Computer Physics Communications 180 (2009) 2526–2533

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

40th Anniversary Issue

Accelerating scientific computations with mixed precision algorithms ☆

Marc Baboulin [a], Alfredo Buttari [b], Jack Dongarra [c,d,e], Jakub Kurzak [c,*], Julie Langou [c], Julien Langou [f], Piotr Luszczek [g], Stanimire Tomov [c]

[a] Department of Mathematics, University of Coimbra, Coimbra, Portugal
[b] French National Institute for Research in Computer Science and Control, Lyon, France
[c] Department of Electrical Engineering and Computer Science, University Tennessee, Knoxville, TN, USA
[d] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[e] University of Manchester, Manchester, United Kingdom
[f] Department of Mathematical and Statistical Sciences, University of Colorado Denver, Denver, CO, USA
[g] MathWorks, Inc., Natick, MA, USA

ARTICLE INFO

Article history:
Received 2 September 2008
Received in revised form 9 November 2008
Accepted 10 November 2008
Available online 13 November 2008

PACS:
02.60.Dc

Keywords:
Numerical linear algebra
Mixed precision
Iterative refinement

ABSTRACT

On modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations. By using a combination of 32-bit and 64-bit floating point arithmetic, the performance of many dense and sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution. The approach presented here can apply not only to conventional processors but also to other technologies such as Field Programmable Gate Arrays (FPGA), Graphical Processing Units (GPU), and the STI Cell BE processor. Results on modern processor architectures and the STI Cell BE are presented.

Program summary

Program title: ITER-REF
Catalogue identifier: AECO_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AECO_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
No. of lines in distributed program, including test data, etc.: 7211
No. of bytes in distributed program, including test data, etc.: 41 862
Distribution format: tar.gz
Programming language: FORTRAN 77
Computer: desktop, server
Operating system: Unix/Linux
RAM: 512 Mbytes
Classification: 4.8
External routines: BLAS (optional)
Nature of problem: On modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations. By using a combination of 32-bit and 64-bit floating point arithmetic, the performance of many dense and sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution.
Solution method: Mixed precision algorithms stem from the observation that, in many cases, a single precision solution of a problem can be refined to the point where double precision accuracy is achieved. A common approach to the solution of linear systems, either dense or sparse, is to perform the LU factorization of the coefficient matrix using Gaussian elimination. First, the coefficient matrix A is factored into the product of a lower triangular matrix L and an upper triangular matrix U. Partial row pivoting is in general used to improve numerical stability resulting in a factorization $PA = LU$, where P is a permutation matrix. The solution for the system is achieved by first solving $Ly = Pb$ (forward substitution) and then solving $Ux = y$ (backward substitution). Due to round-off errors, the computed

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: kurzak@eecs.utk.edu (J. Kurzak).

0010-4655/$ – see front matter Published by Elsevier B.V.
doi:10.1016/j.cpc.2008.11.005

solution, x, carries a numerical error magnified by the condition number of the coefficient matrix A. In order to improve the computed solution, an iterative process can be applied, which produces a correction to the computed solution at each iteration, which then yields the method that is commonly known as the iterative refinement algorithm. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct to the working precision.
Running time: seconds/minutes

Published by Elsevier B.V.

1. Introduction

On modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations. There are two reasons for this. Firstly, 32-bit floating point arithmetic is usually twice as fast as 64-bit floating point arithmetic on most modern processors. Secondly the amount of bytes moved through the memory system is halved. In Table 1, we provide some hardware numbers that support these claims. On AMD Opteron 246, IBM PowerPC 970, and Intel Xeon 5100, the single precision peak is twice the double precision peak. On the STI Cell BE, the single precision peak is fourteen times the double precision peak. Not only single precision is faster than double precision on conventional processors but this is also the case on less mainstream technologies such as Field Programmable Gate Arrays (FPGA) and Graphical Processing Units (GPU). These speedup numbers tempt us and we would like to be able to benefit from it.

For several physics applications, results with 32-bit accuracy are not an option and one really needs 64-bit accuracy maintained throughout the computations. The obvious reason is for the application to give an accurate answer. Also, 64-bit accuracy enables most of the modern computational methods to be more stable; therefore, in critical conditions, one must use 64-bit accuracy to obtain an answer. In this manuscript, we present a methodology of how to perform the bulk of the operations in 32-bit arithmetic, then postprocess the 32-bit solution by refining it into a solution that is 64-bit accurate. We present this methodology in the context of solving a system of linear equations, be it sparse or dense, symmetric positive definite or nonsymmetric, using either direct or iterative methods. We believe that the approach outlined below is quite general and should be considered by application developers for their practical problems.

2. The idea behind mixed precision algorithms

Mixed precision algorithms stem from the observation that, in many cases, a single precision solution of a problem can be refined to the point where double precision accuracy is achieved. The refinement can be accomplished, for instance, by means of the Newton's algorithm [1] which computes the zero of a function $f(x)$ according to the iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1}$$

In general, we would compute a starting point and $f'(x)$ in single precision arithmetic and the refinement process will be computed in double precision arithmetic.

If the refinement process is cheaper than the initial computation of the solution then double precision accuracy can be achieved nearly at the same speed as the single precision accuracy. Sections 2.1 and 2.2 describe how this concept can be applied to solvers of linear systems based on direct and iterative methods, respectively.

