-based IR is how the correction equation $Ac = r$ is solved. Classic IR solvers use a direct method using two triangular solves with respect to the Cholesky factors of A. This method works well for matrices with relatively small condition numbers. However, the quality of the correction vector is often impacted by the low-precision factors, which might lead to a long convergence. As mentioned in Sect. 2, it is important to keep the iteration count small in order to achieve an overall performance gain. The proposition by Carson and Higham [6,7] was to use a GMRES solver to solve $Ac = r$. The solver uses the original matrix A preconditioned by its Cholesky factors. This produces a correction vector of a much higher quality than a classic IR, eventually leading to a faster convergence. As an example, Fig. 3 shows the convergence history of both the classic IR solver and GMRES-based one (IRGMRES) for two matrices of size $10k$. The matrices share the same distribution of eigenvalues, but have different condition numbers. Our observations are (1) IRGMRES usually converges faster than classic IR, and (2) IR fails to

converge for relatively large condition numbers. However, the gap between IR and IRGMRES is not big for well-conditioned matrices. Both variants converge in few iterations, and so the final performance would be similar.



(a) $\kappa_2(A)$=1e+03 (b) $\kappa_2(A)$=1e+06

Fig. 3. Comparisons of the conversion history between IR and IRGMRES. The test matrix in both cases has a clustered distribution of eigenvalues ($\lambda_i = 1, 1, \cdots, \frac{1}{\kappa_2(A)}$).

It is worth mentioning that a conjugate gradient (CG) solver can be used instead of a GMRES solver. In fact, the study by Higham et al. [10] shows that both GMRES and CG converge within mostly similar iteration counts. However, the error analysis in [6,7] is based on the backward stability of GMRES. This means that a new error analysis is required for a CG-based IR solver, since its backward stability requires a well-conditioned matrix or a good preconditioner [8].

6 Scaling and Shifting

Higham et al. [10] proposed some countermeasures to ensure a successful factorization in FP16. The countermeasures avoid the loss of definiteness, overflow, and possibly underflow. In this study, the factorization uses two precisions (FP32 + FP16), so these countermeasures are still legitimate for our implementation. We also point out that the work done in [10] focuses only on the numerical analysis part, with no actual implementation on a high-performance hardware. Since our work focuses more on the performance, we are interested in determining the extent to which these safeguards ensure a successful factorization and convergence without too much impact on the performance. More specifically, our preprocessing works as follows:

1. **Two-sided diagonal scaling**. A lightweight GPU kernel computes the matrix $H = D^{-1}A_{fp32}D^{-1}$, where D is a diagonal matrix such that $D_{ii} = \sqrt{a_{ii}}, i = 1, \cdots, N$. This operation equilibrates the matrix rows and columns,

and reduces their range to [0, 1]. The multiplication by diagonal matrices can be simplified to a row-wise or a column-wise matrix scaling. Therefore, the GPU kernel is very lightweight with a nearly negligible execution time.

2. **An optional diagonal shift**. In order to avoid the loss of positive definiteness, the GPU kernel allows an optional small perturbation on the diagonal of H. Note that the diagonal of H is all ones. This step forms the matrix $G = H + cu_h I$, where u_h is the unit roundoff (machine epsilon) of FP16, and c is a constant parameter. The original proposition is to set c as a small positive integer constant. However, we show that this shift is sometimes unnecessary, and setting it anyway might affect the convergence of the GMRES solver. We also allow $c < 1$, since our shift occurs in FP32, where u_h is possibly a large shift to start with. We can shift by a fraction of u_h.
3. **Matrix scaling**. Finally, the entire matrix is scaled by μ, where $\mu = \frac{\theta x_{max}}{1+cu_h}$. The constant x_{max} is 6.55×10^4. The constant θ is a parameter that is set to 0.1 in all of our experiments, but in general $\theta \in (0, 1)$. The purpose of this scaling operation is to make a better use of the half-precision range. This scaling step avoids overflow and reduces the chances of underflow. Further details can be found in [10].

All of these preprocessing steps are performed by one lightweight GPU kernel. The preprocessing step obviously implies modifications in other numerical steps. In an IRGMRES solver, the matrix A is preconditioned by the Cholesky factors. However, the action of the preconditioner on a vector is obtained by a triangular solve (similar to the classical IR), and then a matrix-vector multiplication with respect to A. Noting that $A = \frac{1}{\mu} DHD$, any triangular solve ($Ap = q$) inside the GMRES solver now solves for y with respect to $D^{-1}q$ and then forms $p = \mu D^{-1}y$. Another GPU kernel that performs diagonal matrix-vector products has been developed for such a purpose.

