LAPACK Working Note 175: Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems)

Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak,
Alfredo Buttari and Jack Dongarra

University of Tennessee, Knoxville TN 37996, USA,
{julie,langou,luszczek,kurzak,buttari,dongarra}@cs.utk.edu

July 31, 2006

Abstract:

Recent versions of microprocessors exhibit performance characteristics for 32 bit floating point arithmetic (single precision) that is substantially higher than 64 bit floating point arithmetic (double precision). Examples include the Intel's Pentium IV and M processors, AMD's Opteron architectures and the IBM's Cell Broad Engine processor. When working in single precision, floating point operations can be performed up to two times faster on the Pentium and up to ten times faster on the Cell over double precision. The performance enhancements in these architectures are derived by accessing extensions to the basic architecture, such as SSE2 in the case of the Pentium and the vector functions on the IBM Cell. The motivation for this paper is to exploit single precision operations whenever possible and resort to double precision at critical stages while attempting to provide the full double precision results. The results described here are fairly general and can be applied to various problems in linear algebra such as solving large sparse systems, using direct or iterative methods and some eigenvalue problems. There are limitations to the success of this process, such as when the conditioning of the problem exceeds the reciprocal of the accuracy of the single precision computations. In that case the double precision algorithm should be used.

1. Introduction

The motivation behind this work is the observation that a number of recent processor architectures exhibit single precision performance that is significantly higher than for double precision arithmetic. An example of this include the IBM Cell multiprocessor which has a theoretical peak of 204.8 GFlop/s in single precision (32 bit floating point arithmetic) and a peak of only 14.6 GFlop/s in double precision (64 bit floating point arithmetic) for a 3.2 GHz processor using the SPU's only [7]. Even the Intel x87 processor with the use of the Streaming SIMD Extensions (SSE) unit on the Pentium III does 4 flops/cycle for single precision, and SSE2 does 2 flops/cycle for double. Therefore, for any processor with SSE and SSE2 (e.g. Pentium IV), the theoretical peak of single is twice that of double, and on a chip with SSE and without SSE2 (e.g. some Pentium III), the theoretical peak of single is four times that of double. AMD Opteron processors share the same relation between SSE and SSE2. Appendix 1 contains additional information on the extensions to the IA-32 instruction set.

Another advantage of computing in single versus double precision is that data movement is cut in half. This helps performance by reducing memory traffic across the bus and enabling larger blocks of user's data to fit into cache. In parallel computations, the total volume of communication is reduced by half and the number of initiated communication is reduced as well (if block sizes are doubled). The effect is that the communication behaves as if the bandwidth is multiplied by two and latency halved by two.

The use of extensions to the ISA of x86-x87 has been put into practice in a number of implementations of the BLAS. This provides a speed improvement of a factor of two in single precision compared to double precision for basic operations such as matrix multiply. Some experimental comparisons of SGEMM versus DGEMM on various architectures are given in Table 2.

The motivation for this paper is to exploit single precision operations whenever possible and resort to double precision at critical stages while attempting to provide the full double precision results.

2. Iterative refinement for Systems of Dense Linear Equations

Iterative refinement for the solution of linear equations is a practical technique that has been in use for many years. Suppose $Ax = b$ has been solved via Gaussian Elimination with partial pivoting and we have the standard factorization $PA = LU$, where L is a lower triangular matrix, U an upper triangular matrix, and P a permutation matrix used for pivoting. The iterative refinement process is:

$r = b - Ax$
Solve $Ly = Pr$
Solve $Uz = y$
$x_+ = x + z$.

As Demmel **[13, pp.60]** points out the iterative refinement process is equivalent to Newton's method applied to $f(x) = b - Ax$. If all the computations were done exactly, we would be done in one step.

We are planning to use a mixed precision iterative refinement process. That is the factorization, $PA = LU$, and the triangular solves $Ly = Pr$ and $Uz = y$ will be computed using single precision and the residual (using the original data) and updating of the solution will be computed using double precision. Most of the floating point operations, the factorization of the matrix A and the forward and back substitutions will be performed in single precision. Only the residual computation and solution update are performed in double precision. The mixed precision approach was analyzed by Wilkinson **[9]** and Moler **[1]**; they showed that, provided A is not too ill–conditioned, it produces a computed solution correct to the working precision, in this case single precision. As pointed out in Demmel **[6]**, the behavior of the method depends strongly on the accuracy with which the residual is computed. The input variables A and b are stored in double precision $\square_d$. The basic solution method is used to solve $Ax = b$ and $Az = r$ in single precision $\square_s$. The residual is computed in double precision $\square_d$ and the solution updated in double precision $\square_d$.

Iterative refinement is a fairly well understood algorithm. For example, Higham **[14]** gives error bounds in single precision (resp. double) for fixed precision iterative refinement performed in single precision arithmetic (resp. double) and Higham **[14]** also gives error bounds in single precision arithmetic for mixed precision iterative refinement (when the refinement is performed in double precision arithmetic). However we did not find in the literature any error bound in double precision arithmetic when mixed precision iterative refinement (single/double) is performed.

Stewart **[8]** provides an error analysis of iterative refinement. This analysis can be adapted to our context and the details are provided **Appendix 2**. The bottom line is that we can achieve the same accuracy using this approach as if we have computed the solution fully in 64 bit floating point precision, provided that the matrix is not too badly conditioned.

The impact of this result is that if the factorization, and the forward and back substitutions are performed in single precision and the residual and update for the solution are performed in double precision, then the iterative refinement process will, as long as the matrix is not too badly conditioned, produce the same accuracy in the computed solution as if the double precision computation has been performed on the factorization, and the forward and back substitutions. The disadvantage is that we must retain a copy of the original matrix to compute the residual. So, the space cost is increased by 1.5 and the potential saving of computational time is a factor of 2 (assuming single precision computations are twice as fast as double precision computations).

It is important to note that we are computing a correction to the solution, z, and then use that correction to update the solution, x_+. A key point is that, while the correction is computed in single precision, the update is computed using double precision.

In **Appendix B**, we recall some results about iterative refinement (we have slightly adapted standard results to our framework). Given the working machine precision ε_s and ε_d and assuming standard floating-point arithmetic (see **[14]**), and assuming that the matrix A is not too-ill conditioned with respect to the single precision ($\square_s$) arithmetic used, namely we will assume that $\psi_K(n)\kappa(A)\varepsilon_s < 1$, the following inequality holds

$$\frac{\|b - Ax_{k+1}\|}{\|A\| \cdot \|x_{k+1}\|} \leq \alpha_B \cdot \frac{\|b - Ax_k\|}{\|A\| \cdot \|x_k\|} + \beta_B,$$

where α_B and β_B are of the form

$\alpha_B = \psi_B(n)\kappa(A)\varepsilon_s$ and $\beta_B = \rho_B(n)\varepsilon_d$.

