Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy*

Alfredo Buttari[1], *Jack Dongarra*[1,2], *Jakub Kurzak*[1], *Piotr Luszczek*[1], *and Stanimire Tomov*[1]

[1]University of Tennessee Knoxville
[2]Oak Ridge National Laboratory

October 22, 2006

Abstract

By using a combination of 32-bit and 64-bit floating point arithmetic the performance of many sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution. These ideas can be applied to sparse multifrontal and supernodal direct techniques, and sparse iterative techniques such as Krylov subspace methods. The approach presented here can apply not only to conventional processors but also to exotic technologies such as Field Programmable Gate Arrays (FPGA), Graphical Processing Units (GPU), and the Cell BE processor.

1 Introduction

Benchmarking analysis and architectural descriptions reveal that on many commodity processors the performance of 32-bit floating point arithmetic (single precision computations) may be significantly higher than 64-bit floating point arithmetic (double precision computations). This is due to a number of factors. First, many processors have vector instructions like the SSE2 instruction set on the Intel IA-32 and IA-64 and AMD Opteron family of processors or the AltiVec unit on the IBM PowerPC architecture; in the SSE2 case a vector unit can complete four single precision operations every clock cycle but can complete only two in double precision. (For the AltiVec, single precision can complete 8 floating point operations per cycle as opposed to 4 floating point operations in double precision.) Another reason lies in the fact that single precision data

*This work was supported in part by the National Science Foundation and Department of Energy.

can be moved at a higher rate through the memory hierarchy as a result of a reduced amount of data to be transferred. Finally, the fact that single precision data occupies less memory than double precision data also means that more single precision values can be held in cache than the double precision counterpart which results in a lower rate of cache misses (the same reasoning can be applied to Translation Look-aside Buffers – TLBs). A combination of these factors can lead to significant enhancements in performance as we will show for sparse matrix computations in the following sections. A remarkable example is the IBM Cell BE processor where the single precision floating point arithmetic peak performance is more than an order of magnitude higher than the double precision (204.8 GFlop/s vs 14.6 GFlop/s for the 3.2 GHz version of the chip). The performance of some linear algebra operations can be improved based on the consideration that the most computationally expensive tasks can be performed exploiting single precision operations and only resorting to double precision at critical stages while attempting to provide the full double precision accuracy. This technique, is supported by the well known theory of iterative refinement [1, 2], which has been successfully applied to the solution of dense linear systems [3]. This work is an extension of the work by Langou [3] to the case of sparse linear systems, covering both direct and iterative solvers.

2 Sparse Direct and Iterative Solvers

Most sparse direct methods for solving linear systems of equations are variants of either multifrontal [4] or supernodal [5] factorization approaches. There is a number of freely available packages which implement these methods. We have chosen for our tests the software package MUMPS [6, 7, 8] as the representative of the multifrontal approach and SuperLU [9, 10, 11, 12] for the supernodal approach. Our main reason for selecting these two software packages is that they are implemented in both single and double precision which is not the case for other freely available solvers such as UMFPACK [13, 14, 15].

Fill-ins, and the associated memory requirements, are inherent for direct sparse methods. And although there are various reordering techniques designed to minimize the amounts of these fill-ins, for problems of increasing size there is a point where the memory requirements become prohibitively high and direct sparse methods are no longer feasible. Iterative methods are a remedy because only a few working vectors and the primary data are required (see for example [16, 17]).

Two popular methods on which we will illustrate the techniques addressed in this paper are the conjugate gradient (CG) method (for symmetric and positive definite matrices) and the generalized minimal residual (GMRES) method (for non-symmetric matrices, see [18]). The preconditioned versions of the two algorithms are given correspondingly in Tables 1 and 2 (the descriptions follow [16, 17]).

The preconditioners, denoted in both cases by M, are operators intended to improve the robustness and the efficiency of the iterative algorithms. In particular, we will use *left* preconditioning, where instead of

$$Ax = b$$

```
        PCG ( b, x_o, E_tol, ... )
 1      r_o = b − A x_o
 2      d_o = 0
 3      for i = 1,2,...
 4          z_{i−1} = M r_{i−1}
 5          β = (r_{i−1} · z_{i−1}) / (r_{i−2} · z_{i−2})
 6          d_i = z_{i−1} + β d_{i−1}
 7          α = (r_{i−1} · z_{i−1}) / (d_i · A d_i)
 8          x_i = x_{i−1} + α d_i
 9          r_i = r_{i−1} − α A d_i
10          check convergence and exit if done
11      end
```

Table 1: The Preconditioned Conjugate Gradient (PCG) algorithm in solving $Ax = b$ with preconditioner M and initial guess x_0.

```
        GMRES ( b, x_o, E_tol, m, ... )
 1      for i = 0,1,...
 2          r_i = b − A x_i
 3          β = h_{1,0} = ||r_i||_2
 4          check convergence and exit if done
 5          for k = 1,...,m
 6              v_k = r_i / h_{k,k−1}
 7              r_i = A M v_k
 8              for j = 1,...,k
 9                  h_{j,k} = r_i · v_j
10                  r_i = r_i − h_{j,k} v_j
11              end
12              h_{k+1,k} = ||r_i||_2
13          end
            // Take V_m = [v_1,...,v_m], H_m = {h_{i,j}}
14          Find W_m = [w_1,...,w_m]^T that minimizes ||b − A(x_i + M V_m W_m)||_2
            // as the minimizer of ||β e_1 − H_m W_m||_2 as well
15          x_i = x_i + M V_m W_m
16      end
```

Table 2: The GMRES(m) algorithm with right preconditioning in solving $Ax = b$ with preconditioner M and initial guess x_0.

we solve $MAx = Mb$, and *right* preconditioning, where the problem is transformed to $AMu = b$, $x = Mu$. Intuitively, to serve its purpose, M needs to be easy to compute, apply, and store, and to approximate A^{-1}.

The basic idea of our approach, namely to use faster but lower precision computations whenever possible, can be used to design preconditioners M featuring the above mentioned two requirements, as we show in the rest of the paper. And since our basic idea can be exploited (in iterative solvers) through proper preconditioning, the applicability of the approach is far-reaching, and not limited to either the preconditioners or the solvers used to demonstrate the idea in this paper.

3 Mixed Precision Iterative Refinement

The iterative refinement technique is a well known method that has been extensively studied and applied in the past. A fully detailed description of this method can be found in [1, 2, 19]. The iterative refinement approach has been used in the past to improve the accuracy of the solution of linear systems and it can be summarized as:

(1) $x_0 \leftarrow A \backslash b$
$k = 1$
untill convergence **do**:
(2) $r_k \leftarrow b - Ax_{k-1}$
(3) $z_k \leftarrow A \backslash r_k$
(4) $x_k \leftarrow x_{k-1} + z_k$
$k+ = 1$
done

Once the system is solved at step 1, the solution can be refined through an iterative procedure where, at each step k, the residual is computed based on the solution at step $k-1$ (step 2), a correction is computed as in step 3 and finally this correction is applied as in step 4. While the common usage of iterative refinement (see [20]) consists of performing all the arithmetic operations with the same precision (either single or double), we have investigated the application of mixed precision iterative refinement where the most expensive steps 1 and 3 are performed in single precision and steps 2 and 4 are performed in double precision. Related work can be found in [21, 22, 23]. The error analysis for the mixed precision iterative refinement is explained in [3] and shows that using this approach it is possible to achieve the same accuracy as if the system was solved in full double precision arithmetics provided that the matrix is not too badly conditioned. From a performance point of view the potential of this method lies in the fact that the most computationally expensive steps 1 and 3 can be performed very fast in single precision arithmetic while the only tasks that require double precision accuracy are the steps 2 and 4 whose cost can be considered much less.

