Towards Half-Precision Computation for Complex Matrices: A Case Study for Mixed-Precision Solvers on GPUs

Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
{ahmad,tomov,dongarra}@icl.utk.edu

***Abstract*—The use of low-precision computations is popular in accelerating machine learning and artificial intelligence (AI) applications. Hardware architectures, such as high-end graphics processing units (GPUs), now support native 16-bit floating-point arithmetic (i.e., half-precision). While half precision provides a natural $2\times/4\times$ speedup against the performance of single/double precisions, respectively, modern GPUs are equipped with hardware accelerators that further boost the FP16 performance. These accelerators, known as tensor cores (TCs), have a theoretical peak performance that is $8\times/16\times$ faster than FP32/FP64 performance, respectively. Such a high level of performance has encouraged researchers to harness the compute power of TCs outside AI applications.**

This paper presents a mixed-precision dense linear solver ($Ax = b$) for complex matrices using the GPU's TC units. Unlike similar efforts that have discussed accelerating $Ax = b$ in real FP16 arithmetic, this paper focuses on complex FP16 precisions. The developed solution uses a "half-complex" precision to accelerate the solution of $Ax = b$ while maintaining complex FP32 precision accuracy. The proposed solver requires the development of a high-performance mixed-precision matrix multiplication (CGEMM-FP16) that accepts half-complex inputs, and uses the TCs' full-precision products and FP32 accumulations for the computation. We discuss two designs and their performance. Similar to the way fast GEMMs power the performance of LAPACK, the mixed-precision CGEMM-FP16 can enable the development of mixed-precision LAPACK algorithms. We illustrate this by integrating both CGEMM-FP16s into the development of mixed-precision LU factorizations of complex matrices. Finally, an iterative refinement solver is used to deliver complex FP32 accuracy using a preconditioned GMRES solver. Our experiments, conducted on V100 GPUs, show that the mixed-precision solver can be up to $2.5\times$ faster than a full single-complex precision solver.

***Index Terms*—Half precision, Tensor cores FP16 arithmetic, mixed-precision solvers**

I. Introduction and Related Work

Many scientific computing applications require the solution of a linear system of equations $Ax = b$, where A is a general dense matrix, b is a right-hand side vector, and x is the solution vector. The same problem is used in the High Performance LINPACK (HPL) benchmark,[1] which is the standard benchmark that is used to rank the fastest 500 supercomputers in the world.[2] The standard LAPACK software [1] provides the `gesv` routine for solving $Ax = b$.

The `gesv` algorithm consists of two main stages. The first one is called `getrf`, which factorizes the matrix A using the LU factorization with partial pivoting. The factorization yields the L and U factors, as well as the pivoting vector $ipiv$. The second stage is `getrs`, which solves the linear system by applying the row interchanges on the right-hand side (`laswp`), and performing two triangular solves (`trsm`) with respect to L and U. The factorization step is usually the dominant one, especially when the number of right-hand sides (`nrhs`) is small. In fact, given an $n \times n$ matrix, the `getrf` routine requires $\frac{2n^3}{3} - \frac{n^2}{2} + \frac{5n}{6}$ floating-point operations (FLOPs). The operation count of the `getrs` routine is `nrhs` $\times(2n^2 - n)$. Throughout the paper, we consider `nrhs` = 1.

Since LU factorization is the dominant step with $\mathcal{O}(n^3)$ complexity, there have been several research efforts to optimize its performance on parallel architectures. Apart from the traditional "parallel optimizations" that target faster execution of critical components of the algorithm—such as matrix multiplication (`GEMM`)—there have been other "algorithmic optimizations" that change the numerical steps of the algorithm itself. One particular algorithmic improvement to `gesv` is to perform the factorization using a "reduced precision." The factorization step would then be much faster, due to the reduced data movement as well as the faster execution of the numerical kernels. As an example, a transition from double precision (64-bit) to single precision (32-bit) execution would witness a natural $2\times$ speedup in the factorization step. However, due to the loss of accuracy in the L and U factors, the final result of `gesv` will no longer satisfy the double-precision accuracy. This is why an extra algorithmic component, in fact a correction step, is added to `gesv`. The correction step iteratively recovers the solution back to double-precision accuracy if the original matrix A satisfies certain conditions. The redesigned algorithm is called a *mixed-precision with iterative refinement (MPIR)* solver, since it uses two different precisions and an iterative procedure to

[1]https://www.netlib.org/benchmark/hpl/

[2]https://www.top500.org

achieve the required accuracy. Early efforts to implement such algorithms in LAPACK were introduced by Langou et al. [2], and Baboulin et al. [3].

The ongoing revolution in machine learning applications and artificial intelligence (AI) sparked a huge demand for high-performance, half-precision arithmetic (16-bit floating-point format). Such a demand is due to the fact that most machine learning algorithms can tolerate the relatively low accuracy and dynamic ranges of half precision [4]. Running at half precision also means more performance, not only because of the faster arithmetic, but also because of the reduction in memory storage and traffic. NVIDIA's graphics processing units (GPUs) introduced half-precision arithmetic with the Pascal architecture. Pascal implements the "binary16" format which is defined by the IEEE-754 standard [5]. The Pascal successor is the Volta architecture, which already powers a number of supercomputers, including the top two supercomputers from the top500 list—the number one, Summit at Oak Ridge National Laboratory (ORNL),[3] and number two, Sierra at Lawrence Livermore National Laboratory (LLNL).[4] Volta comes with hardware acceleration units (called Tensor Cores) for matrix multiplication in FP16. These tensor cores are theoretically $12\times$ faster than the theoretical FP16 peak performance of the preceding architecture. Applications taking advantage of tensor cores can run up to $4\times$ faster than using the regular FP16 arithmetic on the same GPU. The vendor BLAS library, cuBLAS [6], provides a number of matrix multiplication routines that can take advantage of tensor cores. Some other efforts introduced open source routines that can be competitive with cuBLAS [7].