Table 1
Hardware and software details of the systems used for performance experiments.

Architecture	Clock [GHz]	Peak SP/ Peak DP	Memory [MB]	BLAS	Compiler
AMD Opteron 246	2.0	2	2048	Goto-1.13	Intel-9.1
IBM PowerPC 970	2.5	2	2048	Goto-1.13	IBM-8.1
Intel Xeon 5100	3.0	2	4096	Goto-1.13	Intel-9.1
STI Cell BE	3.2	14	512	–	Cell SDK-1.1

1: $LU \leftarrow PA$ (ε_s)
2: solve $Ly = Pb$ (ε_s)
3: solve $Ux_0 = y$ (ε_s)
do $k = 1, 2, \ldots$
4: $r_k \leftarrow b - Ax_{k-1}$ (ε_d)
5: solve $Ly = Pr_k$ (ε_s)
6: solve $Uz_k = y$ (ε_s)
7: $x_k \leftarrow x_{k-1} + z_k$ (ε_d)
check convergence
done

Algorithm 1. Mixed precision, Iterative Refinement for Direct Solvers.

2.1. Direct methods

A common approach to the solution of linear systems, either dense or sparse, is to perform the LU factorization of the coefficient matrix using Gaussian elimination. First, the coefficient matrix A is factored into the product of a lower triangular matrix L and an upper triangular matrix U. Partial row pivoting is in general used to improve numerical stability resulting in a factorization $PA = LU$, where P is a permutation matrix. The solution for the system is achieved by first solving $Ly = Pb$ (*forward substitution*) and then solving $Ux = y$ (*backward substitution*). Due to round-off errors, the computed solution x carries a numerical error magnified by the condition number of the coefficient matrix A.

In order to improve the computed solution, we can apply an iterative process which produces a correction to the computed solution at each iteration, which then yields the method that is commonly known as the *iterative refinement* algorithm. As Demmel points out [2], the nonlinearity of the round-off errors makes the iterative refinement process equivalent to the Newton's method applied to the function $f(x) = b - Ax$. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct to the working precision. Iterative refinement in double/double precision is a fairly well understood concept and was analyzed by Wilkinson [3], Moler [4] and Stewart [5].

The algorithm can be modified to use a mixed precision approach. The factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients (see Algorithm 1). The most computationally expensive operation, the factorization of the coefficient matrix A, is performed using single precision arithmetic and takes advantage of its higher speed. The only operations that must be executed in double precision are the residual calculation and the update of the solution (they are denoted with an ε_d in Algorithm 1). We observe that the only operation with computational complexity of $\mathcal{O}(n^3)$ is handled in single precision, while

all operations performed in double precision are of at most $\mathcal{O}(n^2)$ complexity. The coefficient matrix A is converted to single precision for the LU factorization and the resulting factors are stored in single precision while the initial coefficient matrix A needs to be kept in memory. Therefore, one drawback of the following approach is that the it uses 50% more memory than the standard double precision algorithm.

The method in Algorithm 1 can offer significant improvements for the solution of a sparse linear system in many cases if:

1. single precision computation is significantly faster than double precision computation;
2. the iterative refinement procedure converges in a small number of steps;
3. the cost of each iteration is small compared to the cost of the system factorization. If the cost of each iteration is too high, then a low number of iterations will result in a performance loss with respect to the full double precision solver. In the sparse case, for a fixed matrix size, both the cost of the system factorization and the cost of the iterative refinement step may substantially vary depending on the number of nonzeroes and the matrix sparsity structure. In the dense case, results are more predictable.

Note that the choice of the stopping criterion in the iterative refinement process is critical. Formulas for the error computed at each step of Algorithm 1 can be obtained for instance in [6,7].

2.2. Iterative methods

Direct methods are usually a very robust tool for the solution of sparse linear systems. However, they suffer from fill-in which results in high memory requirements, long execution time, and nonoptimal scalability in parallel environments. To overcome these limitations, various pivot reordering techniques are commonly applied to minimize the amount of generated fill-in and to enable better exploitation of parallelism. Still, there are cases where direct methods pose too high of a memory requirement or deliver poor performance. A valid alternative are iterative methods even though they are less robust and have a less predictable behavior. Iterative methods do not require more memory than what is needed for the original coefficient matrix. Further more, time to solution can be better than that of direct methods if convergence is achieved in relatively few iterations [8,9].

In the context of iterative methods, the refinement method outlined in Algorithm 1 can be represented as

$$x_{i+1} = x_i + M(b - Ax_i), \tag{2}$$

where M is $(LU)^{-1}P$. Iterative methods of this form (i.e. where M does not depend on the iteration number i) are also known as *stationary*. Matrix M can be as simple as a scalar value (the method then becomes a modified Richardson iteration) or as complex as $(LU)^{-1}P$. In either case, M is called a *preconditioner*. The preconditioner should approximate A^{-1}, and the quality of the approximation determines the convergence properties of (2). In general, a preconditioner is intended to improve the robustness and the efficiency of iterative methods. Note that (2) can also be interpreted as a Richardson method's iteration in solving $MAx = Mb$ which is called *left* preconditioning. An alternative is to use *right* preconditioning, whereby the original problem $Ax = b$ is transformed into a problem of solving

$$AMu = b, \qquad x = Mu$$

iteratively. Later on, we will use the right preconditioning for mixed precision iterative methods.