3.1 Mixed Precision Iterative Refinement for Sparse Direct Solvers

Using the MUMPS package for solving systems of linear equations can be described in three distinct steps:

1. System Analysis: in this phase the system sparsity structure is analyzed in order to estimate the element growth which provides an estimate of the memory requirement which 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 $Ux = y$

Once steps 1 and 2 are performed, each iteration of the refinement loop needs only to perform the system solution (i.e. step 3) whose cost is negligible when compared to the system factorization. The implementation of mixed precision iterative refinement method with the MUMPS package can, thus, be summarized as:

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

At the end of each line of the algorithm we indicate the precision used to perform this operation as either ε_s for single precision computation or ε_d for double precision computation. Based on backward stability analysis, we consider that the solution x is of double precision quality when

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

where $\|\cdot\|_{fro}$ is the Frobenius norm and n is the problem size. This provides us a stopping criterion. If some maximum number of iterations is reached, then the algorithm should signal failure to converge. All the control parameters for the MUMPS solver have been set to their default values which means that the matrix scaling, matrix permuting and pivoting order strategies are determined at runtime based on the matrix properties.

3.2 Mixed Precision Iterative Refinement for Sparse Iterative Solvers

The general framework of mixed precision iterative refinement given at the beginning of this Section can be easily extended to sparse iterative solvers. Indeed, it can be interpreted as a preconditioned Richardson iteration in solving

$$MAx = Mb,$$

where the preconditioner M represents A^{-1} as being computed and applied in single/lower precision arithmetic. This interpretation can be further extended to any preconditioned iterative method. And in general, as long as the iterative method at hand is backward stable and converges, one can apply similar reasoning [3] to show that the solution obtained would be accurate in higher precision.

The feasibility of introducing lower precision computations in the preconditioner depends first on whether there is a potential to introduce speedups in the computation, and second on how the method's robustness would change.

We distinguish three approaches for doing this. First, this is when any higher precision data is simply replaced with a corresponding lower precision, but all the computations are still done in higher precision. The success of this approach, regarding speed, depends on what percent of the overall computation is spent on the preconditioner. For example, a simple diagonal preconditioner may not benefit from it, while a domain decomposition-based block diagonal preconditioner, or a multigrid V-cycle, may benefit. Also multigrid-based solvers may benefit both in speed (as the bulk of the computation is in their V/W-cycles) and memory requirements. An example of successful application of this type of approach in CFD was done [24, 25] in a PETSc solver which was accelerated with a Schwartz preconditioner using block-incomplete factorizations over the separate subdomains that are stored in single precision. Regarding robustness, there are various algorithmic issues to consider, including ways to automatically at run time determine limitations of the approach. This brings up another possible idea to explore, which is the use of lower precision arithmetic only for parts of the preconditioner. Examples here may come from adaptive methods which automatically locate the singularities of the solution sought, and hence the corresponding parts of the matrix responsible for resolving them. This information may be used in combination with the solver and preconditioner (e.g. hierarchical multigrid) to achieve both speedup and robustness of the method.

The second approach, a straightforward extension of the first, is when not just the higher precision storage but also the higher precision arithmetic are replaced with lower precision. This is the case that would allow one to apply the technique not only to conventional processors but also to FPGAs, GPUs, Cell BE processor, etc.

The third approach, and the focus of the current work, is to enable the efficient use of lower precision arithmetic to sparse iterative methods in general, when no preconditioner, or when just a simple and computationally inexpensive (relative to the rest of the computation) preconditioner is available. The idea of accomplishing this is to use the preconditioned version of the iterative method at hand and replace the preconditioner M by an iterative method as well, but implemented in reduced precision arithmetic.

Thus, by controlling the accuracy of this iterative *inner* solver more computations can be done in reduced precision and less work needed in the full precision arithmetic.

The robustness of variations of this *nesting* of iterative methods, known in the literature also as *inner-outer* iteration, has been studied before, both theoretically and computationally (see for example [26, 27, 28, 29, 30, 31, 32]). The general appeal of these methods is that computational speedup is possible when the inner solver uses an approximation to the original matrix that is also faster to apply. Moreover, even if no faster matrix-vector product is available, speedup can be often observed due to improved convergence (e.g. see restarted GMRES *vs* GMRES-FGMRES [28] and Subsection 4.3). To our knowledge using mixed precision for performance enhancement has not been done in the framework suggested in this paper. In the subsections below we show a way to do it for CG and GMRES. We would refer below to single precision (SP) as 32-bit and to double precision (DP) as 64-bit floating point arithmetic and also lower and higher precision arithmetic will be correspondingly associated with SP and DP.

3.2.1 CG-based Inner-Outer Iteration Methods

We suggest the PCG-PCG inner-outer algorithm given in Table 3. The algorithm is given as a modification to the reference PCG algorithm from Table 1, and therefore only the lines that change are written out. The inner PCG is in SP where SP data and arithmetic are colored in blue. The preconditioner available for the reference PCG is used in SP in the inner PCG. Note that our initial guess in the inner loop is always taken to be 0 and we perform a fixed number of iterations (step 10 is in brackets since practically we want to avoid exiting due to small residual, unless it is of order of machine's single precision). It is not clear what is the optimal number of inner iterations. In our implementation the first outer iteration is unrolled and the call to `PCG_single` sets it as the number of iterations it took to do a fixed (e.g. 0.3) relative reduction for the residual.

If a preconditioner is not available, we can similarly define a CG-PCG algorithm, where the inner loop is just a CG in SP. Furthermore, other iterative solvers can be used for the inner loop, as long as they result in symmetric and positive definite (SPD) operators. For example stationary methods like Jacobi, Gauss-Seidel (combination of one backward and one forward to result in SPD operator), and SSOR can be used. Note that with these methods a constant number of iterations and initial guess 0 result in a constant preconditioner, and hence in optimal convergence for the outer PCG iteration. The use of a Krylov space method in the inner iteration, as in the currently considered algorithm, results in a non-constant preconditioner. Although there is convergence theory for these cases [28], it still remains to be resolved how to set the stopping criteria, variations in the algorithms, etc. [26, 30] in order to obtain optimal results. For example G. Golub and Q. Ye in [26] consider the inexact PCG (β is taken as $\frac{(r_{i-1}-r_{i-2})\cdot z_{i-1}}{r_{i-2}\cdot z_{i-2}}$), which allows certain local orthogonality relations to be preserved from the standard PCG, which on the other hand gives grounds for theoretically studied aspects of the algorithm. We tried this approach as well, and although our numerical results confirmed the similar findings [26], overall the algorithm described here gave better results. In general non-constant preconditioning deteriorates the CG convergence, often resulting

```
       PCG_PCG ( b, x_o, E_tol, ... )
       ...
 4        PCG_single( r_i-1, z_i-1, NumIters, ... )
       ...
11     end


       PCG_single( b, x, NumIters, ... )
 1     r_o = b; x_o = 0
 2     d_o = 0
 3     for i = 1, ..., NumIters
          ...
10        [check convergence and break if done]
11     end
12     x = x_i
```

Table 3: PCG-PCG algorithm with inner PCG in SP (SP data and arithmetic colored in blue). Only the lines that change from the reference PCG algorithm on Table 1 are given.

in convergence that is characteristic of the steepest descent algorithm. Still, shifting the computational load to the inner PCG, reduces this effect and gives convergence that is comparable to the convergence of a reference PCG algorithm. Note that in our case by setting an inner number of iterations so that a fixed relative reduction for the residual is achieved, we expect to have only a constant number of outer iterations until convergence.