One of the most notable uses of half precision in dense linear algebra is the design of a new generation of mixed-precision solvers. The work done by Haidar et al. [8] introduced a mixed-precision solver that is different in several ways from the ones introduced in [2] and [3]. **First**, the new method uses three precisions (double, single, and half precisions) to solve $Ax = b$ up to double-precision accuracy. **Second**, the factorization step is performed in mixed precision, where everything is computed in single precision except the trailing matrix updates, which are performed using mixed half- and single-precision arithmetic. More precisely, the authors use a "mixed-precision GEMM" from the cuBLAS library, which accepts the multiplication operands in half precision and provides the product in single precision. **Third**, the authors use a new iterative refinement solver based on the GMRES method [9] [10], instead of the classic iterative refinement that is based on triangular solve. The new mixed-precision solver succeeds in solving problems up to double-precision accuracy, with up to $4\times$ speedup against an all double-precision implementation.

In this paper, we extend the mixed-precision solver introduced in [8] to complex matrices. The main challenge of this work is the absence of support to half-complex arithmetic on both hardware and software levels. This drives us to develop a GPU kernel for mixed-precision GEMM that accepts half-complex inputs, and produces a single-complex output. The developed kernel is plugged into a mixed-precision linear solver that can solve $Ax = b$ up to single-complex precision accuracy. The solver uses an iterative refinement technique based on a preconditioned GMRES method. The latter is a more stable way to compute the incremental corrections that are added iteratively to the solution vector $\hat{x}$. Our results show that the developed solution is up to $2.5\times$ faster than a regular solver that uses a full single-complex precision. This work is lined up for integration into the MAGMA library [11] [12].

II. BACKGROUND

Before the introduction of half-precision in modern GPUs, mixed-precision solvers did not target accelerating FP32 systems (both single and single-complex) due to the lack of a lower precision. To the best of our knowledge, this is the first effort that targets accelerating complex systems using "half-complex" precision.

Classic mixed-precision solvers [2] [3] used to perform the entire LU factorization in a reduced precision (e.g., FP32). The refinement phase iteratively updates the solution vector $\hat{x}$ until it is accurate enough. At each refinement iteration, three main steps are performed. The first one is to compute the residual $r = b - Ax$ in the working precision (e.g., FP64). The second step is to solve for the correction vector c, such that $Ac = r$. This step uses the low precision L and U factors of A. The last step is to update the solution vector $\hat{x}_{i+1} = \hat{x}_i + c$. The three steps are repeated until the residual is small enough.

The work done by Haidar et al. [8] shows that a similar approach is not often successful when FP16 is considered. The low accuracy and dynamic range of FP16 make it difficult for classic mixed-precision solvers to converge successfully. The proposition was then to perform a *mixed-precision LU factorization*. Such a factorization uses FP32 arithmetic except for the performance-critical rank-k updates. These updates ($C = C - A \times B$) are performed using a mixed-precision GEMM kernel from the cuBLAS library, such that A and B are demoted to FP16, and C is accumulated in FP32.

Even though the factorization was more accurate than a full FP16 factorization, the classic iterative refinement (`IR`) technique often fails to converge (e.g., following the classic mixed precision solvers' convergence theory [2]).

An alternative approach, which further improves the numerical stability and convergence of the overall mixed-precision solver, is to solve the correction equation ($Ac = r$) using an iterative method, such as GMRES [13]. The resulting overall solver thus uses two nested refinement loops, which is also often referred to as "inner-outer" iterative solvers [14] [15]. The recent work by Carson and Higham [9] [10] analyzes this type of solver 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

[3]https://www.olcf.ornl.gov/summit

[4]https://computing.llnl.gov/computers/sierra

order of $10^{-8}/10^{-16}$ are achievable if the condition number of A satisfies $k_\infty(A) < 10^8/10^{12}$, respectively. The work in [8] 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.

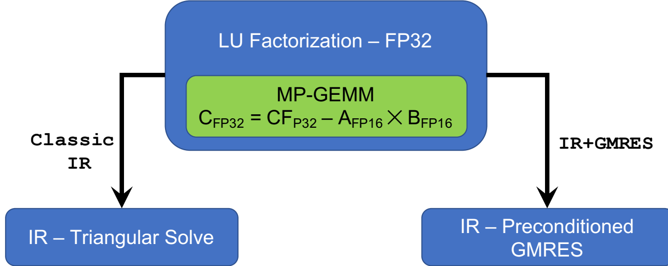

Fig. 1. A simplified overview of the proposed mixed-precision solver

Figure 1 shows the proposed solution to extend the work in [8] to support complex precisions. Our focus is to solve $Ax = b$ up to single-complex precision accuracy. In this context, we need an LU factorization in single-complex precision (`cgetrf`) that uses a mixed-precision complex GEMM $C = C - A \times B$, such that A and B are demoted to half-complex precision, while C is accumulated in single-complex precision. In addition, we implement both the classic `IR` and GMRES-based IR (`IRGMRES`) solvers for comparisons of the numerical behavior. Note that both iterative refinement solvers are implemented using one precision (single-complex). Although the paper does not discuss double-complex precisions, its support is relatively straightforward. The same factorization routine would be used without change, and the new required component is the GMRES solver in double-complex precision. In the following sections, we discuss the main components of our design, with an emphasis on the different design choices that would impact the overall performance.