$\psi_B(n)$, $\rho_B(n)$ and $\psi_K(n)$ are small function of n that are explicitly given in **Appendix B**.

α_B is the rate of convergence and depends on the condition number of the matrix A ($\kappa(A)$) and the single precision used (ε_s). β_B is the limiting accuracy of the method and depends on the double precision used (ε_d).

At convergence we have

$$\lim_{k \to +\infty} \frac{\|b - Ax_k\|}{\|A\| \cdot \|x_k\|} = \beta_B (1 - \alpha_B)^{-1} = \frac{\rho_B(n)}{1 - \psi_B(n)\kappa(A)\varepsilon_s} \varepsilon_d .$$

Normwise backward double precision accuracy is obtained.

3. Practical Implementation

A practical implementation of the method can be found at http://icl.cs.utk.edu/iter-ref. The code is in Fortran 77 and uses LAPACK and the BLAS routines. The iterative refinement is stopped either if the number of iteration exceeds ITERMAX (=30 by default), or if

$$\|b - Ax\|_2 \leq \|x\|_2 \cdot \|A\|_{fro} \cdot \varepsilon \cdot \sqrt{n} .$$

($\|\,.\,\|_{fro}$ corresponds to the Frobenius norm.)
If the iterative refinement procedure does not succeed, that is fails to converge, then the procedure will automatically switch to solving the system of linear equations using double precision arithmetic.

4. Some Numerical Experiments

4.1 Testing

The driver routine, DGESIRSV, given in **Appendix 3** successfully passed the LAPACK testing for DGESV. This consists of 147 matrices and 3 numerical tests are checked. Since those matrices are most of the time pathological, it is a good exercise for the routine to check if it is able to switch to a double precision solve when necessary. Out of those 147 matrices, 15 matrices have a condition number close to 10^{15}, so the iterative refinement does not converge. 18 matrices produce overflow when converted from double to single, and 18 matrices fail in the single LU factorization. For all those matrices DGESIRSV switches, as expected, to DGESV. The 96 remaining matrices converge fine with iterative refinement.

4.2 LAPACK Kernels used

In **Table 1**, we give the description of the LAPACK kernels used.

Subroutine names	Description
[S,D]GEMM	Single/Double precision matrix-matrix multiply
[S,D]GETRF	Single/Double precision LU factorization routine
[S,D]GETRS	Single/Double precision backward and forward solve routine
[S,D]GESV	Solve a linear system of equations = [S,D]GETRF + [S,D]GETRS
DGEMV	Double precision matrix-vector multiply
DGESIRSV	Single precision LU factorization followed by double precision iterative refinement = SGETRF + ITER.(DGEMM+SGETRS)

Table 1: Description of the different LAPACK/BLAS Kernels used in DGESIRSV and DGESV

4.3 Performance Results

The first set of experiments show the performance of the sequential algorithm on a number of systems. In the third and fourth columns of **Table 2**, for each system, we report the ratio of the time to perform SGEMM (Single precision Matrix-Matrix multiply for GEneral matrices) over the time to perform DGEMM (Double precision Matrix-Matrix multiply for GEneral matrices) and the ratio of the time to perform SGETRF (Single precision LU Factorization for GEneral matrices) over the time to perform DGETRF (Double precision LU Factorization for GEneral matrices). As claimed in the introduction this ratio is often 2 (Katmai, Coppermine, Northwood, Prescott, Opteron, UltraSPARC, X1), which means single are twice as fast as double. Then in the fifth and sixth columns we report the results for DGSEV over DGESIRSV. The results from Table 1 show that this method can be very effective on a number, but not all, architectures. The Intel Pentium, AMD Opteron, Sun UltraSPARC, Cray X1, and IBM Power PC architectures, all exhibits a significant benefit from the use of single precision. Systems such as the Intel Itanium, SGI Octane, and IBM Power3 do not show the benefits.
Note that single precision computation is significantly slower than double precision computation on Intel Itanium 2.

Architecture (BLAS)	*n*	DGEMM /SGEMM	DGETRF /SGETRF	DGESV /DGESIRSV	# iter
Intel Pentium III Coppermine (Goto)	3500	2.10	2.24	1.92	4
Intel Pentium III Katmai (Goto)	3000	2.12	2.11	1.79	4
Sun UltraSPARC IIe (Sunperf)	3000	1.45	1.79	1.58	4
Intel Pentium IV Prescott (Goto)	4000	2.00	1.86	1.57	5
Intel Pentium IV-M Northwood (Goto)	4000	2.02	1.98	1.54	5
AMD Opteron (Goto)	4000	1.98	1.93	1.53	5
Cray X1 (libsci)	4000	1.68	1.54	1.38	7
IBM Power PC G5 (2.7 GHz) (VecLib)	5000	2.29	2.05	1.24	5
Compaq Alpha EV6 (CXML)	3000	0.99	1.08	1.01	4
IBM SP Power3 (ESSL)	3000	1.03	1.13	1.00	3
SGI Octane (ATLAS)	2000	1.08	1.13	0.91	4
Intel Itanium 2 (Goto and ATLAS)	1500	0.71			

Table 2: Ratio of Execution Times (speedup) for DGEMM/SGEMM (m=n=k), DGETRF/SGETRF and DGESV/DGESIRSV on various Architectures.

In **Figure 2**, the left graphs show the performance in Gflop/s of various kernels. Ultimately one wants to compare the dashed blue line (DGESIRSV) with the solid blue line (DGESV). One can see that the performance of the DGESIRSV (dark blue line with stars) comes close to the performance of SGETRF (green line with triangles).

The difference between DGESIRSV and SGETRF is mainly due to the $O(n^2)$ components in the iterative refinement; those terms turn out to be in general small but not negligible.

The graphs on the right gives the percent of the most important kernels versus DGETRF. The red line with diamond represents DGESV and is close to 100% (i.e. DGETRS is negligible with respect to DGETRF). The grey line with squares (DGESIRSV) is the sum of the green line with triangle (SGETRF), the cyan line with x (SGETRS), the dark blue line with stars (DGEMV), and the magenta line with circles (conversion double-single, copies, DAXPY, …). We observe that the $O(n^2)$ operations in iterative refinement take up to 10% of the time of DGETRF an $O(n^3)$ operation. This phenomenon is due to the fact that iterative refinement uses kernels that are hard to optimized; SGETRS, DGEMV and Level 1 BLAS). As can be expected, as n increases the performance of DGESIRSV becomes close to the performance of SGETRF.