3.2.2 GMRES-based Inner-Outer Iteration Methods

For our outer loop we take the flexible GMRES (FGMRES [27, 17]). This is a minor modification to the algorithm from Table 2 meant to accommodate non-constant preconditioners. Note that on line 7 the preconditioner M actually depends on v_k, so the update of x_i at line 15 will make the algorithm unstable if first $V_m W_m$ is computed and then apply M. This can be remedied for the price of m additional storage vectors. Our GMRES-FGMRES inner-outer algorithm is given on Table 4 (the additional vectors are introduced at line 7: $z_k = Mv_k$ where M is replaced with the `GMRES_single` solver).

As with PCG-PCG, the algorithm is given as a modification to the reference GMRES(m) algorithm from Table 2, and therefore only the lines that change are written out. The inner GMRES is in SP where SP data and arithmetic are colored in blue. The preconditioner available for the reference GMRES is used in SP in the inner GMRES. Note that again our initial guess in the inner loop is always taken to be 0 and we perform a fixed number of cycles (in this case just 1; step 4 is in brackets since we want to avoid exiting due to small residual, unless it is of order of machine's single precision).

The potential benefits of FGMRES compared to GMRES are becoming better un-

```
        GMRES_FGMRES ( b, x_o, E_tol, m_1, m_2, ... )
        ...
 5        for k = 1,...,m_2
             ...
 7           GMRES_single( v_k, z_k, 1, m_1, ... )
             r_i = A z_k
             ...
15        x_i = x_i + Z_m W_m
16      end


        GMRES_single ( b, x, NumIters, m, ... )
 1      x_o = 0
        for i = 0, ... , NumIters
           ...
 4         [check convergence and break if done]
 5         ...
16      end
17      x = x_i
```

Table 4: GMRES(m_1)-FGMRES(m_2) with inner GMRES(m_1) in SP (SP data and arithmetic colored in blue) and with outer FGMRES(m_2) in DP. Only the lines that change from the reference GMRES(m) algorithm from Table 2 are given.

derstood [28]. Numerical experiment, as we also show, confirm cases of improvements in speed, robustness, and sometime memory requirements for these methods. For example, we show a maximum speedup of close to 12 times on a problem of size 602,091 (see Section 4). The memory requirement for the method is the sum to store the matrix (e.g. in CSR format with nonzero coefficients in DP), the nonzero matrix coefficients in SP, two $\times$ outer restart size $\times$ the vector size in DP, and inner restart size $\times$ the vector size in SP.

The Generalized Conjugate Residuals (GCR) method [31, 33] is comparable to the FGMRES and can replace it successfully as outer iterative solver.

4 Results Overview

4.1 The Test Collection for Mixed Precision Sparse Direct and Iterative Solvers

We tested our implementation of a mixed precision sparse direct solver on a test suite of 41 matrices taken from the University of Florida's Sparse Matrix Collection [34]. The matrices have been selected randomly from the collection since there is no information available about their condition number. A smaller subset of 9 matrices (described in Table 5) will be discussed in this and the next sessions for readability reasons. The matrices in this smaller subset has been chosen in order to provide examples of all the significant features observed on the test suite. The results for all the 41 matrices in the test suite are listed in appendix A.

	Size	Nonzeroes	Cond. num. est.
G64	7000	82918	$O(10^4)$
Si10H16	17077	875923	$O(10^3)$
c-71	76638	859554	$O(10)$
cage11	39082	559722	$O(1)$
dawson5	51537	1010777	$O(10^4)$
nasasrb	54870	2677324	$O(10^7)$
poisson3Db	85623	2374949	$O(10^3)$
rma10	46835	2374001	$O(10)$
wang4	26068	177196	$O(10^3)$

Table 5: Properties of a subset of the tested matrices. The condition number estimates have been computed on the Opteron 246 architecture by means of MUMPS subroutines.

For the iterative sparse methods we used matrices from an adaptive 3D PDEs discretization package. Presented are results on a set of 5 matrices of increasing size, coming from the adaptive discretization of a 3D linear elasticity problem on a tetrahedral mesh, using piecewise linear elements (see Table 6).

Level	Size	Nonzeroes	Cond. num. est.
1	11,142	442,225	$O(10^3)$
2	25,980	1,061,542	$O(10^4)$
3	79,275	3,374,736	$O(10^4)$
4	230,793	9,991,028	$O(10^5)$
5	602,091	26,411,323	$O(10^5)$

Table 6: Properties of the matrices used with the iterative sparse solvers.

4.2 Performance Characteristics of the Tested Hardware Platforms

The implementation of the mixed precision algorithm for sparse direct methods presented in Section 3.1 has been tested on the architectures reported, along with their main characteristics, in Table 7. All of these architectures have vector units except the Sun UltraSparc-IIe one; this architecture has been included with the purpose of showing that even in the case where the same number of single and double precision operations can be completed in one clock cycle, significant benefits can still be achieved thanks to the reduced memory traffic and higher cache hit rate provided by single precision arithmetic.

The implementation of the mixed precision algorithms for sparse iterative solvers described in Section 3.2 has been only tested on the Intel Woodcrest architecture.

Table 8 shows the difference in performance between the single and double precision implementation for the two dense BLAS operations matrix-matrix product (_GEMM) and matrix-vector product (_GEMV). They are the two principal computational kernels of sparse direct solvers: sparse data structures get rearranged to fit the storage requirements of these kernels and thus benefit from their high performance rates (as opposed to the performance of direct operation on sparse data structures). In particular, column 3 (5) reports the ratio between the performance of SGEMM (SGEMV) and DGEMM (DGEMV). The BLAS libraries used are capable of exploiting the vector units where available and, thus, the speedups shown in Table 8 are due to a combination of higher number of floating point operations completed at each clock cycle, reduced memory traffic on the bus and higher cache hit rate.

Table 9 shows the difference in performance for the single and double precision implementation of the two sparse iterative solvers Conjugate Gradient and Generalized Minimum Residual. Columns 2 and 3 report the ratio between the performance of single and double precision CG for a fixed number (100) of iterations in both preconditioned and unpreconditioned cases. Columns 4 and 5 report the same information for the GMRES(20) method where the number of cycles has been fixed to 2. Since the sparse matrix kernels involved in these computations have not been vectorized, the speedup shown in Table 9 is exclusively due to reduced data traffic on the bus and higher cache hit rate.

	Clock freq.	Vector Units	Memory	Compiler	Compiler flags	BLAS
AMD Opteron 246	2 GHz	SSE, SSE2 3DNOW!	2 GB	Intel v9.1	-O3	Goto
Sun UltraSparc-IIe	502 MHz	none	512 MB	Sun v9.0	-fast -xchip=ultra2e -xarch=v8plusa	Sunperf
Intel PIII Copp.	900 MHz	SSE, MMX	512 MB	Intel v9.0	-O3	Goto
PowerPC 970	2.5 GHz	AltiVec	2 GB	IBM v8.1	-O3 -qalign=4k	Goto
Intel Woodcrest	3 GHz	SSE, SSE2 MMX	4 GB	Intel v9.1	-O3	Goto
Intel XEON	2.4 GHz	SSE, SSE2 MMX	2 GB	Intel v8.0	-O3	Goto
Intel Centrino Duo	2.5 GHz	SSE, SSE2 MMX	4 GB	Intel v9.0	-O3	Goto

Table 7: Characteristics of the architectures used to measure the experimental results.