III. Mixed-Precision Matrix Multiplication

The main performance key in our design is the mixed-precision GEMM kernel. While the cuBLAS library supports such an operation for real matrices, no similar functionality exists for complex problems. In order to address this problem, we consider two different approaches based on the data layout.

A. Interleaved vs. Split-Complex layouts

In order to use the cuBLAS library, a user must use "split-complex" computation, meaning that the real and the imaginary parts of A, B, and C must be separated and then merged back when performing the rank-k updates. This applies not only to the cuBLAS library, but also to its lightweight GEMM library cuBLASLt[5], and to the open-source CUTLASS library.[6] Considering dense linear algebra algorithms, split-complex computations are far from beneficial due to the following reasons. **First**, all the existing linear algebra numerical libraries assume an *interleaved layout*, meaning that the real and the imaginary parts of each element are contiguous in memory. It is not practical to rewrite entire algorithms using split-complex computations to make use of the tensor cores. **Second**, while it is relatively easy to develop the GEMM kernel using the split-complex format, other linear algebra components might not be as straightforward. Examples are triangular solve and the pivoting stage in the LU factorization. In these routines, it is more convenient to use the standard interleaved layout. **Third**, complex compute-bound kernels normally reach the peak performance of the underlying hardware earlier than their counterparts that use real arithmetic. This is because complex kernels have more arithmetic intensity than real arithmetic kernels. As an example, the real FP16 scalar operation ($c = c + a \times b$) costs 2 FLOPs (1 addition and 1 multiplication), which results in $\frac{2}{8} = 0.25$ FLOP/byte ratio. Using complex FP16 arithmetic, the same operation would cost 8 FLOPs. The product $a \times b$ costs 6 FLOPs (4 multiplications and 2 additions), and the update of c costs 2 more additions. This results in $\frac{8}{16} = 0.5$ flop/byte ratio (double the arithmetic intensity of the real operation). To justify this experimentally, Figure 2 shows the performance, relative to the theoretical peak performance, of the cuBLAS `sgemm` and `cgemm` routines. The performance test is conducted using the rank-k update operation that exists in the LU factorization. The figure shows that the `cgemm` routine is closer to the peak performance than the `sgemm` routine. Using split-complex computation results in lower arithmetic intensity, and potentially lower performance. In other words, a split-complex `cgemm` will be bound by the performance of the `sgemm` kernel.

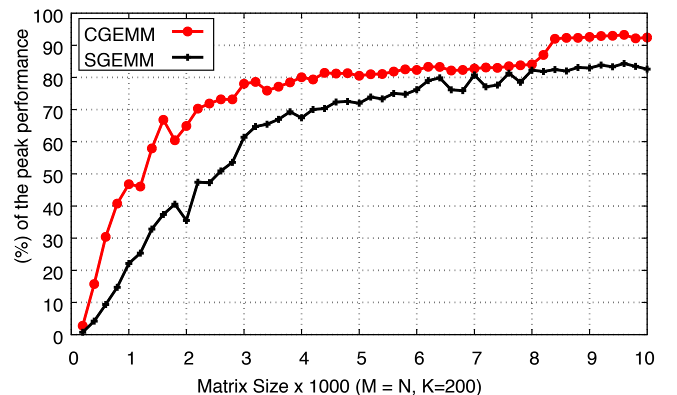

Fig. 2. Performance (as percentage of the peak) of the cuBLAS `sgemm` and `cgemm` routines.

B. Mixed-Precision Half-Complex GEMM

This is why we developed a new GPU kernel that performs *mixed-precision complex GEMM*. The kernel accepts A and B in *half-complex* precision, and produces C in *single-complex* precision. Similar to [7], the kernel uses the tensor core

[5]https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasLt-api
[6]https://github.com/NVIDIA/cutlass

operations by calling the *CUDA WMMA* device functions. Since there is no standard type for half-complex precision, we use the CUDA vector type (`half2`), which exactly fits our requirements for the kernel development. The low 16-bits serve for the real part, and the high 16-bits serve for the imaginary part. Before calling the kernel, the input matrices A and B are demoted from single-complex precision to half-complex precision using lightweight kernels. Since the tensor core units perform only real arithmetic, the developed kernel is built on the concept of reading the matrices in the standard interleaved layout (half-complex precision), and then splitting the real and imaginary parts just before sending them to the tensor cores. The resulting output of the tensor cores is merged back to single-complex before writing it to the main memory.

The work distribution across thread blocks is similar to previous GEMM designs in the MAGMA library [16], [17]. Each thread block is responsible for computing a `BLK_M`×`BLK_N` block of the matrix C. To accomplish this, each thread block reads the corresponding block rows and block columns of A and B. The design of each thread block is drastically different from traditional GEMM kernels. The traversal of A and B is made in steps of `BLK_K`. At each step, a `BLK_M`×`BLK_K` of A (half-complex precision) is multiplied by `BLK_K`×`BLK_N` of B (half-complex precision) to produced a partial result that is accumulated to a block of C (single-complex precision). The A and B blocks are read first into `half2` register arrays, and then the real and imaginary parts are separated into shared-memory buffers. The shared-memory buffers can then be passed to the device-level tensor core functions. The partial product for $A \times B$ is conducted using four tensor core multiplications ($A_r \times B_r$, $A_r \times B_i$, $A_i \times B_r$, and $A_i \times B_i$). To extract the results from tensor cores, shared-memory buffers are used. Register arrays are then used to assemble the final result into single-complex precision. A thread block uses a 2D `DIM_X`×`DIM_Y` thread configuration. Since the tensor cores support very few GEMM sizes, we use the double-sided recursive blocking technique [7] to decouple the GEMM blocking sizes (`BLK_M`, `BLK_N`, and `BLK_K`) from the tensor core sizes (`TC_M`, `TC_N`, and `TC_K`).