1: **for** $i = 0, 1, \ldots$ **do**
2: $r = b - Ax_i$ (ε_d)
3: $\beta = h_{1,0} = \|r\|_2$ (ε_d)
4: check convergence and exit if done
5: **for** $k = 1, \ldots, m_{\text{out}}$ **do**
6: $v_k = r/h_{k,k-1}$ (ε_d)
7: Perform one cycle of $\text{GMRES}_{\text{SP}}(m_{\text{in}})$ in order to (approximately) solve $Az_k = v_k$ (initial guess $z_k = 0$) (ε_s)
8: $r = Az_k$ (ε_d)
9: **for** $j = 1, \ldots, k$ **do**
10: $h_{j,k} = r^T v_j$ (ε_d)
11: $r = r - h_{j,k} v_j$ (ε_d)
12: **end for**
13: $h_{k+1,k} = \|r\|_2$ (ε_d)
14: **end for**
15: Define $Z_k = [z_1, \ldots, z_k]$, $H_k = \{h_{i,j}\}_{1 \leqslant i \leqslant k+1, 1 \leqslant j \leqslant k}$ (ε_d)
16: Find y_k, the vector of size k, that minimizes $\|\beta e_1 - H_k y_k\|_2$ (ε_d)
17: $x_{i+1} = x_i + Z_k y_k$ (ε_d)
18: **end for**

Algorithm 2. Mixed precision, inner–outer FGMRES(m_{out})-GMRES_{SP}(m_{in}).

M needs to be easy to compute, apply, and store to guarantee the overall efficiency. Note that these requirements were addressed in the mixed precision direct methods above by replacing M (coming from LU factorization of A followed by matrix inversion), with its single precision representation so that arithmetic operations can be performed more efficiently on it. Here however, we go two steps further. We replace not only M by an inner loop which is an incomplete iterative solver working in single precision arithmetic [10]. Also, the outer loop is replaced by a more sophisticated iterative method, e.g., based on Krylov subspace.

Note that replacing M 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 [11–17]. The general appeal of these methods is that the computational speedup hinders inner solver's ability to use an approximation of the original matrix A that is fast to apply. In our case, we use single precision arithmetic matrix–vector product as a fast approximation of the double precision operator in the inner iterative solver. Moreover, even if no faster matrix–vector product is available, speedup can often be observed due to improved convergence (e.g., see [15], where Simoncini and Szyld explain the possible benefits of FGMRES-GMRES over restarted GMRES).

To illustrate the above concepts, we demonstrate an inner–outer nonsymmetric iterative solver in mixed precision. The solver is based on the restarted Generalized Minimal RESidual (GMRES) method. In particular, consider Algorithm 2, where the outer loop uses the flexible GMRES (FGMRES [9,14]) and the inner loop uses the GMRES in single precision arithmetic (denoted by GMRES_{SP}). FGMRES, being a minor modification of the standard GMRES, is meant to accommodate non-constant preconditioners. Note that in our case, this non-constant preconditioner is GMRES_{SP}. The resulting method is denoted by FGMRES(m_{out})-GMRES_{SP}(m_{in}) where m_{in} is the restart for the inner loop and m_{out} for the outer FGMRES. Algorithm 2 checks for convergence every m_{out} outer iterations. Our actual implementation checks for convergence at every inner iteration, this can be done with simple tricks at almost no computational cost.

The potential benefits of FGMRES compared to GMRES are becoming better understood [15]. Numerical experiments confirm improvements in speed, robustness, and sometimes memory requirements for these methods. For example, we show a maximum speedup of close to 15 on the selected test problems. The memory requirements for the method are the matrix A in CRS format, the nonzero matrix coefficients in single precision, $2m_{\text{out}}$ number of vectors in double precision, and m_{in} number of vectors in single precision.

Table 2
Test matrices for sparse mixed precision, iterative refinement solution methods.

n	Matrix	Size	Nonzeroes	Symm.	Pos. def.	C. num.
1	SiO	33401	1317655	yes	no	$O(10^3)$
2	Lin	25600	1766400	yes	no	$O(10^5)$
3	c-71	76638	859554	yes	no	$O(10)$
4	cage-11	39082	559722	no	no	$O(1)$
5	raefsky3	21200	1488768	no	no	$O(10)$
6	poisson3Db	85623	2374949	no	no	$O(10^3)$

The Generalized Conjugate Residuals (GCR) method [17,18] is a possible replacement for FGMRES as the outer iterative solver. Whether to choose GCR or FGMRES is not yet well understood.

As in the dense case, the choice of the stopping criterion in the iterative refinement process is critical. In the sparse case, formulas for the errors can be computed following the work of Arioli et al. [19].

3. Performance results

The experimental results reported in this section were measured on the systems described in Table 1. At this moment no software libraries are available to perform sparse computations on the STI Cell BE architecture. For this reason, only mixed precision iterative refinement solvers for dense linear systems are presented for this architecture.

To measure the performance of sparse mixed precision solvers based on both direct and iterative methods, the matrices described in Table 2 were used.