Table 1. Eigenvalue distributions used in the test matrices.

Distribution Name	Specification ($i = 1, 2, \cdots, N$)
Arithmetic	$\lambda_i = 1 - (\frac{i-1}{N-1})(1 - \frac{1}{\kappa_2(A)})$
Clustered	$\lambda_1 = 1, \lambda_i = \frac{1}{\kappa_2(A)}$ for $i > 1$
Logarithmic	$\log(\lambda_i)$ uniform on $[\log(\frac{1}{\kappa_2(A)}), \log(1)]$
Geometric	$\lambda_i = \kappa_2(A)^{(\frac{1-i}{N-1})}$
Custom-clustered	$\lambda_i = 1$ for $i \leq \lfloor \frac{N}{10} \rfloor$, $\frac{1}{\kappa_2(A)}$ otherwise

7 Performance Results

Test Matrices and General Outlines. Our experiments use a matrix generator that is available in MAGMA, which is similar to the LAPACK routine

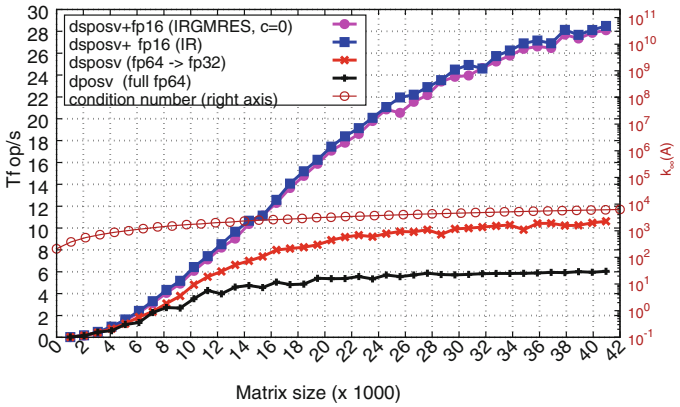
`dlatms`. It generates random dense SPD matrices with (1) a specified 2-norm condition number $\kappa_2(A)$, and (2) a specified distribution of eigenvalues. The matrix is generated as the product $A = V\lambda V^T$, where λ is the diagonal matrix of eigenvalues and V is a random orthogonal matrix. Performance results are shown for matrices with different types of distributions and different condition numbers. Table 1 shows the distributions used in this paper.

Throughout this section, the performance is measured in tera FLOPs per second (teraFLOP/s). In order to have a fair comparison, a constant number of FLOPs for each matrix size is divided by the time-to-solution of each tested solver. That constant is equal to the operation count of a full FP64 solver, which is equal to $(\frac{N^3}{3} + \frac{5N^2}{2} + \frac{N}{6})$ for one right-hand side. Performance figures have the left Y-axis with a fixed maximum value of 30 teraFLOP/s. The right Y-axis displays the infinity norm condition number ($\kappa_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty$), since this condition number is the one used in the error analysis of the IRGMRES solver [6,7]. The 2-norm condition number is constant across a single figure, and is equivalent to the ratio between the maximum and the minimum eigenvalues. We accept convergence when the residual $r = \frac{\|b-Ax\|_\infty}{N\|A\|_\infty}$ is at most $\mathcal{O}(10^{-14})$. Each performance graph features some or all of the following solvers:

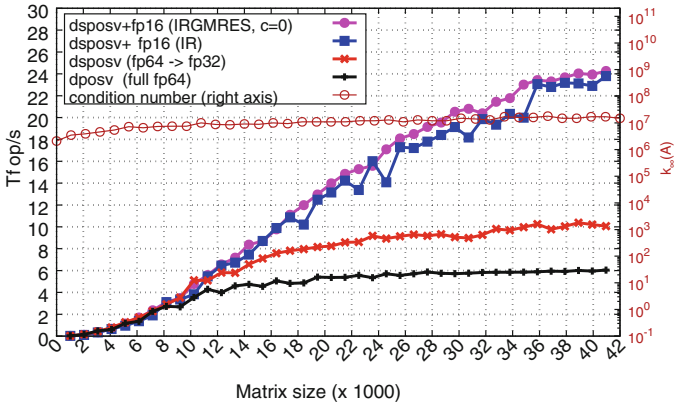
- `dposv`: a direct solver in full double precision.
- `dsposv`: a classic MP-IR solver with two precisions (FP64→FP32).
- `dsposv-fp16-ir` : our new MP-IR solver with three precisions.
- `dsposv-fp16-irgmres` : our new MP-IRGMRES solver with three precisions. This solver always scales and equilibrates the matrix, but the shift is optional. The time of the these preprocessing steps is included in the final timing of the solver.