In order to have a fair study, we developed a similar routine based on the cuBLAS library. The latter solution uses the mixed-precision GEMM kernel in cuBLAS that assumes A and B are in half precision, and C is single precision. We developed simple lightweight kernels to separate and merge the real and imaginary components in the main memory. The rank-k update in the LU factorization is $C = C - A \times B$, which can be performed using four calls to cuBLAS.

In order to have a fair comparison, we developed two GEMM driver routines that can be plugged directly into the standard LU factorization algorithm (`cgetrf`). The requirements are: (1) Single-complex precision inputs and outputs (e.g., like a standard `cgemm`); (2) Converge to/from half-precision underneath transparently; and (3) Standard “interleaved” layout for both inputs and outputs so that we can conveniently execute other steps of the LU factorization algorithm. The first of the two driver routines uses our developed *MAGMA kernel* with the necessary conversions from single-complex to half-complex precisions. The second driver routine uses cuBLAS with the necessary precision conversion routines, as well as the split and merge routines.

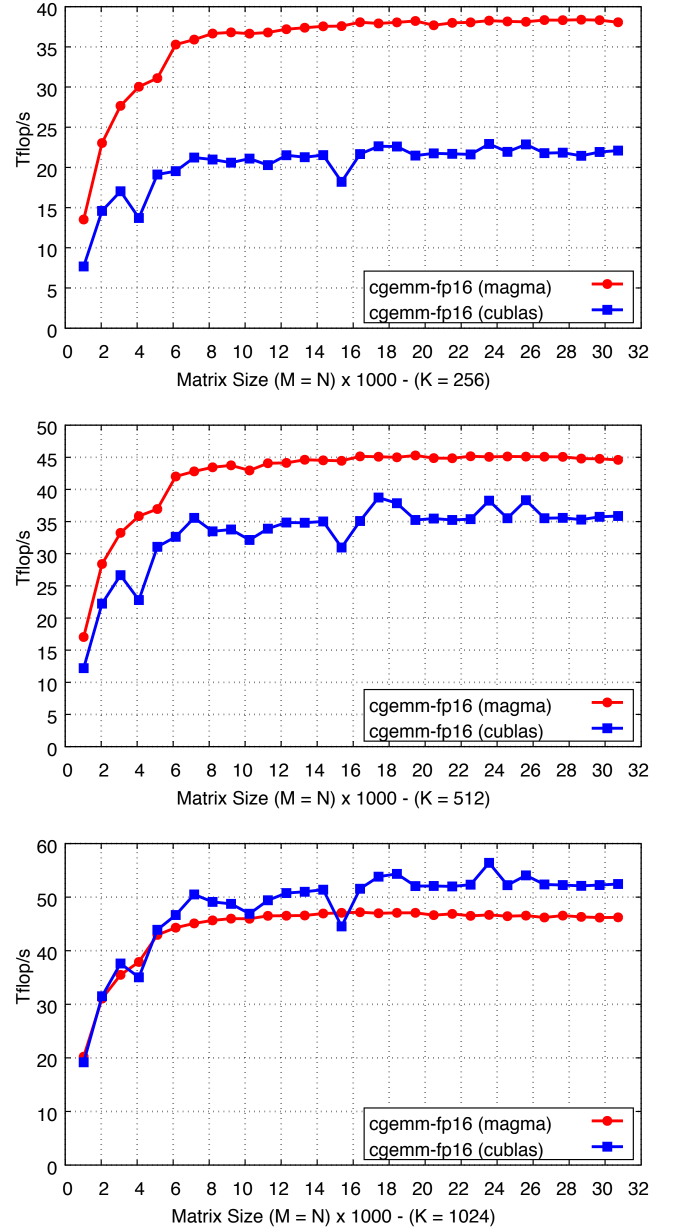

Fig. 3. Performance of the `cgemm` routine when accelerated internally with FP16 arithmetic. Results are shown for rank-k updates on a Tesla V100 GPU.

Figure 3 shows the performance comparisons between the MAGMA-based routine (`cgemm-fp16-magma`), and the cuBLAS-based routine (`cgemm-fp16-cublas`). All performance comparisons represent rank-k updates for discrete values of k (i.e., the blocking size of the factorization). Typically, the blocking size is a tuning parameter for the algorithm, and is chosen to guarantee a close-to-the-peak GEMM performance without causing a “too wide” panel during the factorization

stage. Typical values for k are 256 to 512 in single-complex precision.