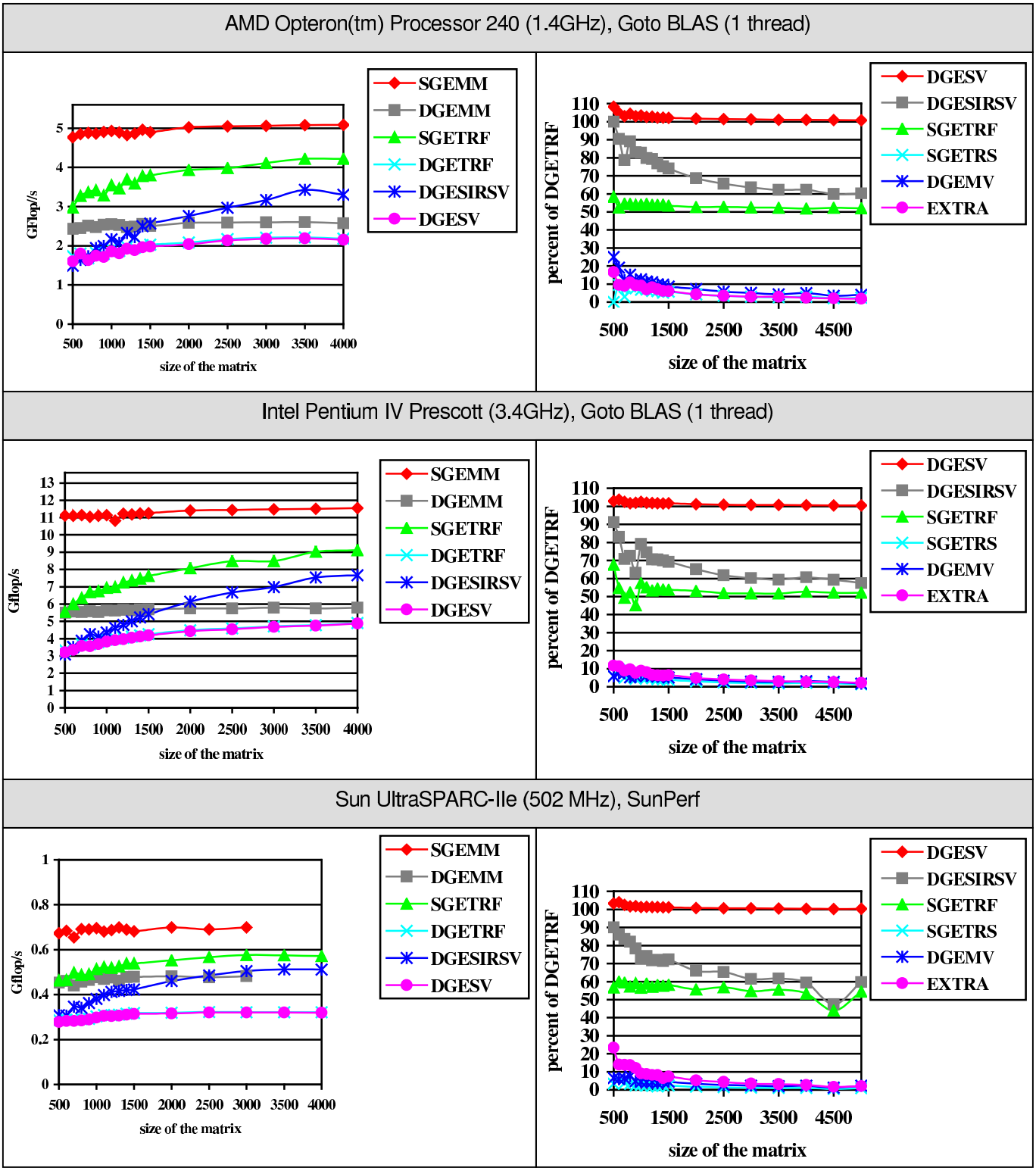

Figure 2: Performance Comparison DGESIRSV/DGESV and DGESIRSV's Kernels on three Different Architectures.

The penalty of this method is that storage is increased by a factor 1.5 over what is required for the standard double precision solution method. The performance enhancements primarily come about when the speed of single precision arithmetic is significantly greater than double precision arithmetic.

The next set of experiments is for a parallel implementation along the lines of ScaLAPACK [11]. In this case *n* is in general fairly large and, as we can observe in Table 3 or Figure 3, the cost of the iterative refinement $O(n^2)$ becomes negligible with respect to PDGETRF $O(n^3)$. Using PDGESIRSV is almost twice (1.83) as fast as opposed to using PDGESV for the same accuracy.

Architecture (BLAS-MPI)	# procs	*n*	PDGETRF /PSGETRF	PDGESV / PDGESIRSV	# iter
AMD Opteron (Goto – OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto – OpenMPI MX)	64	32000	1.90	1.83	6

Table 3: Performance Comparison between PDGETRF/PSGETRF and PDGESV/PDGESIRSV on an AMD Opteron Cluster with Myrinet Interconnects.

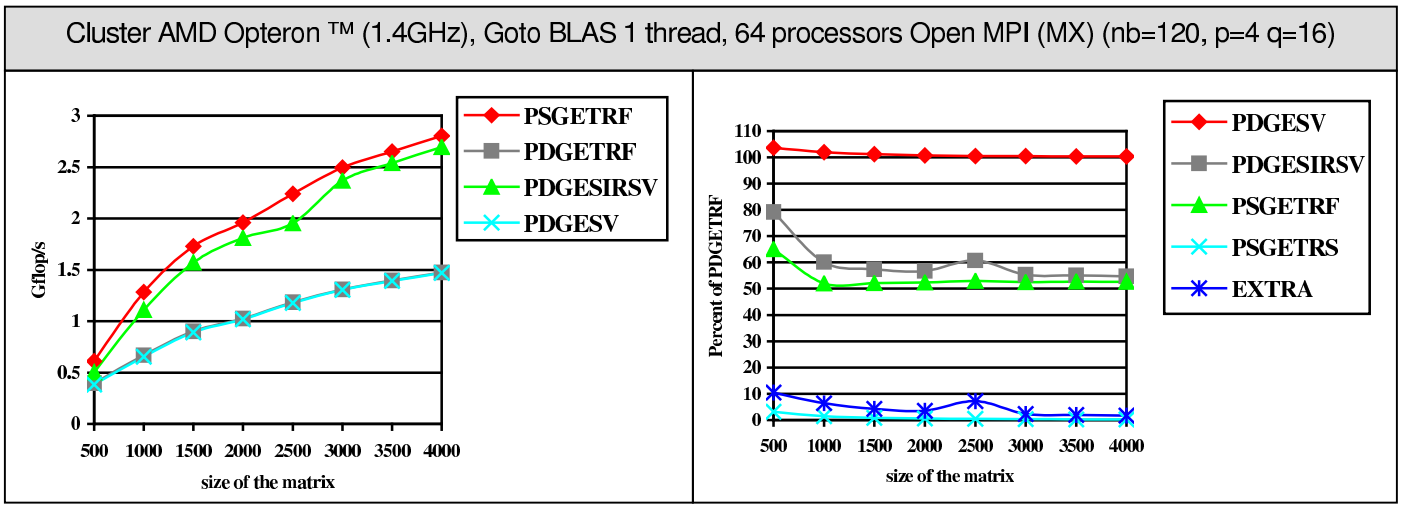

Figure 3: Performance Comparison between PDGETRF/PSGETRF and PDGESV/PDGESIRSV on an AMD Opteron Cluster with Myrinet Interconnects.