Based on backward stability analysis, the solution x can be considered as accurate as the double precision one when

$$\|b - Ax\|_2 \leqslant \|x\|_2 \cdot \|A\|_2 \cdot \varepsilon \cdot \sqrt{n},$$

where $\|\cdot\|_2$ is the spectral norm. However, for the following experiments, a full double precision solution is computed first and then the mixed precision iterative refinement is stopped when the computed solution is as accurate as the full double precision one.

3.1. Direct methods

3.1.1. Dense matrices

Mixed precision iterative refinement solvers were developed for both symmetric and nonsymmetric dense linear systems by means of the methods and subroutines provided by the BLAS [20–24] and LAPACK [25] software packages. For the nonsymmetric case, step 1 in Algorithm 1 is implemented by means of the `SGETRF` subroutine, steps 2, 3 and 5, 6 with the `SGETRS` subroutine, step 4 with the `DGEMM` subroutine and step 7 with the `DAXPY` subroutine. For the symmetric case the `SGETRF`, `SGETRS` and `DGEMM` subroutines were replaced by the `SPOTRF`, `SPOTRS` and `DSYMM` subroutines, respectively. Further details on these implementations can be found in [26,27].

As already mentioned, iterative refinement solvers require 1.5 times as much memory as a regular double precision solver. It is because the mixed precision iterative refinement solvers need to store at the same time both the single precision and the double precision versions of the coefficient matrix. It is true for dense as well as sparse matrices.

Table 3 shows the speedup of the mixed precision, iterative refinement solvers for dense matrices with respect to full, double precision solvers. These results show that the mixed precision iterative refinement method can run very close to the speed of the full single precision solver while delivering the same accuracy as the full double precision one. On the AMD Opteron, Intel Woodcrest and IBM PowerPC architectures, the mixed precision, iterative solver can provide a speedup of up to 1.8 for the nonsymmetric solver and 1.5 for the symmetric one for large enough problem sizes. For small problem sizes the cost of even a few iterative refinement iterations is high compared to the cost of the factorization and thus the mixed precision iterative solver is less efficient than the double precision one.

Table 3
Performance improvements for direct dense methods when going from a full double precision solve (reference time) to a mixed precision solve.

	Nonsymmetric	Symmetric
AMD Opteron 246	1.82	1.54
IBM PowerPC 970	1.56	1.35
Intel Xeon 5100	1.56	1.43
STI Cell BE	8.62	10.64

Parallel implementations of Algorithm 1 for the symmetric and nonsymmetric cases have been produced in order to exploit the full computational power of the Cell processor (see also Fig. 1). Due to the large difference between the speed of single precision and double precision floating point units,[1] the mixed precision solver performs up to 7 times faster than the double precision peak in the nonsymmetric case and 11 times faster for the symmetric positive definite case. Implementation details for this case can be found in [28,29].

3.1.2. Sparse matrices

Most sparse direct methods for solving linear systems of equations are variants of either multifrontal [30] or supernodal [31] factorization approaches. Here, we focus only on multifrontal methods. For results on supernodal solvers see [32]. There are a number of freely available packages that implement multifrontal methods. We have chosen for our tests a software package called MUMPS [33–35]. The main reason for selecting this software is that it is implemented in both single and double precision, which is not the case for other freely available multifrontal solvers such as UMFPACK [36–38].

Using the MUMPS package for solving systems of linear equations comprises of three separate steps:

1. System Analysis: in this phase the system sparsity structure is analyzed in order to estimate the element fill-in, which provides an estimate of the memory that will be allocated in the following steps. Also, pivoting is performed based on the structure of $A + A^T$, ignoring numerical values. Only integer operations are performed at this step.
2. Matrix Factorization: in this phase the $PA = LU$ factorization is performed. This is the computationally most expensive step of the system solution.
3. System Solution: the system is solved in two steps: $Ly = Pb$ and $Ux = y$.

The Analysis and Factorization phases correspond to step 1 in Algorithm 1 while the solution phase correspond to steps 2, 3 and 5, 6.

The speedup of the mixed precision, iterative refinement approach over the double precision one for sparse direct methods is shown in Table 4, and Fig. 2. The figure reports the performance ratio between the full single precision and full double precision solvers (light colored bars) and the mixed precision and full-double precision solvers (dark colored bars) for six matrices from real world applications. The number on top of each bar shows how

[1] As indicated in Table 1, the peak for single precision operations is 14 times more than the peak for double precision operations on the STI Cell BE.