	Size	SGEMM/ DGEMM	Size	SGEMV/ DGEMV
AMD Opteron 246	3000	2.00	5000	1.70
Sun UltraSparc-IIe	3000	1.64	5000	1.66
Intel PIII Copp.	3000	2.03	5000	2.09
PowerPC 970	3000	2.04	5000	1.44
Intel Woodcrest	3000	1.81	5000	2.18
Intel XEON	3000	2.04	5000	1.82
Intel Centrino Duo	3000	2.71	5000	2.21

Table 8: Performance comparison between single and double precision arithmetics for matrix-matrix and matrix-vector product operations.

	SCG/DCG		SGMRES/DGMRES	
Size	no prec.	prec.	no prec.	prec.
11,142	2.24	2.11	2.04	1.98
25,980	1.49	1.50	1.52	1.51
79,275	1.57	1.50	1.58	1.50
230,793	1.73	1.72	1.74	1.69
602,091	1.50	1.50	1.67	1.63

Table 9: Performance comparison between single and double precision arithmetics on a fixed number of iterations of Conjugate Gradient (100 iterations) and Generalized Minimal RESidual (2 cycles of GMRES(20)) methods both with and without diagonal scaling preconditioner. The runs were performed on Intel Woodcrest (3GHz on a 1333MHz bus).

4.3 Experimental results

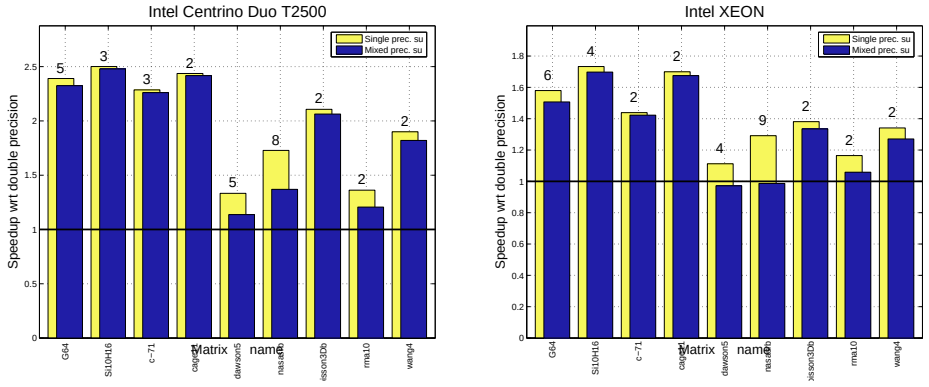

Figure 1: Experimental result measured on the Intel Centrino Duo (using a single processor) (*left*) and the Intel XEON (*right*) architectures. The yellow bars reports the ratio between the performance of the single precision solver and the double precision one; the blue bars report the ratio between the mixed precision solver and the double precision one. The number of iterations required to converge is given by the number above the bars.

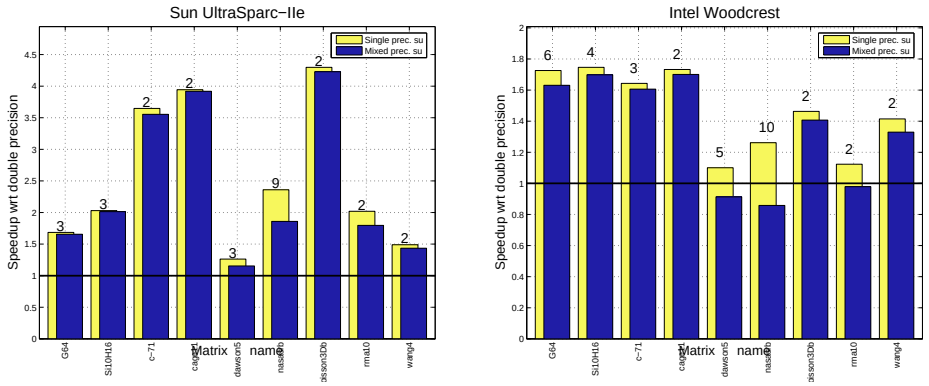

Figure 2: Experimental result measured on the Sun UltraSparc-IIe (*left*) and the Intel Woodcrest (*right*) architectures. The yellow bars reports the ratio between the performance of the single precision solver and the double precision one; the blue bars report the ratio between the mixed precision solver and the double precision one. The number of iterations required to converge is given by the number above the bars.

Figures 1 to 4 show that the single precision solver is always faster than the double precision solver (i.e. the yellow bars are always above the thick horizontal line that corresponds to 1). This is mainly due to both reduced data movement and better exploitation of vector arithmetics (via SSE2 or AltiVec where present) since multifrontal

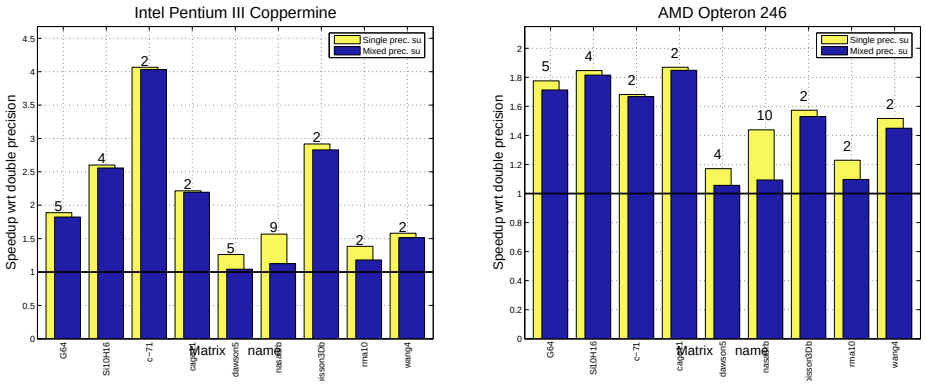

Figure 3: Experimental result measured on the Intel Pentium III Coppermine (*left*) and the AMD Opteron 246 (*right*) architectures. The yellow bars reports the ratio between the performance of the single precision solver and the double precision one; the blue bars report the ratio between the mixed precision solver and the double precision one. The number of iterations required to converge is given by the number above the bars.

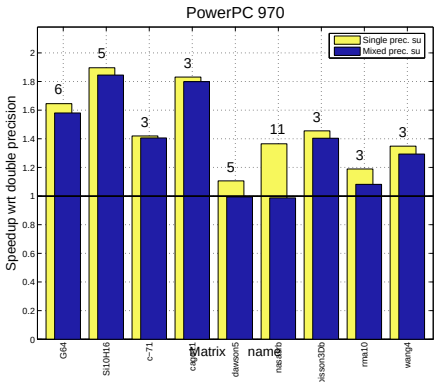

Figure 4: Experimental result measured on the PowerPC 970 architecture. The yellow bars reports the ratio between the performance of the single precision solver and the double precision one; the blue bars report the ratio between the mixed precision solver and the double precision one. The number of iterations required to converge is given by the number above the bars.

methods have the ability to do matrix-matrix products.

The results presented also show that mixed precision iterative refinement is capable of providing considerable speedups with respect to the full double precision solver while providing the same (in many cases also better) accuracy. To run these experiments a convergence criterion different than that discussed in Section 3.1 has been used. To make the comparison fair, in fact, the iterative refinement is stopped whenever the residual norm is the same as that computed for the full double precision solver.