Ideally, the MAGMA routine should outperform the cuBLAS routine regardless of the matrix sizes, due to the increased arithmetic intensity. This is observed for relatively small k, with around 70% performance advantage for $k = 256$, and around 24% improvement for $k = 512$. However, we observe that the cuBLAS routine has some winning scenarios when k is larger than ≈ 850. The explanation to this behavior is two-fold. First, the cuBLAS SGEMM kernel is highly optimized with assembly-based implementations [18] [19]. Its performance scales very well for large sizes, with an asymptotic performance that is close to the GPU peak. Second, the MAGMA kernel uses device-level application programming interfaces (APIs) for utilizing the tensor cores. These APIs impose some restriction on the multiplication sizes, as well as the leading dimensions of the shared-memory buffers, which creates a huge number of shared-memory bank conflicts.

IV. Factorization Strategy and Performance

Another aspect of the developed solution is the execution strategy of the LU factorization. For almost a decade, the LU factorization in the MAGMA library was designed to take advantage of both the CPU and GPU. This *hybrid design* uses a *lookahead* technique that splits the trailing matrix update into two updates. The first one updates the next panel, which is sent to the CPU. The CPU factorizes the received panel while the GPU is performing the second (and usually larger) update. To achieve the best performance out of this design, the GPU must be kept busy performing rank-k updates, which means that execution time of the second update on the GPU must be greater than the time needed to factorize the panel plus the time needed send the panel back and forth between CPU and the GPU.

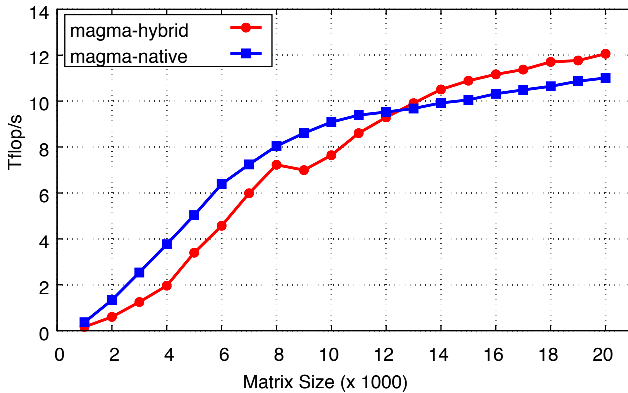

Fig. 4. Performance of the single-complex LU factorization in the MAGMA library. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU. MAGMA is built using CUDA 10.1 and MKL 2018.0.1

Hybrid designs require a relatively fast CPU so that the GPU does not go into idle states. If the panel factorization on the CPU is not fast enough, hybrid factorization might experience a slowdown in performance. This issue has been addressed with the introduction of native (GPU-only) factorizations, which do not rely on the CPU except for scheduling the execution on the GPU. Native LU factorization has been discussed in [20]. It can be faster than the hybrid design if the CPU does not send the factorized panel back to the GPU on time. Figure 4 shows a comparison between both designs for the single complex LU factorization (`cgetrf`). Although the hybrid design uses a very optimized CPU factorization (from the Intel MKL library), it trails the performance of the native factorization for small sizes. This is because the trailing updates are very fast on the GPU (mostly `cgemm` calls). The hybrid design needs a large matrix so that it can hide the CPU activity within the GPU workload.

A. Mixed-Precision LU Factorization

Now we investigate what happens when we replace the `cgemm` calls in the LU factorization with the mixed-precision kernels discussed in Section III. Recall that we have two solutions (based on MAGMA and cuBLAS), as well as two factorization strategies (hybrid and native), which yields four combinations in total. Whenever cuBLAS is used, the blocking size of the factorization is set to at least 1024. Otherwise, it will be slower than its MAGMA variant anyway. When the MAGMA mixed-precision GEMM is used, the blocking size is set to at most 512.

We begin with Figure 5, which shows the performance results for native factorization. The performance graphs of the two mixed-precision factorizations are nearly the same. There is a very slight advantage for using `magma-cgemm-fp16` for matrices smaller than 20k in size. Such an advantage goes to cuBLAS for larger matrices. Both factorizations have about $2.5\times$ speedup against the full-precision factorization. An interesting observation here is that despite the clear advantage for the cuBLAS-based GEMM for a 1024 blocking size (Figure 3), this advantage does not quite propagate to the native factorization. Recall that we must use large blocking sizes for `cublas-cgemm-fp16` so that it is advantageous over `magma-cgemm-fp16`. This large blocking size results in very wide panels that are being factorized in full precision.

Figure 6 shows the performance results for hybrid execution, where we observe a clear advantage for using `magma-cgemm-fp16`. As mentioned before, very wide panels can cause performance drops. In hybrid factorizations, these wide panels are factorized on the CPU rather than on the GPU. Moreover, since the trailing updates are now much faster due to the use of half precision, it becomes more difficult to hide the CPU activity, which is done in full precision. This results in idle states for the GPU, and an overall slowdown in performance. Since we can use thinner panels for `magma-cgemm-fp16` without observable performance penalties, the CPU has less work to do in full precision, which means more opportunity for hiding its workload. There is a 25% performance improvement when `magma-cgemm-fp16` is used for hybrid factorizations.