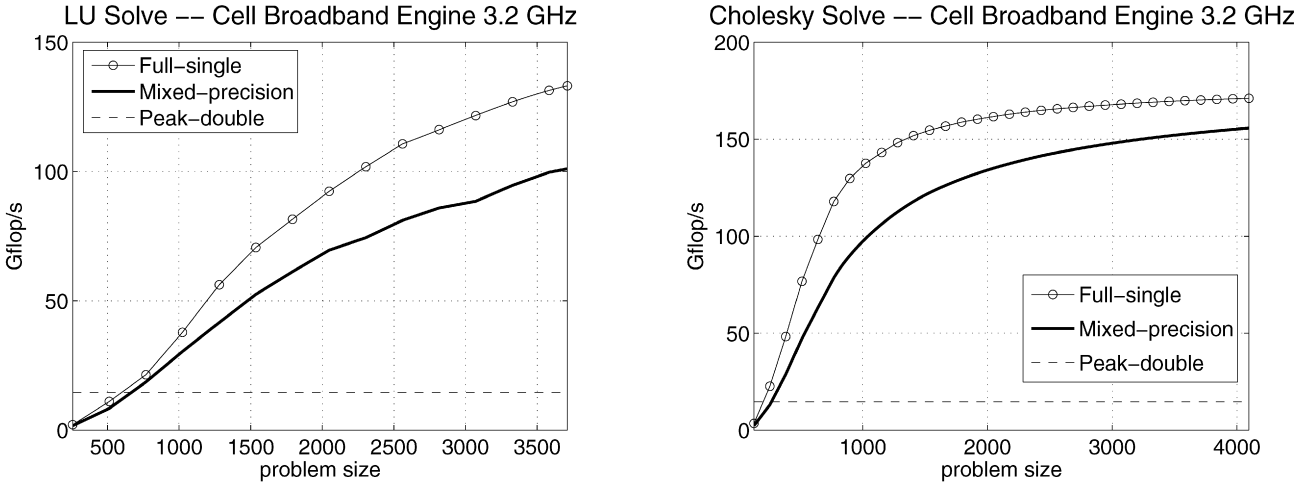

Fig. 1. Mixed precision, iterative refinement method for the solution of dense linear systems on the STI Cell BE processor.

Table 4
Performance improvements for direct sparse methods when going from a full double precision solve (reference time) to a mixed precision solve.

	Matrix number					
	1	2	3	4	5	6
AMD Opteron 246	1.827	1.783	1.580	1.858	1.846	1.611
IBM PowerPC 970	1.393	1.321	1.217	1.859	1.801	1.463
Intel Xeon 5100	1.799	1.630	1.554	1.768	1.728	1.524

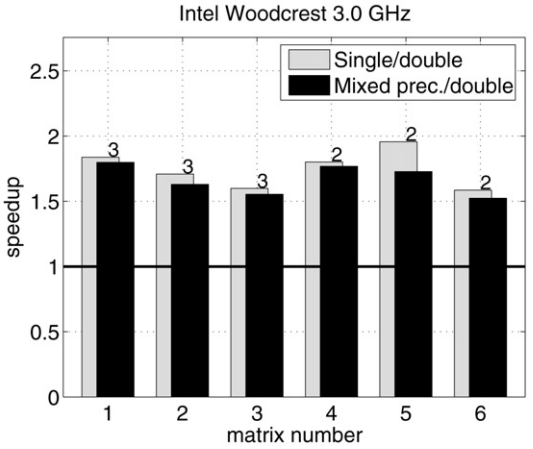

Fig. 2. Mixed precision, iterative refinement with the MUMPS direct solver on an Intel Woodcrest 3.0 GHz system.

many iterations are performed by the mixed precision, iterative method to achieve double precision accuracy.

3.2. Iterative methods

Similar to the case of sparse direct solvers, we demonstrate the numerical performance of Algorithm 2 on the architectures from Table 1 and on the matrices from Table 2.

Fig. 3 (left) shows the performance ratio of the mixed precision inner–outer FGMRES-GMRES$_{SP}$ *vs.* the full double precision inner–outer FGMRES-GMRES$_{DP}$. In other words, we compare two inner–outer algorithms that are virtually the same. The only difference is that their inner loop's incomplete solvers are performed in correspondingly single and double precision arithmetic.

Fig. 3 (right) shows the performance ratio of the mixed precision inner–outer FGMRES-GMRES$_{SP}$ *vs.* double precision GMRES. This is an experiment that shows that inner–outer type iterative methods may be very competitive compared to their original counterparts. For example, we observe a speedup for matrix #2 of up to 6 which is mostly due to an improved convergence of the inner–outer GMRES *vs.* standard GMRES. In particular, about 3.5 of the 5.5-fold speedup for matrix # 2 on the IBM PowerPC architecture is due to improved convergence, and the rest 1.57 speedup is due to single *vs* double precision arithmetic. The restart values used for this computation are given in Table 5. The restart values m_{in} and m_{out} were manually tuned, m was taken as $2m_{\text{out}} + m_{\text{in}}$ in order to use the same amount of memory space for the two different methods, or additionally increased when needed to improve the reference GMRES solution times.

4. Numerical remarks

Following the work of Skeel [39], Higham [40] gives error bounds for the single and double precision, iterative refinement algorithm when the entire algorithm is implemented with the same precision (single or double, respectively). Higham also gives error bounds in single precision arithmetic, with refinement performed in double precision arithmetic [40]. The error analysis in double precision, for our mixed precision algorithm (Algorithm 1), is given by Langou et al. [27]. Arioli and Duff [41] gives the error analysis for a mixed precision algorithm based on a double precision FGMRES preconditioned by a single precision LU factorization. These errors bounds explain that mixed precision iterative refinement will work as long as the condition number of the coefficient matrix is smaller than the inverse of the lower precision used. For practical reasons, we need to resort to the standard double precision solver in the cases when the condition number of the coefficient matrix is larger than the inverse of the lower precision used.