Matrices with an Arithmetic Distribution of Eigenvalues. Figure 4a shows a "best case scenario" for a small $\kappa_2(A)$. The infinity norm condition number is capped at 10^4. Both `dsposv-fp16-ir` and `dsposv-fp16-irgmres` converge within 3 iterations at most, which yields significant performance gains. The asymptotic performance reaches 28.5 teraFLOP/s, which is 4.7× faster than `dposv`, and 2.7× faster than `dsposv`. Figure 4b shows the impact of increasing the condition number. The `dsposv-fp16-irgmres` solver converges within 7–8 iterations in most cases, while the `dsposv-fp16-ir` solver converges within 6–11 iterations, leading to performance drops at some points. The increased iteration count on both sides leads to a drop in the asymptotic performance, which is now measured at 24 teraFLOP/s. This is still 4× faster than `dposv` and 2.3× faster than `dsposv`.

Matrices with a Clustered Distribution of Eigenvalues. Figure 5a shows a performance similar to the best case scenario of Fig. 4a. However, there is a slight advantage for using the `dsposv-fp16-irgmres` solver. It converges in 3–4 iterations, while the `dsposv-fp16-ir` solver requires 3–6 iterations. The `dsposv-fp16-irgmres` solver maintains asymptotic speedups of 4.5×/2.6× against `dposv`/`dsposv`, respectively. Now we increase $\kappa_2(A)$ to 10^8, which results in $\kappa_\infty(A)$ in the range of 10^9. No convergence was achieved except for the



(a) $\kappa_2(A)$=5e+00

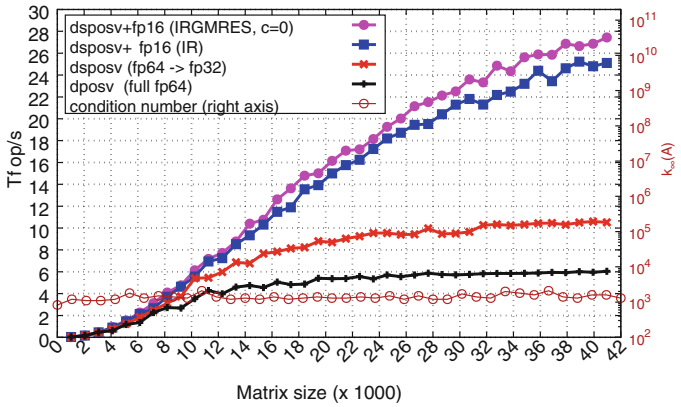


(b) $\kappa_2(A)$=1e+05

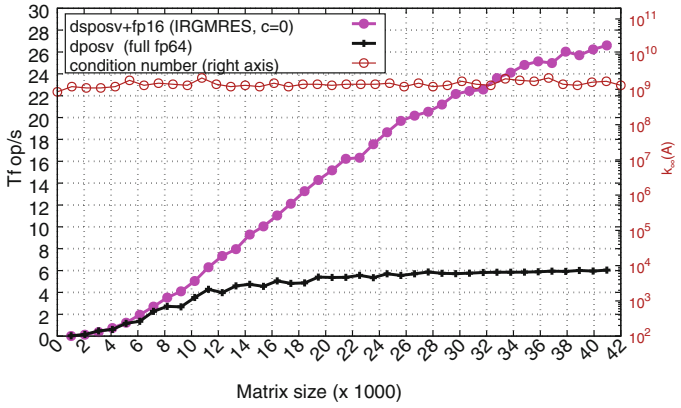
Fig. 4. Performance on matrices with an arithmetic distribution of eigenvalues.

`dsposv-fp16-irgmres` solver. This is a test case where classic IR fails in both `dsposv` and `dsposv-fp16-ir` . As Fig. 5b shows, the `dsposv-fp16-irgmres` solver requires 5 iterations for this type of matrices, leading to an asymptotic performance that is 4.4× faster than `dposv`. The result of this experiemnt also encourages using the GMRES-based IR with single-precision factorization. While this combination is not discussed this paper, the performance would be similar to `dsposv` in Fig. 4a.