By combining Figures 5 and 6, we observe that, for matrix sizes smaller than 20k, it is best to use native factorization

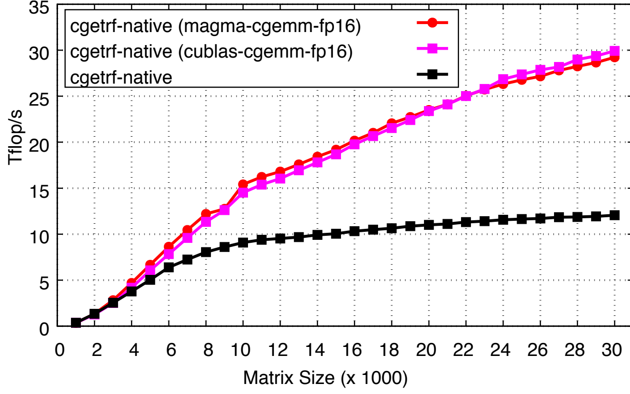

Fig. 5. Performance of the mixed-precision LU factorization using native execution. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU. MAGMA is built using CUDA 10.1 and MKL 2018.0.1

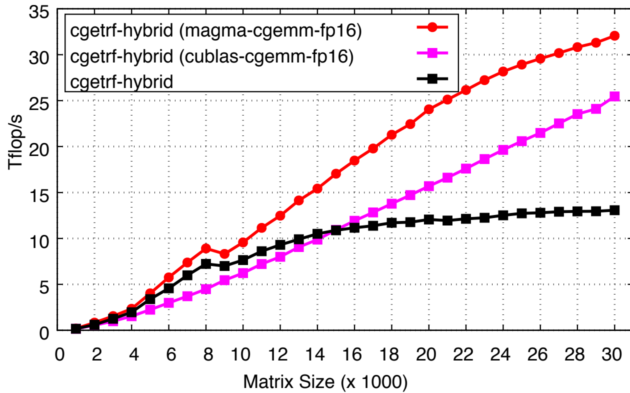

Fig. 6. Performance of the mixed-precision LU factorization using hybrid execution. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU. MAGMA is built using CUDA 10.1 and MKL 2018.0.1

with the MAGMA GEMM kernel. For sizes larger than that, we should switch to hybrid execution using the MAGMA GEMM kernel. Note that these decisions vary from one system to another. For example, if a faster CPU is used along with an optimized CPU software, hybrid factorizations may have the advantage over a wider range of sizes. On the other hand, a slower CPU may force abandoning hybrid factorizations altogether.

V. Iterative Refinement: Classic vs. GMRES-Based

Now that we have discussed the factorization stage, our attention now turns to the solution stage. The detailed discussion about the numerical behavior of `IR` and `IRGMRES` can be found in [8]. We experimentally tested the two approaches for the solution phase on several types of matrices. Our conclusion is that `IR` converges only for problems with very small condition numbers. Otherwise, the `IRGMRES` provides a more stable solution that works well for condition numbers in the order of 10^5. As an example, Figure 7 shows the convergence of both solvers on a relatively large matrix with $k_\infty(A)= 10^5$ and clustered singular values. The classic `IR` technique has a very slow convergence rate and eventually fails to reach a solution within the allowed number of iterations. The `IRGMRES` converges within just 10 iterations. Our final solution, therefore, will use `IRGMRES` only for a more robust numerical behavior.

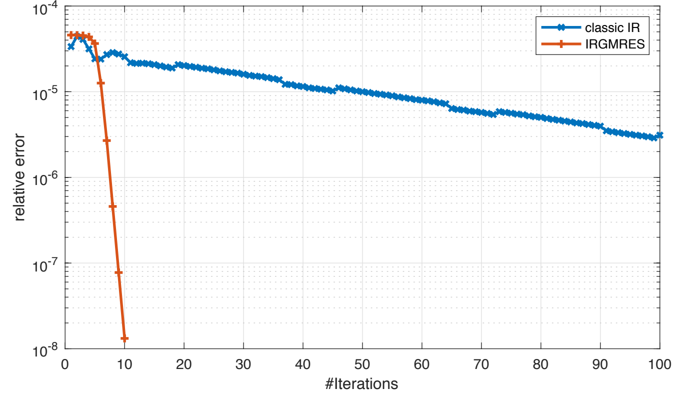

Fig. 7. Convergence history of both `IR` and `IRGMRES` on a matrix of size 20k. $k_\infty(A)= 10^5$. Clustered distribution of singular values ($\sigma_i = 1, 1, \cdots, \frac{1}{k_\infty(A)}$).

VI. Final Performance Results

This section shows the final performance of our mixed-precision complex solver (`MP-cgesv`). The developed solution solves $Ax = b$ up to single-complex precision accuracy using a mixed-precision LU factorization. The latter is accelerated using mixed-precision matrix multiplication that utilizes the FP16 compute power of the tensor cores on the GPU. All performance tests are conducted on a dual-socket Intel Haswell CPU, with 10 cores per socket (Intel Xeon E5-2650 v3 running at 2.3GHz), and a Tesla V100 PCIe GPU. The solution is developed as part of the MAGMA library, which is compiled using CUDA 10.1 and Intel MKL 2018.0.1 (for fast hybrid executions). Similar to the work done on real precisions [8], the performance tests span different types of matrices with different properties and different condition numbers.

Figure 8 shows the final performance in a “best case” scenario. The matrices used in this test are diagonally dominant with a small condition number. This type of matrices are numerically easy to solve. The solution phase with the preconditioned GMRES requires no more than one iteration to converge. The overhead of the solution phase is, therefore, almost negligible compared with the factorization phase. We observe an asymptotic speedup of $2.5\times$ against the classic `cgesv` using fixed single-complex precision.