In Fig. 4, we show the number of iterations needed for our mixed precision method to converge to better accuracy than the one of the associated double precision solve. The number of iterations is shown as a function of the condition number of the coefficient matrix (κ) in the context of direct dense nonsymmetric solve. For each condition number, we have taken 200 random matrices of size 200-by-200 with a prescribed condition number and we report the mean number of iterations until convergence. The maximum number of iterations allowed was set to 30 so that 30 means failure to converge (as opposed to convergence in 30 iterations). Datta [42] has conjectured that the number of iterations necessary for convergence was given by

$$\left\lceil \frac{\ln(\varepsilon_d)}{\ln(\varepsilon_d) + \ln(\kappa)} \right\rceil.$$

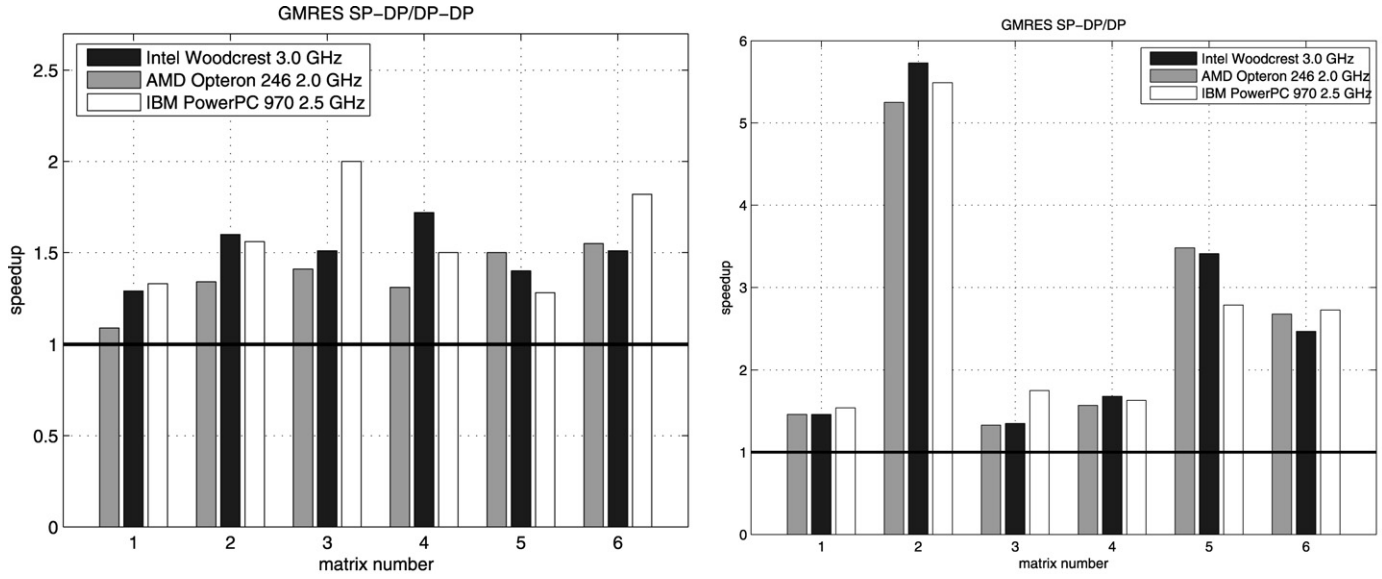

Fig. 3. Mixed precision iterative refinement with FGMRES-GMRES$_{SP}$ from Algorithm 2 *vs.* FGMRES-GMRES$_{DP}$ (left) and *vs.* full double precision GMRES (right).

Table 5
Restart values for the GMRES-based iterative solvers.

Matrix n	m_{in}	m_{out}	m
1	30	20	150
2	20	10	40
3	100	9	300
4	10	4	18
5	20	20	300
6	20	10	50

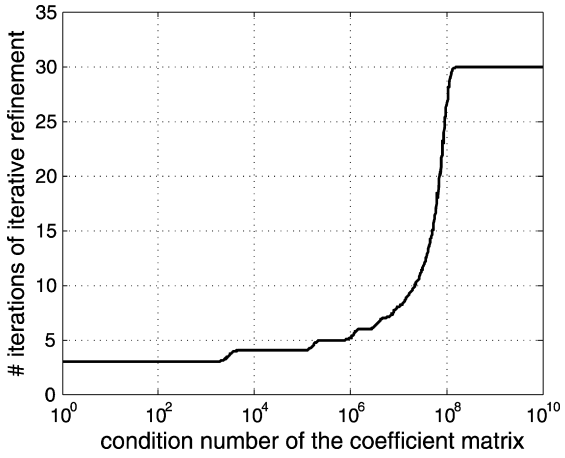

Fig. 4. Number of iterations needed for our mixed precision method to converge to an accuracy better than the one of the associated double precision solve as a function of the condition number of the coefficient matrix in the context of direct dense nonsymmetric solves.

We can generalize this formula in the context of our mixed precision approach

$$\left\lceil \frac{\ln(\varepsilon_d)}{\ln(\varepsilon_s)+\ln(\kappa)} \right\rceil.$$

When $\kappa\varepsilon_s$ is above 1, then the formula is not valid anymore. This is characterized in practice by an infinite number of iterations, i.e. lack of convergence of the method.