5. Quadruple Precision

As an extension to this study, we present in this section results for iterative refinement in quadruple precision on an Intel Xeon 3.2GHz. The iterative refinement code computes a condition number estimate; the input matrices are random matrices with uniform distribution. For quadruple precision arithmetic, we use the reference BLAS compiled with 'ifort -O3' the Intel Fortran compiler (with -O3 optimization flags 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 4. The obtained accuracy is of about 10^{-32} for QGETRF and QDGETRF as expected. No more than 3 steps of iterative refinement are needed. The speedup goes from 10 for a matrix of size 100 to close to 100 for a matrix of size 1000. In Table 5, 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 [16].

	QGESV	QDGESV	
n	time (s)	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 4: Iterative Refinement in Quadruple Precision on a Intel Xeon 3.2GHz.

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

Table 5: Time for the various Kernels in the Quadruple Accuracy Versions for n=900 .

6. Experiments on IBM's Cell Broad Engine

Iterative refinement can also be used in the context of architecture like IBM's Cell Broad Engine. We report here preliminary experiments.

The Cell Broad Engine used for the study is made of 8 SPUs and 1 PPU. Each SPU runs at 3.2 GHz and is able to perform in single 8 operations per cycle. Therefore the theoretical peak of the machine in single precision arithmetic is 3.2 GHz x 8 ops/cycle/SPU x 8 SPUs = 204.8 GFlops/s. This number does not include the PPU, in our code, the PPU mostly control the SPUs and does not participate in the actual computation. For double precision arithmetic, we have 4 operations per cycle and a cycle latency of 7 for a peak of 14.6 Gflop/s, this makes double precision operations 14 times slower than single precision operations.

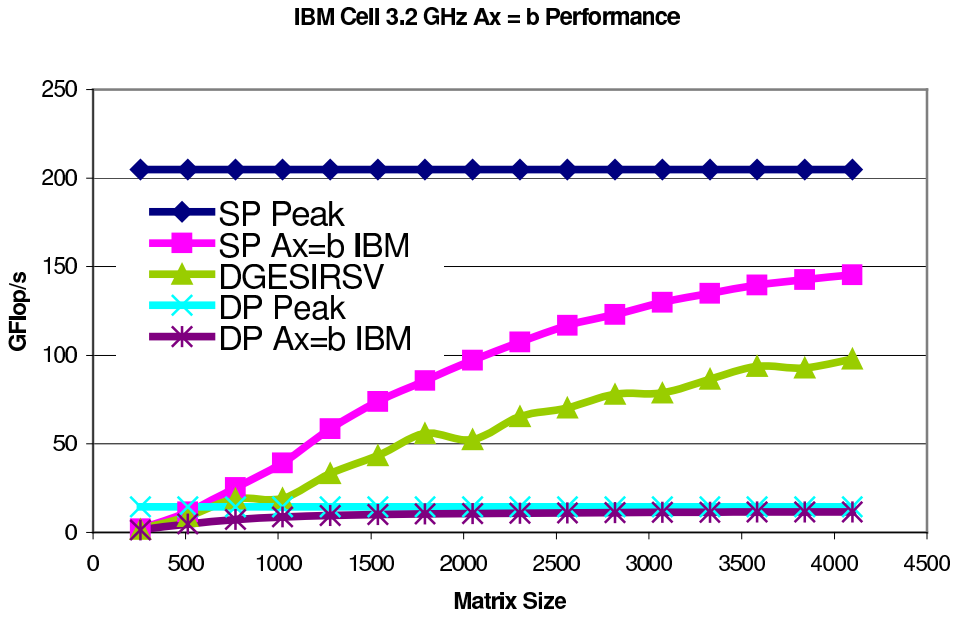

Figure 4: DGESV/DGESIRSV on the IBM Cell Broad Engine.

The main routine to perform the LU factorization in single precision has been borrowed from IBM's LINPACK SP (SGETRF); we have then used our F77 DGESIRSV routine and hand coded the other blocks from BLAS and LAPACK missing.

The results are presented in Figure 4. A 4000-by-4000 matrix can be solved in less than half of a second, which is equivalent to solving the system in double precision at a speed of nearly 100 GFLOPS.

This means that on the Cell the iterative refinement technique fulfills its promise. It achieves performance which is 4.7 times greater than the total double precision peak of the Cell including the PPU (21 GFLOPS), 6.7 times greater than the double precision peak of the SPUs (14.63 GFLOPS), and 8.3 times greater than the actual performance of the solve in double precision (11.8 GFLOPS).

7. Extensions

The ideas expressed here for solving systems for general dense problems can be extended to the case of solving symmetric positive definite matrices using Cholesky factorization, dealing with linear least squares problems with the QR factorization. In addition the iterative refinement concept can be applied to eigenvalue/singular value computation (see [2,3,4,5]). Applying these ideas to sparse matrices is also an option that will be pursued in the future (see [15] for preliminary work in this area).

To summarize, the main benefit comes from performing the bulk of the computation in single precision where the rate of execution is usually higher, perhaps by a factor of 2 and achieving the same accuracy as if the entire computation was performed in double precision. The disadvantage is that the storage requirements are increased by a factor of 1.5 and the input need to be not too ill conditioned.

8. Conclusion

Exploiting 32 bit floating point arithmetic for performance reasons and obtaining full precision (64 bit results) are desirable goals. The results described here are fairly general and can be applied to various problems in linear algebra such as solving dense and large sparse systems using direct or iterative methods and some eigenvalue problems. There are limitations to the success of this process, such as when the conditioning of the problem

exceeds the reciprocal of the accuracy of the single precision computations. In that case the double precision algorithm should be used.

The use of these techniques will have application on the IBM Cell and perhaps extend to Graphical Processing Units (GPUs), such as Nvidia and ATI, where 32 bit floating point arithmetic is native and performed extremely fast. These GPUs may not even have 64 bit floating point hardware and as such the 64 bit operations would have to be emulated in software.

Notes and Comments

The authors would like to thank Clint R. Whaley for insightful comments on machine hardware specification. Neither equilibration nor scaling is performed in the routine DGESIRSV. Adding equilibration and scaling will certainly enhanced the range of applicability of the method. Access to the 3.2 GHz IBM Cell processor was provided by Mercury Computer Systems Inc. We would like to thank IBM for their assistance and access to the technical staff for help with software on the Cell systems.