Matrices with Logarithmic/Geometric Distributions of Eigenvalues. It is clear that by trying harder-to-solve matrices, the `dsposv-fp16-irgmres` solver requires more iterations, which would impact the final performance of the solver. Figure 6 shows two example for such a case, where the benefit of using half-precision is limited only to large matrices. The condition number



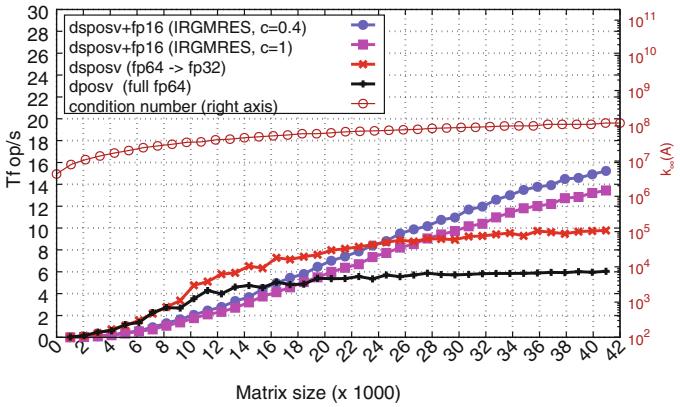
(a) $\kappa_2(A)$=1e+02



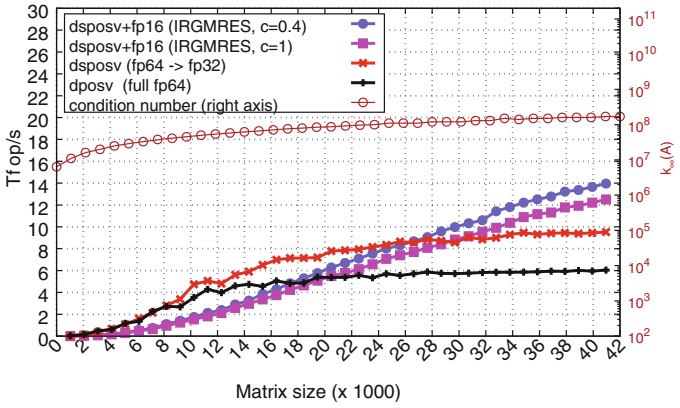
(b) $\kappa_2(A)$=1e+08

Fig. 5. Performance on matrices with a clustered distribution of eigenvalues.

$\kappa_\infty(A)$ is intentionally high to show such a behavior. Several useful observations can be taken away from these results. **First**, this is the first time we see a benefit for the matrix preprocessing stage. Both `dsposv-fp16-ir` and and the `dsposv-fp16-irgmres` (without preprocessing) fail during the factorization, meaning that the matrix loses its positive-definiteness during the mixed-precision updates. **Second**, our proposition for smaller shifts proves to achieve a better performance against limiting the constant c to an integer. **Third**, the number of iterations for the `dsposv-fp16-irgmres` solver ($c = 0.4$) is asymptotically measured at 27 for Fig. 6a, and at 32 for Fig. 6. Such large iteration counts consume most of the performance gains achieved in the factorization. Performance speedups are observed only for large matrices ($N \geq 27$k). Figure 6a shows



(a) Logarithmic distribution, $\kappa_2(A)$=1.2e+05



(b) Geometric distribution, $\kappa_2(A)$=1.7e+05

Fig. 6. Performance on matrices with logarithmic (a) and geometric (b) distributions of eigenvalues.

an asymptotic speedup of 2.5×/1.56× against `dposv`/`dsposv`, respectively. The respective speedups of Fig. 6b are measured at 2.3×/1.46×.

Matrices with a Custom-Clustered Distributions of Eigenvalues. This distribution assigns 10% of the eigenvalues to 1, and the other 90% to $\frac{1}{\kappa_2(A)}$. Figure 7 shows the results, in which the two variants of `dsposv-fp16-irgmres` (with/without preprocessing) successfully converge. However, the preprocessed solver converges within 15–16 iterations in most cases, as opposed to at least 37 iterations without preprocessing. This means that the produced Cholesky factors without preprocessing do not form a good preconditioner for A. The performance gains for the preprocessed solver are noticeable much earlier than its regular

variant. The asymptotic speedups for the preprocessed `dsposv-fp16-irgmres` are 3.3×/1.96× against `dposv`/`dsposv`, respectively.