Table 6
Iterative refinement in quadruple precision on a Intel Xeon 3.2 GHz.

n	QGESV time (s)	QDGESV time (s)	Speedup
100	0.29	0.03	9.5
200	2.27	0.10	20.9
300	7.61	0.24	30.5
400	17.81	0.44	40.4
500	34.71	0.69	49.7
600	60.11	1.01	59.0
700	94.95	1.38	68.7
800	141.75	1.83	77.3
900	201.81	2.33	86.3
1000	276.94	2.92	94.8

Table 7
Time for the various Kernels in the Quadruple Accuracy Versions for $n = 900$.

Driver name	Time (s)	Kernel name	Time (s)
QGESV	201.81	QGETRF	201.1293
		QGETRS	0.6845
QDGESV	2.33	DGETRF	0.3200
		DGETRS	0.0127
		DLANGE	0.0042
		DGECON	0.0363
		QGEMV	1.5526
		ITERREF	1.9258

5. Extension to quadruple precision

As an extension to this study, we present in this section results for iterative refinement in quadruple precision on an Intel Xeon 3.2 GHz. The iterative refinement code computes a condition number estimate for input matrices having random entries drawn from a uniform distribution. For quadruple precision arithmetic, we use the reference BLAS compiled with the Intel Fortran compiler *ifort* (with -O3 optimization flag on) since we do not have an optimized BLAS in quadruple precision. The version of the compiler is 8.1. Results are presented in Table 6. The obtained accuracy is between 10 and 32 for QGETRF and QDGETRF as expected. No more than 3 steps of iterative refinement are needed. The speedup is between 10 for a matrix of size 100 to close to 100 for a matrix of size 1000. In Table 7, we give the time for the different kernels used in QGESV and QDGESV. Interestingly enough the time for QDGESV

is dominated by QGEMV and not DGETRF! Recent research using related idea can be found in [43].

6. Extension to other algorithms

Mixed precision algorithms can easily provide substantial speed-up for very little code effort by mainly taking into account existing hardware properties.

We have shown how to derive mixed precision version of variety of algorithms for solving general linear systems of equations. Mixed precision iterative refinement technique has also be used in the context of symmetric positive definite systems [28] using a Cholesky factorization. In the context of overdetermined least squares problems, the iterative refinement technique can be applied to the augmented system (where both the solution and the residual are refined, as described in [44]), to the QR factorization, to the semi-normal equations or to the normal equations [45]. Iterative refinement can also be applied for eigenvalue computation [46] and for singular value computation [47].

We hope this manuscript will encourage scientists to extend this approach to their own applications that do not necessarily originate from linear algebra.

7. Conclusions

The main conclusion of the research presented in this paper is that in mixed-precision algorithms can provide performance benefits in solving linear systems of equations using dense direct methods, sparse direct methods and sparse iterative methods. If the problem being solved is well conditioned, most of the computational work can be performed in single (32-bit) precision and full (64-bit) precision can be recovered by a small amount of extra work.

8. Description of the individual software components

mixed-precision.c Provides a simple example of invoking the DSGESV routine to solve a dense linear system of equations using mixed precision approach (single/double). The code performs memory allocation, data initialization, invokes the solver and, after calculations complete, reports accuracy of the solution, number of refinement steps and performance in Gflop/s. The code demonstrates the use of legacy FORTRAN BLAS interface as well as CBLAS interface.

dsgesv.f Implements the main routine solving dense system of linear equations using the technique of iterative refinement to achieve the speed of single precision arithmetics, while providing double precision results. Relies on BLAS and a number of FORTRAN routines to provide the required building blocks for the computation.

***.f** The remaining FORTRAN routines in the top-level directory implement all the necessary components of the DSGESV routine and rely on a set of BLAS for the basic linear algebra operations.

BLAS/ Contains reference FORTRAN implementation of all the BLAS routines required by the algorithm, which can be used if optimized BLAS library is not available. In such case, however, only a small fraction of achievable performance will be delivered.

Makefile.reference_BLAS Contains makefile to compile a standalone version of the code, which does not require the BLAS library. The routines from the ../BLAS subdirectory are compiled and linked in instead. Invokes GFORTRAN and GCC to compile the source code.

Makefile.optimized_BLAS Contains an example makefile to compile the code on an Intel system using the BLAS provided by the Math Kernel Library (MKL). Invokes Intel ICC and IFORT to compile the source code.

9. Installation instructions

The code can be compiled on any system where GCC and GFORTRAN compilers are available by invoking the command:

```
> make -f Makefile.reference_BLAS
```

However, only mediocre performance will be observed. It is strongly suggested that the code is linked with an optimized implementation of the BLAS library, such as MKL or ACML. In order to do so the user needs to modify Makefile.optimized_BLAS to reflect the configuration of the system and invoke it as follows:

```
> make -f Makefile.optimized_BLAS
```

10. Test run description

The compilation process produces the executable “mixed_precision”, which can be invoked to test the code. The program takes problem size as its sole argument. To test the accuracy and performance of the code for a problem defined by a 1000x1000 matrix, one can type at command prompt:

```
> mixed_precision 1000
```

Different results will be obtained depending on the BLAS implementation used, the compiler optimizations used and the values of the random data initializations. As long as the number of refinement steps is less than 5 and the residual norm is $O(1.0e{-}12)$ the installation can be deemed to be working correctly.