References

[1] Moler, C. B.: Iterative Refinement in Floating Point. J. ACM (2) (1967) 316–321.
[2] Dongarra, J. J.: Algorithm 589: SICEDR: A FORTRAN Subroutine for Improving the Accuracy of Computed Matrix Eigenvalues. ACM Transactions on Mathematical Software. (4) (1982) 371–375.
[3] Dongarra, J. J., Moler, C. B., and Wilkinson, J. H.: Improving the Accuracy of Computed Eigenvalues and Eigenvectors. SIAM Journal on Numerical Analysis. (1) (1983) 23–45.
[4] Dongarra, J. J.: Improving the Accuracy of Computed Singular Values. SIAM Journal on Scientific and Statistical Computing. (4) (1983) 712–719.
[5] Dongarra, J. J., Geist, G. A., and Romine, C. H.: Algorithm 710: FORTRAN Subroutines for Computing the Eigenvalues and Eigenvectors of a General Matrix by Reduction to General Tridiagonal Form. ACM Transactions on Mathematical Software. (4) (1992) 392–400.
[6] Demmel, J., Hida, Y., Kahan, W., Li, X. S., Mukherjee, S., and Riedy, E. J.: Error Bounds from Extra Precise Iterative Refinement. Technical Report No. UCB/CSD-04-1344, LAPACK Working Note 165August 2004.
[7] Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D.: Introduction to the Cell multiprocessor. IBM J. Res. and Dev. (4/5) (2005) 589–604.
[8] Stewart, G. W.: *Introduction to Matrix Computations*. Academic Press, New York, 1973.
[9] Wilkinson, J. H.: *The Algebraic Eigenvalue Problem*. Oxford, U.K.: Clarendon, 1965.
[10] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J. W., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D.: *LAPACK Users' Guide*. SIAM, third edition, 1999. http://www.netlib.org/lapack/.
[11] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J. W., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C.: *ScaLAPACK Users' Guide*. SIAM Press, 1997.
[12] Rice, J. R.: *Matrix Computations and Numerical Software*. New York, Mc Graw-Hill, 1981.
[13] Demmel, J. W.: *Applied Numerical Linear Algebra*. SIAM Press, 1997.
[14] Higham, N. J.: *Accuracy and Stability of Numerical Algorithms*. 2nd Edition. SIAM Press, 2002.
[15] Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components, *IEEE Proceedings on Field-Programmable Custom Computing Machines FCCM 2006*, Napa, California, USA, April 2006.
[16] Geddes, K. O., Zheng, W. W.: Exploiting fast hardware floating point in high precision computation. *In the Proceedings of the 2003 international symposium on Symbolic and algebraic computation*. Philadelphia, PA, USA. p.111-118, 2003.

Appendix 1

Here is a summary of the x86-x87 ISA Extensions:

MMX
Set of "MultiMedia eXtensions" to the x86 ISA. Mainly new instructions for integer performance, and maybe some prefetch. For Intel, all chips starting with the PentiumMMX processor possess these extensions. For AMD, all chips starting with the K6 possess these extensions.

SSE
Streaming SIMD (Single Instruction Multiple Data) Extensions. SSE is a superset of MMX (i.e., a chip with SSE automatically possesses MMX) These instructions are used to speed up single precision (32 bit) floating point arithmetic. By operating on 4 single precision values with one instruction, they allow for a theoretical peak of 4 FLOPs (FLoating point OPerations) every cycle (eg, a 500Mhz PIII can theoretically perform 2GFlop/s (2 billion FLoating point Operations Per Second)). The results returned by SSE are IEEE compliant (as are classical x86 floating point results). For Intel, all chips listed starting with the Pentium III possess SSE extensions. For AMD, all chips starting from Athlon4 possess SSE.

3DNow!
AMD's extension to MMX that does almost the exact same thing SSE does, except the single precision arithmetic is not IEEE compliant (i.e. it is not as fault-tolerant as x86 arithmetic). It is also a superset of MMX (but not of SSE; 3DNow! was released before SSE). It is supported only on AMD, starting with the K6-2 chip.

Enhanced 3DNow!
An extension to 3DNow! starting with the Athlon onward. Some additional prefetch commands.

3DNow! Professional
AMD's extension that is essentially Enhanced 3DNow! + SSE. Available on AMD chips starting with the Athlon4.

SSE2
Additional instructions that perform double precision floating arithmetic. Allows for 2 double precision FLOPs every cycle. For Intel, supported on the Pentium 4 and for AMD, supported on the Opteron.

The following table lists some of the some of the processors, and the constant to multiply the cycle time by to get peak performance (an entry of 0 indicates that processor does not have the given ISA extension).

	Single				double	
Processor	**x87**	**SSE**	**3DNOW!**	**SSE3**	**x87**	**SSE2**
Pentium / Pentium II	1	0	0	0	1	0
Pentium III	1	4	0	0	1	0
Pentium 4	1	4	0	0	1	2
Athlon/Enhanced Athlon	2	0	4	0	2	0
Athlon 4/MP/XP	2	4	4	0	2	0
AMD Opteron	1	0	0	4	1	2

Appendix 2

We reproduce and expand here a classical proof about iterative refinement. Original proofs can be found in **Stewart "*Introduction to Matrix Computations*" page 200-205** or **Stewart "*Matrix Algorithms: Volume I: Basic Decompositons*" page 221-223** or **Higham "*Accuracy and Stability of Numerical Algorithms*", 2nd edition, Chapter 12, page 231-243**. The model of arithmetic used is the standard one described in those two books.

Originality of the proof given here is two folds:

1. while most of the error analysis results on iterative refinement are given in term of the lower accurate precision, we provide here error analysis in term of the higher accurate precision,
2. To keep the analysis simple, we provide normwise error bounds (as opposed to Higham for example who deals with componentwise error bounds), we believe that the resulting proof is less powerful in term of results but is more easily readable for a standard and non expert reader. Extending the results to componentwise error bound can be done following Higham proof.

Algorithm and floating-point arithmetic relations

We are considering the iterative refinement algorithm

Initialize x_1
for *k= 1, 2, …*
(1) $r_k = b - Ax_k$ ($\square_d$)
(2) Solve $Ad_k = r_k$ ($\square_s$)
(3) $x_{k+1} = x_k + d_k$ ($\square_d$)
end for

performed in floating-point arithmetic where the residual r_k (step 1) and the new approximate solution x_{k+1} (step 3) are computed using double precision ($\square_d$) arithmetic, and the correction vector d_k (step 2) is computed using in single precision ($\square_s$) arithmetic.

We assume that **step 2** is performed using a backward stable algorithm for example Gaussian elimination with partial pivoting, the GMRES iterative method, … Backward stability implies that there exists H_k such that

(1) $(A + H_k)d_k = r_k$ where $\|H_k\| \le \phi(n)\varepsilon_s\|A\|$,

where $\phi(n)$ is a reasonably small function of n. In other words, **Equation (1)** states that the computed solution d_k is an exact solution for an approximated problem.