	Reference BLAS			Goto BLAS		
Name	t_{SP}	t_{DP}	t_{DP}/t_{SP}	t_{SP}	t_{DP}	t_{DP}/t_{SP}
benzene	34.290	42.400	123.65%	46.030	40.390	87.75%
sme3Db	11.030	14.200	128.74%	13.470	14.340	106.46%
Kuu	0.160	0.160	100.00%	0.180	0.160	88.89%
airfoil_2d	0.390	0.440	112.82%	0.390	0.440	112.82%
twotone	6.830	8.140	119.18%	7.280	8.600	118.13%
torso2	2.310	2.760	119.48%	2.360	2.810	119.07%
bcsstk39	5.800	6.270	108.10%	7.330	7.350	100.27%
ecl32	69.070	91.360	132.27%	91.350	85.510	93.61%
bbmat	24.470	31.010	126.73%	28.310	29.320	103.57%
raefsky3	5.190	5.880	113.29%	5.360	5.850	109.14%
heart1	3.250	3.920	120.62%	3.730	3.520	94.37%
epb3	1.320	1.560	118.18%	1.410	1.690	119.86%
Zhao1	6.300	8.440	133.97%	7.230	8.230	113.83%
wathen120	1.020	1.270	124.51%	1.140	1.360	119.30%
mult_dcop_01	0.230	0.270	117.39%	0.240	0.260	108.33%
finan512	1.520	1.670	109.87%	1.510	1.720	113.91%
ex40	0.650	0.700	107.69%	0.760	0.840	110.53%
venkat01	2.820	3.390	120.21%	3.420	4.140	121.05%
bundle1	7.190	9.230	128.37%	8.500	8.000	94.12%
wang4	16.720	21.580	129.07%	22.020	20.890	94.87%
Si10H16	303.550	390.960	128.80%	394.840	367.780	93.15%
graham1	0.760	0.890	117.11%	0.780	0.930	119.23%

Table 10: Time to solution of sequential SuperLU in single and double precision for selected sparse matrices on the Intel Woodcrest with reference and optimized BLAS.

The performance of the mixed precision solver is usually very close to that of the single precision mainly because the cost of each iteration is often negligible if compared to the cost of the matrix factorization. It is important to note that, in some cases, the speedups reach very high values (more than 4.0 faster for the Poisson3Db matrix on the Sun UltraSparc-IIe architecture). This is due to the fact that, depending on the matrix size, the fill-in generated in the factorization phase and the available memory on the system, the memory requirements may be too high which forces the virtual memory system to swap pages on disk resulting in a considerable loss of performance; since double precision data is twice as large as single precision, disk swapping may affect only the double precision solver and not the single precision one. It can be noted that disk swapping issues are not affecting the results measured on those machines that are equipped with a bigger memory while it's usual on th Intel Pentium III and the Sun UltraSparc-IIe architectures that only have 512 MB of memory. Finally the data presented in figures 1 to 4 show that for some cases the mixed precision iterative refinement solver does not provide a speedup. This can be mainly associated to three causes (or any combination of them):

1. the difference in performance between the single and the double precision solver

is too small. In this case even a few iterations of the refinement phase will compensate the small speedup. This is, for example, the case of the Dawson5 matrix on the PowerPC 970 architecture.

2. the number of iterations to convergence is too high. The number of iterations to convergence is directly related to the matrix condition number (see [3] for details). The case of the Nasasrb matrix on the PowerPC 970 architecture show that even if the single precision solver is almost 1.4 times faster, the mixed precision solver is slower than the double precision one because of the high number of iterations (11) needed to achieve the same accuracy. If the condition number is too high, the method may not converge at all; this is the cable of the qaf8k matrix (see Appendix A).

3. the cost of each iteration is high as compared to the performance difference between the double precision solver and the single precision one. In this case even a few iterations can eliminate the benefits of performing the system factorization in single precision. As an example take the case of the Rma10 matrix on the Intel Woodcrest architecture; two iteration steps on this matrix take 0.1 seconds which is almost the same time needed to perform 6 iteration steps on the G64 matrix.

It is worth noting that, apart from the cases where the method does not converge, whenever the method results in a slowdown, the loss is on average only 7%. Table 10 shows timings of sequential version of SuperLU on selected matrices from our test collection for single and double precision solvers. Both reference and Goto BLAS timings are shown. The sequential version of SuperLU calls matrix-vector multiply (_GEMV) as its computational kernel. This explains rather modest gains (if any) in the performance of single precision solver over the double precision one: only up to 30%. The table also reveals that when optimized BLAS are used, the single precision is slower than double for some matrices: an artifact of small sizes of dense matrices passed to BLAS and the level of optimization of the BLAS for this particular architecture. The results are similar for other tested architectures which leads to a conclusion that there is not enough benefit of using our mixed precision approach for this version of SuperLU.

Finally, we present our results on the mixed precision iterative sparse solvers from Subsection 3.2. All the results are from runs on Intel Woodcrest (3GHz on a 1333MHz bus).

In Figure 5 we give the speedups in using mixed SP-DP *vs* DP-DP CG (in blue). Namely, on the left we have the results for CG-PCG and on the right for PCG-PCG with diagonal preconditioner in the inner loop PCG. Similarly, in Figure 6, we give the results for GMRES-FGMRES (on the left), and PGMRES-FGMRES (on the right). Also, we give a comparison with the speedups of using SP *vs* DP for just the reference CG and PCG (correspondingly left and right in Figure 5; in yellow), and SP *vs* DP for the reference GMRES and PGMRES (in Figure 6 in yellow). Note that in a sense the speedups in yellow should represent the maximum that could be achieved by using the mixed precision algorithms. The fact that we get close to this maximum performance shows that we have successfully shifted the load from DP to SP arithmetic (with overall computation having less than 5% in DP arithmetic). The reason that the performance

speedup for SP-DP *vs* DP-DP GMRES-FGMRES in Figure 6, left, 4th matrix is higher than the speedup of SP GMRES *vs* DP GMRES is that the SP-DP GMRES-FGMRES did one less outer cycle until convergence than the DP-DP GMRES-FGMRES.

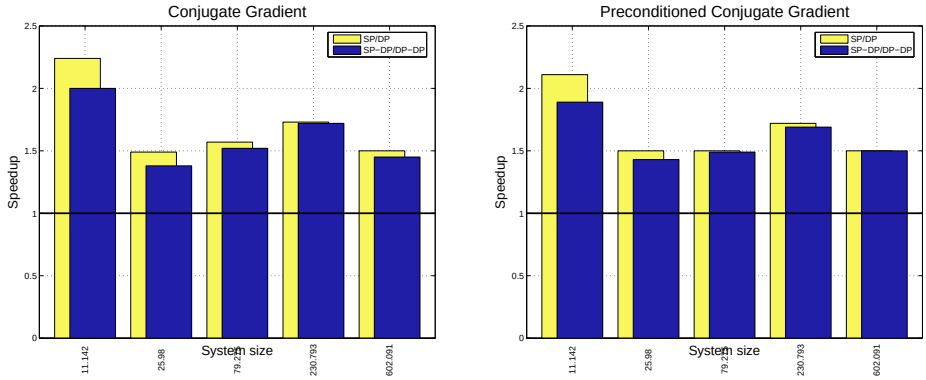

Figure 5: **Left**: Speedup of using SP *vs* DP CG (in yellow) and the SP-DP *vs* DP-DP CG-PCG (in blue). **Right**: Similar graph comparison but for the PCG algorithm (see also Subsection 3.2.1. The computations are on a Intel Woodcrest (3GHz on a 1333MHz bus).