References

[1] T.J. Ypma, SIAM Review 37 (1995) 531.
[2] J.W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997.
[3] J.H. Wilkinson, Rounding Errors in Algebraic Processes, Prentice-Hall, 1963.
[4] C.B. Moler, J. ACM 14 (1967) 316.
[5] G.W. Stewart, Introduction to Matrix Computations, Academic Press, 1973.
[6] J.W. Demmel, et al., ACM Trans. Math. Software 32 (2006) 325.
[7] W. Oettli, W. Prager, Numer. Math. 6 (1964) 405.
[8] R. Barrett, et al., Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, Society for Industrial and Applied Mathematics, Philadelphia, 1994. Also available as postscript file at http://www.netlib.org/templates/Templates.html.
[9] Y. Saad, Iterative Methods for Sparse Linear Systems, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[10] K. Turner, H.F. Walker, SIAM J. Sci. Stat. Comput. 13 (1992) 815.
[11] O. Axelsson, P.S. Vassilevski, SIAM J. Matrix Anal. Appl. 12 (1991) 625.
[12] G.H. Golub, Q. Ye, SIAM J. Scientific Comput. 21 (2000) 1305.
[13] Y. Notay, SIAM J. Scientific Comput. 22 (2000) 1444.
[14] Y. Saad, A flexible inner–outer preconditioned GMRES algorithm, Technical Report 91-279, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota, 1991.
[15] V. Simoncini, D.B. Szyld, SIAM J. Numer. Anal. 40 (2003) 2219.
[16] J. van den Eshof, G.L.G. Sleijpen, M.B. van Gijzen, J. Comput. Appl. Math. 177 (2005) 347.
[17] C. Vuik, J. Comput. Appl. Math. 61 (1995) 189.
[18] H.A. van der Vorst, C. Vuik, Numer. Linear Algebra Appl. 1 (1994) 369.
[19] M. Arioli, J.W. Demmel, I.S. Duff, SIAM J. Matrix Anal. Appl. 10 (1989) 165.
[20] J.J. Dongarra, J.D. Croz, I.S. Duff, S. Hammarling, ACM Trans. Math. Software 16 (1990) 18.
[21] J.J. Dongarra, J.D. Croz, I.S. Duff, S. Hammarling, ACM Trans. Math. Software 16 (1990) 1.
[22] J.J. Dongarra, J.D. Croz, S. Hammarling, R.J. Hanson, ACM Trans. Math. Software 14 (1988) 18.
[23] J.J. Dongarra, J.D. Croz, S. Hammarling, R.J. Hanson, ACM Trans. Math. Software 14 (1988) 1.
[24] C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, ACM Trans. Math. Software 5 (1979) 308.
[25] E. Anderson, et al., LAPACK Users’ Guide, 3rd edition, SIAM, Philadelphia, 1999.
[26] A. Buttari, et al., Int. J. High Performance Comput. Appl. 21 (2007) 457.
[27] J. Langou, et al., Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006.
[28] J. Kurzak, A. Buttari, J.J. Dongarra, IEEE Trans. Parallel Distrib. Systems 19 (2008) 1.

[29] J. Kurzak, J.J. Dongarra, Concurrency Computat.: Pract. Exper. 19 (2007) 1371.
[30] I.S. Duff, J.K. Reid, ACM Trans. Math. Software 9 (1983) 302.
[31] C. Ashcraft, R. Grimes, J. Lewis, B.W. Peyton, H. Simon, Intern. J. Supercomput. Appl. 1 (1987) 10.
[32] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, S. Tomov, ACM Trans. Math. Software 34 (2008) 17 (Article 17, 22 pages).
[33] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, Comput. Methods Appl. Mech. Eng. 184 (2000) 501.
[34] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, J. Koster, SIAM J. Matrix Anal. Appl. 23 (2001) 15.
[35] P.R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Parallel Comput. 32 (2006) 136.
[36] T.A. Davis, ACM Trans. Math. Software 25 (1999) 1.
[37] T.A. Davis, ACM Trans. Math. Software 30 (2004) 196.
[38] T.A. Davis, I.S. Duff, SIAM J. Matrix Anal. Appl. 18 (1997) 140.
[39] R.D. Skeel, Math. Comput. 35 (1980) 817.
[40] N.J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd edition, SIAM, 2002.
[41] M. Arioli, I.S. Duff, Using FGMRES to obtain backward stability in mixed precision, in: Technical Report RAL-TR-2008-006, Rutherford Appleton Laboratory, 2008.
[42] B.D. Datta, Numerical Linear Algebra and Applications, Brooks Cole Publishing Company, 1995.
[43] K.O. Geddes, W.W. Zheng, Exploiting fast hardware floating point in high precision computation, in: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation. Philadelphia, PA, USA, 2003, pp. 111–118.
[44] J.W. Demmel, Y. Hida, X.S. Li, E.J. Riedy, Extra-precise iterative refinement for overdetermined least squares problems, Technical Report EECS-2007-77, UC Berkeley, 2007, Also LAPACK Working Note 188.
[45] Å. Björck, Numerical Methods for Least Squares Problems, SIAM, 1996.
[46] J.J. Dongarra, C.B. Moler, J.H. Wilkinson, SIAM J. Numer. Anal. 20 (1983) 23.
[47] J.J. Dongarra, SIAM J. Scientific Statist. Comput. 4 (1983) 712.