Step 1 and step 3 are performed in double precision arithmetic and thus the classical error bounds hold:

(2) $r_k = fl(b - Ax_k) \equiv b - Ax_k + e_k$ where $\|e_k\| \le \varphi_1(n)\varepsilon_d(\|A\| \cdot \|x_k\| + \|b\|)$,

(3) $x_{k+1} = fl(x_k + d_k) \equiv x_k + d_k + f_k$ where $\|f_k\| \le \varphi_2(n)\varepsilon_d(\|x_k\| + \|d_k\|)$.

Results and interpretation

Using **Equation (1), (2)** and **(3)**, for any *k*, we will prove that

(4) $\|x - x_{k+1}\| \le \alpha_F\|x - x_k\| + \beta_F\|x\|$,

where α and β are defined as

(5) $\alpha_F = \dfrac{\phi(n)\kappa(A)\varepsilon_s}{1-\phi(n)\kappa(A)\varepsilon_s} + 2\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 2\left(1+\varphi_1(n)\varepsilon_d\right)\varphi_2(n)\kappa(A)\varepsilon_d$

(6) $\beta_F = 4\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 4\left(1+\varphi_1(n)\varepsilon_d\right)\varphi_2(n)\kappa(A)\varepsilon_d$.

Note that α_F and β_F are of the form

(7) $\alpha_F = \psi_F(n)\kappa(A)\varepsilon_s$ and $\beta_F = \rho_F(n)\kappa(A)\varepsilon_d$.

For Equation (4) to hold, we will need to assume that the matrix A is not too-ill conditioned with respect to the single precision (ε_s) arithmetic used, namely we will assume that

(8) $\left(\rho_F(n)\kappa(A)\varepsilon_s\right)\left(1-\psi_F(n)\kappa(A)\varepsilon_s\right)^{-1} < 1$.

This results is proved in **Section Forward Error Analysis**.

Assuming $\alpha_F < 1$, it will then follow that

(9) $\left\|x - x_{k+1}\right\| \le {\alpha_F}^k \left\|x - x_1\right\| + \beta_F \dfrac{1-{\alpha_F}^k}{1-\alpha_F}\left\|x\right\|$,

and so x_k converges to $\tilde{x} \equiv \lim_{k\to+\infty} x_k$ where

$$\lim_{k\to+\infty}\left\|x - x_k\right\| = \left\|x - \tilde{x}\right\| \le \beta_F\left(1-\alpha_F\right)^{-1}\left\|x\right\| = \frac{\rho_F(n)\kappa(A)\varepsilon_d}{1-\psi_F(n)\kappa(A)\varepsilon_s}\left\|x\right\|.$$

This last result is a standard result for the iterative refinement algorithm. This result states that assuming $\varepsilon_d < {\varepsilon_s}^2$, so that $\rho_F(n)\kappa(A)\varepsilon_d < \varepsilon_s$, one can drive the forward error to the level:

(10) $\lim_{k\to+\infty} \dfrac{\left\|x - x_k\right\|}{\left\|x\right\|} \le O(\varepsilon_s)$.

α_F is the rate of convergence and depends on the condition number of the matrix A ($\kappa(A)$) and the single precision used (ε_s). β_F is the limiting accuracy of the method and depends on the double precision used (ε_d).

Result (6) is of interest in mixed precision iterative refinement when one wants to reduce the forward error with respect to the single precision used (ε_s). However in our case, we are interested in double precision (ε_d) accuracy, thus we will write:

(11) $\lim_{k\to+\infty} \dfrac{\left\|x - x_k\right\|}{\left\|x\right\|} \le \dfrac{\rho(n)\kappa(A)\varepsilon_d}{1-\psi(n)\kappa(A)\varepsilon_s}$.

Unfortunately this bound offers nothing surprising and states that one can not reduce the forward error less than the machine precision used time the condition number. To have more insight in the iterative refinement procedure we will need to move to backward error analysis.

(In practice we would have liked to get rid of the forward analysis of the iterative refinement algorithm since it leads us to a bound that we feel uninteresting in our case. However we will need the forward analysis to bound $\| x_k \|$ and $\| d_k \|$ in term of $\| x_{k+1} \|$, which turns to be useful in the backward analysis.)

Regarding the backward error analysis we will prove in **Section Backward Error Analysis** that

(12) $\dfrac{\left\|b - Ax_{k+1}\right\|}{\left\|A\right\|\cdot\left\|x_{k+1}\right\|} \le \alpha_B \cdot \dfrac{\left\|b - Ax_k\right\|}{\left\|A\right\|\cdot\left\|x_k\right\|} + \beta_B$,

where

(13) $\alpha_B = \dfrac{\phi(n)\kappa(A)\gamma\varepsilon_s}{1-\phi(n)\kappa(A)\varepsilon_s} + 2\varphi_1(n)\gamma\varepsilon_d$,

(14) $\beta_B = \left(4\varphi_1(n)\gamma + \varphi_2(n)(1+2\gamma)(1-\varphi_2(n)\varepsilon_d)^{-1}\right)\varepsilon_d$.

Note that α_B and β_B are of the form

(15) $\alpha_B = \psi_B(n)\kappa(A)\varepsilon_s$ and $\beta_B = \rho_B(n)\varepsilon_d$.

For Equation **(11)** to hold, we will need to assume that the matrix A is not too-ill conditioned with respect to the single precision ($\square_s$) arithmetic used, namely we will assume that

(16) $\psi_F(n)\kappa(A)\varepsilon_s + (\rho_F(n)\kappa(A)\varepsilon_s)(1-\psi_F(n)\kappa(A)\varepsilon_s)^{-1} < 1$, and

(17) $(\rho_B(n)\varepsilon_d)(1-\psi_B(n)\kappa(A)\varepsilon_s)^{-1} < 1$.

α_B is the rate of convergence and depends on the condition number of the matrix A ($\kappa(A)$) and the single precision used ($\square_s$). β_B is the limiting accuracy of the method and depends on the double precision used ($\square_d$).

At convergence we have

(18) $\lim\limits_{k\to+\infty} \dfrac{\|b-Ax_k\|}{\|A\|\cdot\|x_k\|} = \beta_B(1-\alpha_B)^{-1} = \dfrac{\rho_B(n)}{1-\psi_B(n)\kappa(A)\varepsilon_s}\varepsilon_d$.

Equation **(15)** means that the solver is normwise backward stable.

Forward error analysis

From Stewart **[*Matrix Algorithms*, Vol. 1, Ch. 1, Th 4.18]**, we know that if $\phi(n)\kappa(A)\varepsilon_s < 1/2$ then $(A+H_k)$ is nonsingular and

(19) $(A+H_k)^{-1} = (I+F_k)A^{-1}$ where $\|F_k\| \le \dfrac{\phi(n)\kappa(A)\varepsilon_s}{1-\phi(n)\kappa(A)\varepsilon_s} < 1$.