Figure 9 shows the performance on another type of easy problem. The matrices used in this test have positive eigenvalues and their singular values have an arithmetic distribution, such that $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k_\infty(A)}), i = 1 \cdots n$. The condition number is order of 10^5. The preconditioned GMRES solver requires 3 iterations on average to converge. The overhead of the solution phase is still minimal, but the

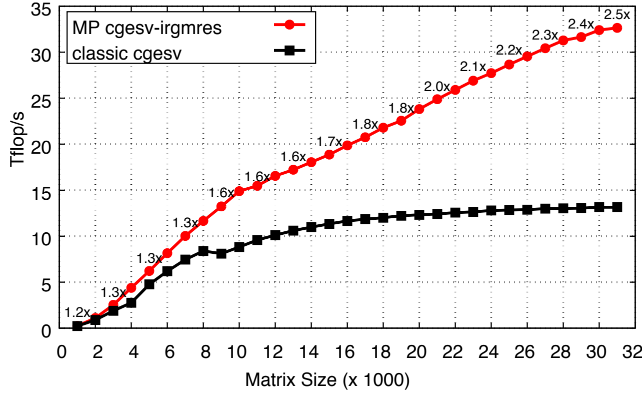

Fig. 8. Performance for diagonally dominant matrices. $k_\infty(A) \leq 10^2$. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU.

performance is not quite as high as in Figure 8. The asymptotic speedup is reduced to $2.4\times$.

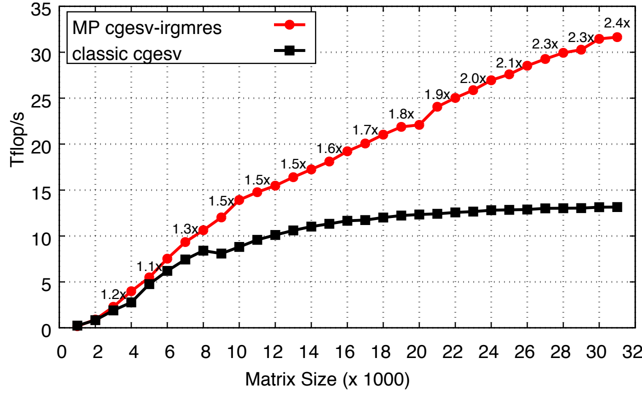

Fig. 9. Performance for matrices with positive eigenvalues and arithmetic distribution of singular values ($\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k_\infty(A)})$), $k_\infty(A)$≈4.3e+5. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU.

In Figure 10, we slightly increase the difficulty by using a logarithmic uniform distribution of singular values, while maintaining positive eigenvalues and a condition number of 10^5. The new distribution of eigenvalues indeed affects our solution. We observe speedups only for matrices larger than 8k in size. The total number of iterations is between 9 and 12 across all the test points. This results in about a 10% performance drop with respect to the best case scenario in Figure 8, and an asymptotic speedup of $2.3\times$ against the classic `cgesv` routine.

We discuss Figures 11 and 12 together. Figure 11 shows a similar performance behavior on a different type of problem. The matrices have a condition number in the order of 10^4, with a clustered distribution of singular values. The eigenvalues are not necessarily positive or real. The GMRES-based refinement stage requires 9 to 10 iterations, and an asymptotic speedup of $2.2\times$ is observed. In Figure 12 we maintain the same condition number, but change the singular-value distribution. This is

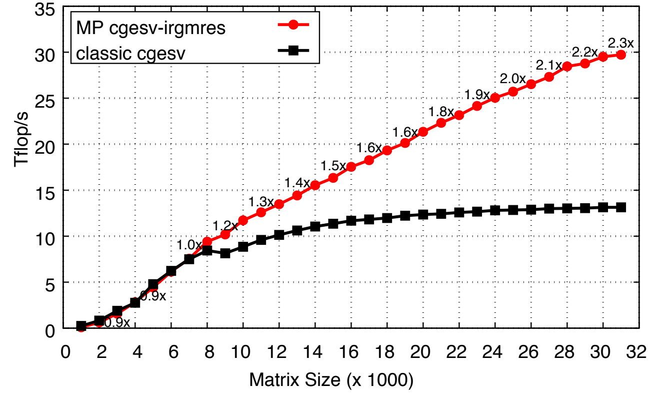

Fig. 10. Performance for matrices with positive eigenvalues and logarithmic uniform distribution of singular values ($log(\sigma_i)$ uniform between $\log(\frac{1}{k_\infty(A)})$ and $\log(1)$), $k_\infty(A)$≈1e+5. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU.

a possible "worst case" scenario, where we barely observe any performance advantage for mixed-precision solvers. The required number of iterations grows consistently with the matrix size, ranging between 7 and 135 iterations. The large number of iterations consumes the performance advantage of the factorization phase, which causes the overall performance to significantly drop. A slim 10% speedup is observed for problems larger than 22k×22k. Below this size, it is actually better to use the full-precision solver.

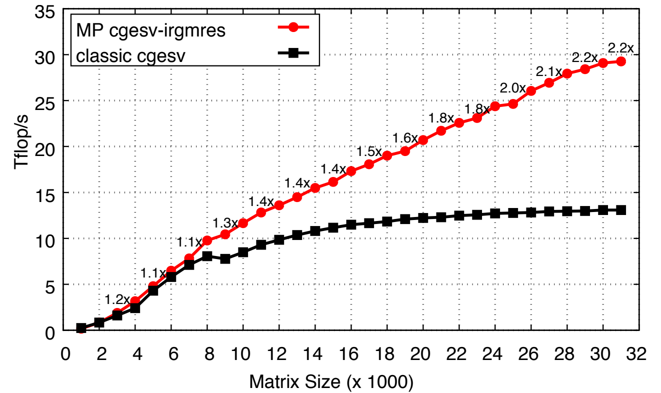

Fig. 11. Performance for matrices with clustered singular values ($\sigma_i = 1, 1, \cdots, \frac{1}{k_\infty(A)}$), $k_\infty(A)$≈4.3e+4. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU.