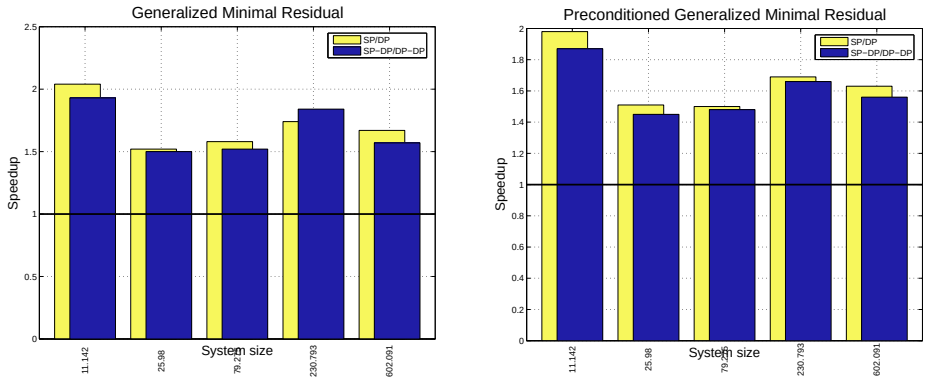

Figure 6: **Left**: Speedup of using SP *vs* DP GMRES (in yellow) and the SP-DP *vs* DP-DP GMRES-FGMRES (in blue). **Right**: Similar graph comparison but for the PGMRES algorithm (see also Subsection 3.2.2. The computations are on a Intel Woodcrest (3GHz on a 1333MHz bus).

Finally, we show results comparing the SP-DP methods with the reference DP methods. In Figure 7 left is a comparison for CG, and on the right for GMRES. The numbers on top of the bars on the left graph indicate the overhead, as number of iterations, that took the mixed precision method to converge versus the reference DP method (e.g. overhead of 0.1 indicates 10% more iterations were performed in the mixed SP-DP *vs* the DP method). Even with the overhead we see a performance speedup of at

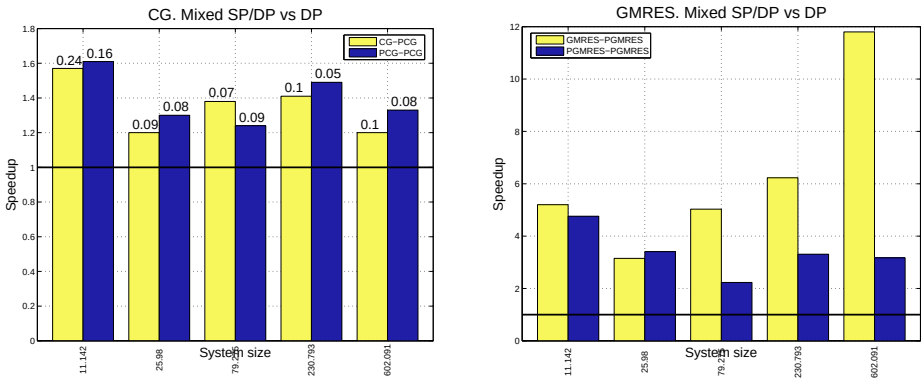

Figure 7: **Left**: Speedup of mixed SP-DP CG-PCG *vs* DP CG (yellow) and SP-DP PCG-PCD *vs* DP CG (blue) with diagonal preconditioner. **Right**: Similar graph comparison but for the GMRES based algorithms. The computations are on a Intel Woodcrest (3GHz on a 1333MHz bus).

least 20% over the tested matrices. For the GMRES-based mixed precision methods we see a significant improvement, based on reduced number of iterations and the effect of the SP speedup (from 45 to 100% as indicated in Figure 6). For example, the speedup factor of 12 for the biggest problem is due to speedup factors of approximately 7.5 from improved convergence and 1.6 from effects associated with the introduced SP storage and arithmetic.

It is not known how to choose the restart size m to get optimal results even for the reference `GMRES(m)`. A reasonable working assumption is the bigger m the better, because one assumes that bigger m will get closer to the full GMRES. Following this assumption though does not guarantee better execution time, and sometimes the convergence can get even worse [35]. An interesting approach is self adaptivity (see [36]). Here, to do a fair comparison, we run it for $m = 25$, 50 (PETSc's default [37]), 100, 150, 200, and 300, and chose the best execution time. Experiments show that the mixed precision method suggested is stable in regard to changing the restart values in the inner and outer loops. The experiments presented are for inner and outer $m = 20$. Note that this choice also results in less memory requirements than GMRES with $m \approx 70$ and higher (for most of the runs GMRES(100) was best among the above choices for m), since the overhead in terms of DP vectors is $20+20$ (outer GMRES) $+10$ (20 SP vectors in the inner loop) $+20$ (matrix coefficients in SP; there are approximately 40 non-zeroes per row; see Table 6). In all the cases presented we had the number of inner cycles set to one.

Finally, we note on speedup for direct and iterative methods and its effect on performance. The speed up of moving to single precision for _GEMM-calling code (MUMPS) was approaching 2 and thus guaranteed success of our mixed-precision iterative refinement just as it did for the dense matrix operations. Not so for the _GEMV-calling code (SuperLU) for which the speedup did not exceed 30% and thus no performance improvement was expected. However, for most of the iterative methods, the speedup

was around 50% and still we claim our approach to be successful. Inherently the reason for speedup is the same for both settings (SuperLU and the iterative methods): the reduced memory bus traffic and possible super-linear effects when data fits in cache while being stored in single precision. But for the SuperLU case there is the direct method overhead: the maintenance of evolving sparse data structures which is done in fixed-point arithmetic so it does not benefit from using single precision floating-point arithmetic and hence yields the overall performance gains insufficient for our iterative refinement approach.

5 Future Work

We are considering a number of extensions and new directions for our work. The most broad category is the parallel setting. MUMPS is a parallel code but it was used in a sequential setting in this study. Similarly, SuperLU has a parallel version which differs from the sequential counterpart in a very important way: it uses the matrix-matrix multiply kernel (_GEMM). This would give a better context for comparing multifrontal and supernodal approaches as they use the same underlying computational library. The only caveat is the lack of a single precision version of the parallel SuperLU solver. Another aspect brought by the latter solver is using static pivoting – while it vastly improves numerical stability of parallel SuperLU it also improves the convergence of the iterative refinement that follows. This should result in less iterations and shorter solve time.

Using PETSc and its parallel framework for (among others) iterative methods could give us opportunity to investigate our approach for a wider range of iterative methods and preconditioning scenarios. First though, we would have to overcome a technical obstacle of combining a two versions of PETSc (one using single and one using double precision) in a single executable.

We have done preliminary experiments on an actual IBM Cell BE hardware (as opposed to the simulator which does not account accurately for memory system effects – a crucial component of sparse methods) with sparse matrix operations and are encouraged by the results to port our techniques in full. This would allow us to study their behavior with much larger gap in the performance of the two precisions.

Our algorithms and their above descriptions focus solely on two precisions: single and double. We see it however in a more broader context of higher and lower precision where, for example, a GPU performs computationally intensive operations in its native 16-bit arithmetic and consequently the solution is refined using 128-bit arithmetic emulated in software (if necessary). As mentioned before, the limit factor is conditioning of the system matrix. In fact, an estimate (up to the order of magnitude) of the condition number (often available from previous runs or the physical problem properties) may become an input parameter to an adaptive algorithm [38] that attempts to utilize the fastest hardware available if its limited precision can guarantee convergence.

Also, the methods for sparse eigenvalue problems that result in Lanczos and Arnoldi algorithms are amenable to our techniques and we would like to study their theoretical and practical challenges.

References

[1] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[2] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[3] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In *Proceedings of SC06*, Tampa, Florida, Nomveber 11-17 2006. See http://icl.cs.utk.edu/iter-ref.

[4] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, September 1983.

[5] Cleve Ashcraft, R. Grimes, J. Lewis, Barry W. Peyton, and Horst Simon. Progress in sparse matrix methods in large sparse linear systems on vector supercomputers. *Intern. J. of Supercomputer Applications*, 1:10–30, 1987.