From **Equation (1)** and **Equation (3)** we have

$$x - x_{k+1} = x - x_k - (A+H_k)^{-1}r_k - f_k,$$

then using **Equation (2)** and **Equation (18)**, we get

$$x - x_{k+1} = x - x_k - (I+F_k)A^{-1}(b - Ax_k + e_k) - f_k,$$

rearranging a litltle bit, we have

$$x - x_{k+1} = x - x_k - (I+F_k)(x - x_k + A^{-1}e_k) - f_k,$$

and this finally gives us:

$$x - x_{k+1} = -F_k(x - x_k) - (I+F_k)A^{-1}e_k - f_k.$$

Taking the norms of both sides and using the fact that $\|F\| < 1$ gives us

$$\|x - x_{k+1}\| \le \|F_k\|\cdot\|x - x_k\| + 2\cdot\|A^{-1}\|\cdot\|e_k\| + \|f_k\|.$$

Using **Equation (2)** and **Equation (3)**, we get

(20) $\|x - x_{k+1}\| \le \|F_k\|\cdot\|x - x_k\| + 2\varphi_1(n)\varepsilon_d\|A^{-1}\|\cdot(\|A\|\cdot\|x_k\| + \|b\|) + \varphi_2(n)\varepsilon_d(\|x_k\| + \|d_k\|)$.

In **Equation (20)**, **(21)**, and **(22)**, we are going to bound $\|x_k\|$, $\|A\|\cdot\|x_k\| + \|b\|$ and $\|d_k\|$ respectively by $\|x - x_k\|$ and $\|x\|$. Next step will be to inject those three bounds in **Equation (19)**, and then we will be done with our final result on forward error.

Triangle inequality gives us

(21) $\|x_k\| \le \|x - x_k\| + \|x\|$.

Then using the fact that $Ax = b$,

(22) $\|A\| \cdot \|x_k\| + \|b\| \le \|A\| \cdot \|x - x_k\| + 2 \cdot \|A\| \cdot \|x\|$.

Finally using **Equation (1)** and **Equation (4)**,

$$\|d_k\| = \|(A + H_k)^{-1} r_k\| = \|(I + F_k) A^{-1} r_k\| \le 2\|A^{-1}\| \cdot \|r_k\| \ .$$

Since from **Equation (2)** we have

$$\|r_k\| \le \|b\| + \|A\| \cdot \|x_k\| + \|e\| \le (1 + \varphi_1(n)\varepsilon_d) \cdot (\|A\| \cdot \|x_k\| + \|b\|) \ ,$$

using **Equation (20)** in this latter inequality we obtain

(23) $\|d_k\| \le 2 \cdot (1 + \varphi_1(n)\varepsilon_d) \cdot \kappa(A) \cdot (\|x - x_k\| + 2 \cdot \|x\|)$.

Injecting **Equation (20)**, **(21)**, and **(22)** in **Equation (19)** leads us to our first result:

$$\|x - x_{k+1}\| \le \left[\frac{\phi(n)\kappa(A)\varepsilon_s}{1 - \phi(n)\kappa(A)\varepsilon_s} + 2\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 2(1 + \varphi_1(n)\varepsilon_d)\varphi_2(n)\kappa(A)\varepsilon_d \right] \cdot \|x - x_k\|$$
$$+ (4\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 4(1 + \varphi_1(n)\varepsilon_d)\varphi_2(n)\kappa(A)\varepsilon_d) \cdot \|x\| \ .$$

If we define α_F and β_F as

(24) $\alpha_F = \frac{\phi(n)\kappa(A)\varepsilon_s}{1 - \phi(n)\kappa(A)\varepsilon_s} + 2\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 2(1 + \varphi_1(n)\varepsilon_d)\varphi_2(n)\kappa(A)\varepsilon_d$,

(25) $\beta_F = 4\varphi_1(n)\kappa(A)\varepsilon_d + \varphi_2(n)\varepsilon_d + 4(1 + \varphi_1(n)\varepsilon_d)\varphi_2(n)\kappa(A)\varepsilon_d$,

then we find that

$$\|x - x_{k+1}\| \le \alpha_F \|x - x_k\| + \beta_F \|x\| \ ,$$

where $\alpha_F = \psi(n)\kappa(A)\varepsilon_s$ and $\beta_F = \rho(n)\kappa(A)\varepsilon_d$.

Bound on $\|x_k\|$ and $\|d_k\|$ in term of $\|x_{k+1}\|$

A result that will be useful later is to note that, assuming $x_1 = 0$ for simplicity and without loss of too much generality, from Equation **(9)**, we can write the two following inequalities:

$$\|x_k\| \le \left[1 + \alpha_F{}^{k-1} + \beta_F \frac{1 - \alpha_F{}^{k-1}}{1 - \alpha_F} \right] \cdot \|x\| \ ,$$

$$\|x\| \le \left[1 - \alpha_F{}^{k} - \beta_F \frac{1 - \alpha_F{}^{k}}{1 - \alpha_F} \right]^{-1} \cdot \|x_{k+1}\| \ .$$

Assuming that $\alpha_F + \frac{\beta_F}{1 - \alpha_F} < 1$, we get

$$\|x_k\| \le \frac{\left[1 + \alpha_F{}^{k-1} + \beta_F \frac{1 - \alpha_F{}^{k-1}}{1 - \alpha_F} \right]}{\left[1 - \alpha_F{}^{k} - \beta_F \frac{1 - \alpha_F{}^{k}}{1 - \alpha_F} \right]} \cdot \|x_{k+1}\| \ ,$$

and, by defining

$$\gamma_k \equiv \frac{\left[1+{\alpha_F}^{k-1}+\beta_F\frac{1-{\alpha_F}^{k-1}}{1-\alpha_F}\right]}{\left[1-{\alpha_F}^{k}-\beta_F\frac{1-{\alpha_F}^{k}}{1-\alpha_F}\right]} \le \gamma,$$

we have, for any k,

(26) $\|x_k\| \le \gamma \cdot \|x_{k+1}\|$.

Using **Equation (3)** we have

$$\|d_k\| = \|x_{k+1} - x_k - f_k\| \le \|x_{k+1}\| + (1+\varphi_2(n)\varepsilon_d)\|x_k\| + \varphi_2(n)\varepsilon_d\|d_k\|,$$

so with **Equation (26)** this gives us

(27) $\|d_k\| \le (1-\varphi_2(n)\varepsilon_d)^{-1}(1+\gamma+\varphi_2(n)\gamma\varepsilon_d)\|x_{k+1}\|$.

Note that here we have assumed

(28) $\alpha_F + \frac{\beta_F}{1-\alpha_F} < 1$.