The take away message from this section is that the performance of mixed-precision solvers depends on several matrix properties, such as the condition number, the distribution of singular values, and the eigenvalue properties. In general, a very large problem gives an advantage for mixed-precision solvers. The performance gain in the mixed-precision factorization stage is significant, which gives room to execute more GMRES iterations without a big impact on the overall performance.

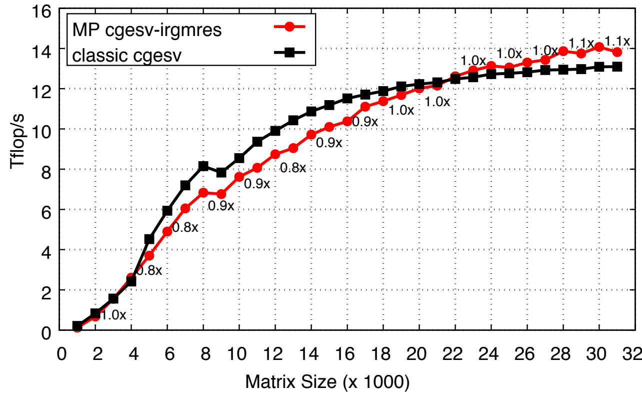

Fig. 12. Performance for matrices with arithmetic distribution of singular values ($\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k_\infty(A)})$), $k_\infty(A) \approx$4.3e+4. Results are shown on a 20-core Haswell CPU and a Tesla V100 GPU.

VII. Conclusion and Future Work

This paper introduced a mixed-precision linear solver that can accelerate the solution of complex linear systems by performing half-complex computations on the GPU. The developed solution solves $Ax = b$ up to single-complex precision accuracy by taking advantage of the tensor core units on NVIDIA Volta GPUs. Thanks to a complex mixed-precision GEMM kernel, the LU factorization runs up to $2.5\times$ faster than a full-precision factorization. Iterative refinement based on a preconditioned GMRES ensures numerical stability across a wide range of problems, and enables the overall solver to run up to $2.5\times$ faster than the full-precision solver.

Solving up to double-complex precision accuracy is relatively simple. The factorization stage remains the same, while only the `IRGMRES` needs to be rewritten to compute the correction equation in double-complex precision. Other future directions include a comprehensive autotuning of the mixed-precision GEMM kernel, and improving the numerical robustness of the solver by means of matrix scaling [21]. We also hope that our results encourage the vendors to consider standard half-complex computation using interleaved data layouts.

Acknowledgment

This work is partially supported by NSF grant No. OAC-1740250 and CSR 1514286, NVIDIA, and by the Exascale Computing Project (17-SC-20-SC).

References

[1] “LAPACK - Linear Algebra PACKage,” "http://www.netlib.org/lapack/".

[2] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. J. Dongarra, “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, November 11-17, 2006, Tampa, FL, USA*, 2006, p. 113. [Online]. Available: https://doi.org/10.1145/1188455.1188573

[3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, “Accelerating Scientific Computations with Mixed Precision Algorithms,” *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

[4] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep Learning with Limited Numerical Precision,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303

[5] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–70, Aug 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4610935

[6] “NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS),” available at https://developer.nvidia.com/cublas.

[7] A. Abdelfattah, S. Tomov, and J. J. Dongarra, “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, May 20-24, 2019*, 2019, pp. 111–122.

[8] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, “Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC ’18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11. [Online]. Available: https://doi.org/10.1109/SC.2018.00050

[9] E. Carson and N. Higham, “A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems,” *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, 2017. [Online]. Available: https://doi.org/10.1137/17M1122918

[10] ——, “Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions,” *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, 2018. [Online]. Available: https://doi.org/10.1137/17M1140819

[11] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects,” *J. Phys.: Conf. Ser.*, vol. 180, no. 1, 2009.

[12] “MAGMA: Matrix Algebra on GPU and Multicore Architectures,” available at http://icl.cs.utk.edu/magma/.

[13] Y. Saad and M. H. Schultz, “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, Jul. 1986. [Online]. Available: https://doi.org/10.1137/0907058

[14] Y. Saad, “A Flexible Inner-outer Preconditioned GMRES Algorithm,” *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 461–469, Mar. 1993. [Online]. Available: http://dx.doi.org/10.1137/0914028

[15] V. Simoncini and D. Szyld, “Flexible Inner-Outer Krylov Subspace Methods,” *SIAM Journal on Numerical Analysis*, vol. 40, no. 6, pp. 2219–2239, 2002. [Online]. Available: https://doi.org/10.1137/S0036142902401074

[16] R. Nath, S. Tomov, and J. Dongarra, “An Improved Magma Gemm For Fermi Graphics Processing Units,” *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010. [Online]. Available: https://doi.org/10.1177/1094342010385729

[17] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, “Performance, design, and autotuning of batched GEMM for gpus,” in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[18] J. Lai and A. Seznec, “Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494986

[19] S. Gray, “A full walk through of the SGEMM implementation,” https://github.com/NervanaSystems/maxas/wiki/SGEMM, 2015.

[20] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, “Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2700–2712, 2018. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2842785

[21] N. Higham, S. Pranesh, and M. Zounon, “Squeezing a matrix into half precision, with an application to solving linear systems,” *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2536–A2551, 2019. [Online]. Available: https://doi.org/10.1137/18M1229511