[6] Patrick R. Amestoy, Iain S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[7] Patrick R. Amestoy, Iain S. Duff, J.-Y. L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23:15–41, 2001.

[8] Patrick R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput.*, 32:136–156, 2006.

[9] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29:110–140, 2003.

[10] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.

[11] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. A asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(3):915–952, 1999.

[12] Xiaoye S. Li. *Sparse Gaussian Elimination on High Performance Computers*. Computer Science Department, University of California at Berkeley, 1996. Ph.D. thesis (SuperLU software available at http://www.nersc.gov/~xiaoye/SuperLU/).

[13] Timothy A. Davis Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 18:140–158, 1997.

[14] Timothy A. Davis. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25:1–19, 1999.

[15] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30:196–199, 2004.

[16] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, Jack Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics., 1994. Also available as postscript file at http://www.netlib.org/templates/Templates.html.

[17] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[18] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual method for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, pages 856–869, 1986.

[19] G. W. Stewart. *Matrix algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[20] James Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2):325–351, 2006.

[21] R. Strzodka and D. Göddeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *IEEE Proceedings on Field–Programmable Custom Computing Machines (FCCM 2006)*. IEEE Computer Society Press, May 2006. to appear.

[22] R. Strzodka and D. Göddeke. Mixed precision methods for convergent iterative schemes, May 2006. EDGE 2006, 23.-24. May 2006, Chapel Hill, North Carolina.

[23] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *Simulationstechnique 18th Symposium in Erlangen, September 2005*, volume Frontiers in Simulation, pages 139–144. SCS Publishing House e.V., 2005. ASIM 2005.

[24] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Latency, bandwidth, and concurrent issue limitations in high-performance cfd. Technical Report ANL/MCS-P850-1000, Argonne National Laboratory, October 2000.

[25] William D. Gropp, Dinesh K. Kaushik, David E. Keyes, and Barry F. Smith. High-performance parallel implicit CFD. *Parallel Computing*, 27(4):337–362, 2001.

[26] Gene H. Golub and Qiang Ye. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing*, 21(4):1305–1320, 2000.

[27] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. Technical Report 91-279, Departmen of Computer Science and Egineering, University of Minnesota, Minneapolis, Minnesota, 1991.

[28] Valeria Simoncini and Daniel B. Szyld. Flexible inner-outer krylov subspace methods. *SIAM J. Numer. Anal.*, 40(6):2219–2239, 2002.

[29] O. Axelsson and P. S. Vassilevski. A black box generalized conjugate gradient solver with inner iterations and variable-step preconditioning. *SIAM J. Matrix Anal. Appl.*, 12(4):625–644, 1991.

[30] Y. Notay. Flexible conjugate gradient, 2000.

[31] C. Vuik. New insights in gmres-like methods with variable preconditioners. *J. Comput. Appl. Math.*, 61(2):189–204, 1995.

[32] J. van den Eshof, G. L. G. Sleijpen, and M .B. van Gijzen. Relaxation strategies for nested Krylov methods. Technical Report TR/PA/03/27, CERFACS, Toulouse, France, 2003.

[33] H. A. van der Vorst and C. Vuik. GMRESR: A family of nested GMRES methods. Technical Report DUT-TWI-91-80, Delft, The Netherlands, 1991.

[34] University of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[35] Mark Embree. The tortoise and the hare restart gmres. *SIAM Review*, 45:259–266, 2003.

[36] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Kathy Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), February 2005. See http://www.spiral.net/ieee-special-issue/overview.html.

[37] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[38] Jack J. Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. Technical Report Lapack Working Note 157, ICL-UT-02-07, Innovative Computing Lab, University of Tennessee, August 2002. To appear IJHPCA 17(2) 2003, http://icl.cs.utk.edu/iclprojects/pages/sans.html.

A Detailed Results

	Intel Core Duo	PowerPC 970	Intel XEON	Intel PentiumIII Coppermine	AMD Opteron 246	Intel Wood-crest	Sun Ultrasparc IIe
Gset/G64	2.39 2.32 (5)	1.65 1.58 (6)	1.58 1.51 (6)	1.89 1.82 (5)	1.78 1.71 (5)	1.73 1.63 (6)	1.69 1.66 (3)
MathWorks/Kuu	1.23 0.89 (3)	1.12 0.96 (3)	1.11 0.92 (3)	1.26 0.93 (3)	1.18 1.02 (2)	1.12 0.94 (2)	1.26 0.99 (3)
PARSEC/Si10H16	2.50 2.48 (3)	1.90 1.85 (5)	1.73 1.70 (4)	2.60 2.56 (4)	1.85 1.82 (4)	1.75 1.70 (4)	2.03 2.01 (3)
Zhao/Zhao1	1.45 1.34 (2)	1.14 1.08 (3)	1.13 1.03 (2)	1.33 1.24 (2)	1.26 1.19 (2)	1.15 1.05 (2)	2.78 2.61 (2)
Engwirda/airfoil 2d	1.24 1.11 (2)	1.11 1.05 (3)	1.11 1.03 (2)	1.25 1.13 (2)	1.15 1.08 (2)	1.08 0.99 (2)	1.25 1.15 (2)
Simon/bbmat	2.01 1.91 (5)	1.50 1.40 (6)	1.43 1.34 (5)	1.67 1.51 (7)	1.57 1.48 (5)	1.47 1.34 (5)	2.15 2.04 (5)
Boeing/bcsstk39	1.73 1.42 (5)	1.40 1.11 (6)	1.28 1.05 (5)	1.58 1.22 (5)	1.47 1.20 (5)	1.26 0.96 (5)	1.56 1.33 (4)
PARSEC/benzene	2.32 2.29 (2)	1.70 1.67 (3)	1.57 1.54 (2)	1.97 1.94 (2)	1.80 1.78 (2)	1.69 1.65 (2)	1.62 1.60 (2)
GHS indef/blockqp1	1.02 0.80 (3)	1.05 0.86 (4)	1.06 0.84 (3)	1.06 0.80 (3)	1.17 0.97 (3)	1.10 0.96 (3)	1.04 0.83 (3)
Lourakis/bundle1	1.37 1.20 (2)	1.19 1.03 (4)	1.15 1.00 (3)	1.31 1.06 (3)	1.22 1.08 (3)	1.14 1.00 (2)	1.28 1.14 (2)
GHS indef/c-71	2.29 2.26 (3)	1.42 1.41 (3)	1.44 1.42 (2)	4.07 4.03 (2)	1.68 1.67 (2)	1.64 1.61 (3)	3.65 3.55 (2)
vanHeukelum/cage11	2.44 2.42 (2)	1.83 1.80 (3)	1.70 1.68 (2)	2.22 2.20 (2)	1.87 1.85 (2)	1.73 1.70 (2)	3.94 3.92 (2)
DRIVCAV/cavity26	1.19 0.78 (2)	1.94 1.63 (3)	1.11 0.96 (2)	1.27 0.99 (2)	1.14 0.97 (2)	1.23 1.03 (2)	2.22 1.81 (2)
Rothberg/cfd1	1.85 1.75 (3)	1.42 1.34 (4)	1.38 1.30 (3)	1.62 1.52 (3)	1.52 1.44 (3)	1.35 1.25 (3)	3.49 3.35 (3)
GHS indef/dawson5	1.33 1.14 (5)	1.11 0.99 (5)	1.11 0.97 (4)	1.26 1.04 (5)	1.17 1.06 (4)	1.10 0.91 (5)	1.26 1.15 (3)
Sanghavi/ecl32	2.02 1.95 (3)	1.31 1.27 (4)	1.40 1.31 (4)	1.68 1.62 (3)	1.56 1.49 (4)	1.46 1.35 (4)	1.85 1.79 (3)
Averous/epb3	1.12 1.01 (2)	1.04 0.96 (4)	1.04 0.94 (2)	1.10 1.00 (2)	1.07 0.96 (3)	1.05 0.91 (3)	1.27 1.13 (3)
FIDAP/ex40	1.53 1.36 (3)	1.31 1.20 (4)	1.25 1.13 (3)	1.48 1.31 (3)	1.29 1.20 (3)	1.18 1.01 (4)	1.42 1.30 (3)
Mulvey/finan512	1.07 0.96	1.03 0.97	1.01 0.90	1.07 0.97	1.04 0.96	1.03 0.91	1.19 1.07