Backward error analysis

From Stewart **[*Matrix Algorithms*, Vol. 1, Ch. 1, Th 4.18]**, we know that if $\phi(n)\kappa(A)\varepsilon_s < 1/2$ then $(A+H_k)$ is nonsingular and

(29) $(A+H_k)^{-1} = A^{-1}(I+G_k)$ where $\|G_k\| \le \frac{\phi(n)\kappa(A)\varepsilon_s}{1-\phi(n)\kappa(A)\varepsilon_s} < 1$.

(Note:

As Stewart mentioned, the $(I+X_k)$ matrix can be put on the left or on the right side of A. See **Equation (19)** for the left side and **Equation (29)** for the right side. Here the proof.

Since $\|A^{-1}H_k\| < 1$, we have $\|(A+H_k)^{-1}\| < \frac{\|A^{-1}\| \cdot \|A^{-1}H_k\|}{1-\|A^{-1}H_k\|}$ (see Stewart).

For the right side (**Equation (29)**), one goes as

$$(A+H_k)^{-1} - A^{-1} = (I - A^{-1}(A+H_k))(A+H_k)^{-1} = -A^{-1}H_k(A+H_k)^{-1},$$

$$(A+H_k)^{-1} = A^{-1}(I+G_k) \text{ where } G_k = -H_k(A+H_k)^{-1}.$$

For the right side (**Equation (19)**), one goes as

$$(A+H_k)^{-1} - A^{-1} = (A+H_k)^{-1}(I-(A+H_k)A^{-1}) = -(A+H_k)^{-1}H_kA^{-1},$$

$$(A+H_k)^{-1} = (I+F_k)A^{-1} \text{ where } F_k = -(A+H_k)^{-1}H_k.$$

From **Equation (1)** and **Equation (3)** we have

$$x - x_{k+1} = x - x_k - (A+H_k)^{-1}r_k - f_k,$$

then using **Equation (2)** and **Equation (29)**, we get

$$x - x_{k+1} = x - x_k - A^{-1}(I+G_k)(b - Ax_k + e_k) - f_k,$$

multiplying (on the left A) we finally get

$$b - Ax_{k+1} = -G_k(b - Ax_k) - (I+G_k)e_k - Af_k.$$

Taking the norms of both sides and using the fact that $\|F\| < 1$ gives us

$$\|b - Ax_{k+1}\| \le \|G_k\| \cdot \|b - Ax_k\| + 2 \cdot \|e_k\| + \|A\| \cdot \|f_k\|.$$

Using **Equation (2)** and **Equation (3)** we have

$$\|b - Ax_{k+1}\| \le \|G_k\| \cdot \|b - Ax_k\| + (2\varphi_1(n) + \varphi_2(n))\varepsilon_d \cdot \|A\| \cdot \|x_k\| + 2\varphi_1(n)\varepsilon_d \cdot \|b\| + \varphi_2(n)\varepsilon_d \cdot \|A\| \cdot \|d_k\|.$$

Assuming **Equation (28)** holds, we can use **Equation (26)** (we recall : $\|x_k\| \le \gamma \cdot \|x_{k+1}\|$), **Equation (27)** (we recall: $\|d_k\| \le (1 - \varphi_2(n)\varepsilon_d)^{-1}(1 + \gamma + \varphi_2(n)\gamma\varepsilon_d)\|x_{k+1}\|$). Using the fact that $\|b\| = \|b - Ax_k\| + \|A\| \cdot \|x_k\|$ we get:

$$\begin{aligned}\|b - Ax_{k+1}\| \le{}& (\|G_k\| + 2\varphi_1(n)\varepsilon_d) \cdot \|b - Ax_k\| \\ &+ (4\varphi_1(n)\gamma + \varphi_2(n)\gamma + \varphi_2(n)(1 - \varphi_2(n)\varepsilon_d)^{-1}(1 + \gamma + \varphi_2(n)\gamma\varepsilon_d))\varepsilon_d \cdot \|A\| \cdot \|x_{k+1}\|.\end{aligned}$$

So finally

$$\frac{\|b - Ax_{k+1}\|}{\|A\| \cdot \|x_{k+1}\|} \le \alpha_B \cdot \frac{\|b - Ax_k\|}{\|A\| \cdot \|x_k\|} + \beta_B,$$

where

$$\alpha_B = \frac{\phi(n)\kappa(A)\gamma\varepsilon_s}{1 - \phi(n)\kappa(A)\varepsilon_s} + 2\varphi_1(n)\gamma\varepsilon_d,$$

$$\beta_B = (4\varphi_1(n)\gamma + \varphi_2(n)(1 + 2\gamma)(1 - \varphi_2(n)\varepsilon_d)^{-1})\varepsilon_d.$$

Appendix 3: Algorithm

Algorithm DGESIRSV Mixed precision iterative refinement

Input: $A \in \mathbf{R}^{n*n}_{(64)}$, $b \in \mathbf{R}^{n}_{(64)}$ (64-bit precision)
Output: $x \in \mathbf{R}^{n}_{(64)}$ accurate in 64-bit precision (in the backward error sense)

Make 32-bit precision copy of A and b
$A_{(32)}, b_{(32)} \leftarrow A, b$
Compute LU factorization in 32-bit precision: $L_{(32)}\ U_{(32)} \leftarrow P_{(32)}A_{(32)}$
$L_{(32)}, U_{(32)}, P_{(32)} \leftarrow$ **SGETRF**$(A_{(32)})$
Apply back-solve in 32-bit precision with 32-bit precision factors
$x^{(1)}_{(32)} \leftarrow$ **SGETRS**$(L_{(32)}, U_{(32)}, P_{(32)}, b_{(32)})$
Promote the solution from 32-bit precision to 64-bit precision
$x^{(1)} \leftarrow x^{(1)}_{(32)}$
$i \leftarrow 0$
repeat
  $i \leftarrow i+1$
  Compute residual in 64-bit precision
  $r^{(i)} \leftarrow b - Ax^{(i)}$
  Demote the residual from 64-bit precision to 32-bit precision
  $r^{(i)}_{(32)} \leftarrow r^{(i)}$
  Back-solve on 32-bit precision residual and 32-bit precision factors
  $z^{(i)}_{(32)} \leftarrow$ **SGETRS**$(L_{(32)}, U_{(32)}, P_{(32)}, r^{(i)}_{(32)})$
  Promote the correction from 32-bit precision to 64-bit precision
  $z^{(i)} \leftarrow z^{(i)}_{(32)}$
  Update solution in 64-bit precision
  $x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$
until $(||r^{(i)}||_2 < sqrt(n) \leftarrow ||A||_{fro}\ ||x^{(i)}||_2$ or (i > 30)

if $(||r^{(i)}||_2 \leftarrow sqrt(n) \leftarrow ||A||_{fro}\ ||x^{(i)}||_2$ then
  Refinement procedure failed to converge
  Compute solution in 64-bit precision using LU factorization
  $x^{(i+1)} \leftarrow$ **DGESV**(A, b)
end if