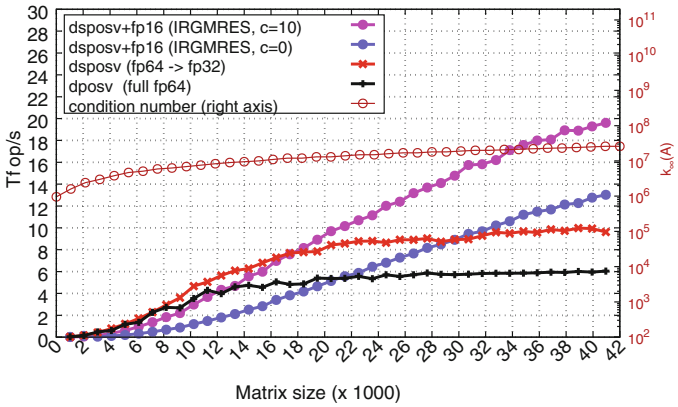


Fig. 7. Performance on matrices with a custom-clustered distribution ($\kappa_2(A) = 10^4$).

8 Conclusion and Future Work

This paper presented an FP16-accelerated dense linear solver for SPD systems. The proposed solution combines a mixed-precision Cholesky factorization with a GMRES-based iterative refinement algorithms in order to achieve double precision accuracy. Optional safeguards are developed (scaling and shifting) to ensure successful factorization and solve for matrices with relatively large condition numbers. The accelerated solver can be up to 4.7× faster than a direct solve in full FP64 precision.

Future directions include integrating the GMRES-based IR solver into dual-precision solvers (i.e., FP32→FP64), which would improve their performance for matrices with higher condition numbers. It is also useful to study the impact of the preprocessing stage (especially the diagonal shift) on the convergence of the GMRES-based IR solver. As per our results, there is no single setting that works well across the board, and each matrix has to be treated separately. Another potential direction is to add support for the complex precision (Hermitian Positive Definite systems), which requires half-complex BLAS routines.

References

1. LAPACK - Linear Algebra PACKage. http://www.netlib.org/lapack/
2. IEEE standard for floating-point arithmetic. IEEE Std 754–2008, pp. 1–70, August 2008. https://doi.org/10.1109/IEEESTD.2008.4610935. https://ieeexplore.ieee.org/document/4610935

3. Abdelfattah, A., Tomov, S., Dongarra, J.J.: Fast batch matrix multiplication for small sizes using half precision arithmetic on GPUs. In: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, 20–24 May 2019, pp. 111–122 (2019)
4. Agullo, E., et al.: Numerical linear algebra on emerging architectures the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**(1), 012937 (2009)
5. Baboulin, M., et al.: Accelerating scientific computations with mixed precision algorithms. Comput. Phys. Commun. **180**(12), 2526–2533 (2009)
6. Carson, E., Higham, N.: A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. SIAM J. Sci. Comput. **39**(6), A2834–A2856 (2017). https://doi.org/10.1137/17M1122918
7. Carson, E., Higham, N.: Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM J. Sci. Comput. **40**(2), A817–A847 (2018). https://doi.org/10.1137/17M1140819
8. Greenbaum, A.: Estimating the attainable accuracy of recursively computed residual methods. SIAM J. Matrix Anal. Appl. **18**(3), 535–551 (1997). https://doi.org/10.1137/S0895479895284944
9. Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2018), pp. 47:1–47:11. IEEE Press, Piscataway (2018). https://doi.org/10.1109/SC.2018.00050
10. Higham, N., Pranesh, S.: Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. Technical report 1749–9097, November 2019. http://eprints.maths.manchester.ac.uk/2736/
11. MAGMA: Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/magma/
12. Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Dongarra, J.J.: Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In: Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, 11–17 November 2006, Tampa, FL, USA. p. 113 (2006). https://doi.org/10.1145/1188455.1188573
13. NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS). https://developer.nvidia.com/cublas
14. NVIDIA cuSOLVER: A Collection of Dense and Sparse Direct Solvers. https://developer.nvidia.com/cusolver
15. Saad, Y.: A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput. **14**(2), 461–469 (1993). https://doi.org/10.1137/0914028
16. Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**(3), 856–869 (1986). https://doi.org/10.1137/0907058
17. Simoncini, V., Szyld, D.: Flexible inner-outer Krylov subspace methods. SIAM J. Numer. Anal. **40**(6), 2219–2239 (2002). https://doi.org/10.1137/S0036142902401074
18. Tomov, S., Dongarra, J.J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5–6), 232–240 (2010). https://doi.org/10.1016/j.parco.2009.12.005