continued on the next page

	Intel Core Duo	PowerPC 970	Intel XEON	Intel PentiumIII Coppermine	AMD Opteron 246	Intel Wood crest	Sun Ultrasparc IIe
	(2)	(3)	(2)	(2)	(2)	(2)	(2)
Graham/graham1	1.33 1.10 (3)	1.14 1.02 (4)	1.12 0.98 (3)	1.27 1.05 (3)	1.11 1.00 (3)	1.11 0.95 (3)	1.18 1.03 (3)
Norris/heart1	1.98 1.78 (3)	1.53 1.43 (3)	1.38 1.30 (2)	1.66 1.49 (2)	1.55 1.46 (2)	1.43 1.28 (2)	1.90 1.76 (2)
kivap001	2.48 2.46 (2)	1.42 1.32 (3)	1.57 1.54 (2)	3.87 3.29 (2)	1.82 1.79 (2)	1.62 1.58 (2)	3.07 1.85 (2)
kivap004	2.00 1.95 (2)	1.47 1.42 (3)	1.38 1.31 (3)	1.70 1.62 (3)	1.56 1.52 (2)	1.43 1.37 (2)	3.05 2.98 (2)
kivap005	1.83 1.73 (3)	1.39 1.30 (4)	1.30 1.22 (3)	1.60 1.49 (3)	1.47 1.40 (3)	1.35 1.24 (3)	1.52 1.44 (3)
kivap006	2.00 1.93 (3)	1.47 1.41 (3)	1.38 1.31 (3)	1.67 1.58 (3)	1.57 1.53 (2)	1.43 1.36 (2)	1.54 1.50 (2)
kivap007	2.14 2.10 (2)	1.57 1.51 (3)	1.46 1.42 (2)	1.79 1.74 (2)	1.66 1.62 (2)	1.51 1.45 (2)	2.91 2.86 (2)
Sandia/mult dcop 01	1.04 1.02 (2)	1.01 1.00 (2)	1.00 0.97 (2)	1.01 1.00 (2)	1.03 1.01 (2)	1.02 1.00 (2)	1.02 1.01 (2)
Nasa/nasa4704	1.38 0.81 (6)	1.31 0.84 (7)	1.13 0.76 (7)	1.42 0.68 (11)	1.29 0.83 (8)	1.20 0.82 (6)	1.47 1.00 (6)
Nasa/nasasrb	1.73 1.37 (8)	1.37 0.99 (11)	1.29 0.99 (9)	1.57 1.13 (9)	1.44 1.09 (10)	1.26 0.86 (10)	2.36 1.86 (9)
Nemeth/nemeth01	1.09 0.99 (2)	1.04 0.98 (3)	1.04 0.97 (2)	1.12 1.00 (2)	1.06 1.01 (2)	1.03 0.96 (2)	1.11 1.02 (2)
Nemeth/nemeth26	1.14 1.07 (2)	1.07 1.01 (3)	1.04 0.99 (2)	1.14 1.04 (2)	1.08 1.04 (2)	1.04 0.98 (2)	1.04 0.99 (2)
FEMLAB/poisson3Db	2.11 2.06 (2)	1.46 1.40 (3)	1.38 1.34 (2)	2.92 2.83 (2)	1.57 1.53 (2)	1.46 1.41 (2)	4.30 4.23 (2)
Cunningham/qa8fk	2.15 1.77 (20) *	1.60 1.17 (21) *	1.48 1.11 (20) *	2.34 1.79 (20) *	1.66 1.29 (20) *	1.51 1.06 (20) *	3.80 3.35 (20) *
Simon/raefsky3	1.91 1.75 (2)	1.47 1.34 (3)	1.39 1.29 (2)	1.69 1.51 (2)	1.57 1.44 (2)	1.38 1.23 (2)	2.46 2.28 (2)
Bova/rma10	1.36 1.21 (2)	1.19 1.08 (3)	1.16 1.06 (2)	1.38 1.18 (2)	1.23 1.10 (2)	1.12 0.98 (2)	2.02 1.80 (2)
FEMLAB/sme3Db	1.79 1.60 (4)	1.37 1.14 (7)	1.33 1.16 (5)	1.58 1.29 (5)	1.46 1.22 (6)	1.32 1.05 (6)	1.86 1.47 (6)
Norris/torso2	1.18 1.07 (2)	1.06 1.00 (3)	1.07 0.97 (2)	1.16 1.06 (2)	1.11 1.02 (2)	1.06 0.95 (2)	1.13 1.04 (2)
ATandT/twotone	2.36 2.35 (0) *	1.52 1.52 (1) *	1.28 1.28 (0) *	1.51 1.51 (0) *	1.71 1.71 (0) *	1.61 1.61 (0) *	2.69 2.69 (0) *
	1.52	1.26	1.23	1.46	1.34	1.19	1.79

continued on the next page

	Intel Core Duo	PowerPC 970	Intel XEON	Intel PentiumIII Coppermine	AMD Opteron 246	Intel Wood crest	Sun Ultrasparc IIe
Simon/venkat01	1.34 (2)	1.12 (3)	1.10 (2)	1.26 (2)	1.18 (2)	1.03 (2)	1.60 (2)
Wang/wang4	1.90 1.82 (2)	1.35 1.29 (3)	1.34 1.27 (2)	1.58 1.51 (2)	1.52 1.45 (2)	1.41 1.33 (2)	1.49 1.44 (2)
GHS psdef/wathen120	1.19 1.04 (3)	1.06 1.00 (3)	1.08 0.99 (2)	1.16 1.05 (2)	1.12 1.03 (2)	1.07 0.98 (2)	1.13 1.05 (2)

	Size	Nonzeroes	Cond. Num. Est.
Gset/G64	7000	82918	$O(10^4)$
MathWorks/Kuu	7102	340200	$O(10^4)$
PARSEC/Si10H16	17077	875923	$O(10^3)$
Zhao/Zhao1	33861	166453	$O(1)$
Engwirda/airfoil_2d	14214	259688	$O(10^3)$
Simon/bbmat	38744	1771722	$O(10^3)$
Boeing/bcsstk39	46772	2089294	$O(10^6)$
PARSEC/benzene	8219	242669	$O(10^3)$
GHS_indef/blockqp1	60012	640033	$O(1)$
Lourakis/bundle1	10581	770901	$O(10)$
GHS_indef/c-71	76638	859554	$O(10)$
vanHeukelum/cage11	39082	559722	$O(1)$
DRIVCAV/cavity26	4562	138187	$O(10^3)$
Rothberg/cfd1	70656	1828364	$O(10^5)$
GHS_indef/dawson5	51537	1010777	$O(10^4)$
Sanghavi/ecl32	51993	380415	$O(10^5)$
Averous/epb3	84617	463625	$O(10^4)$
FIDAP/ex40	7740	458012	$O(10^2)$
Mulvey/finan512	74752	596992	$O(1)$
Graham/graham1	9035	335504	$O(10^2)$
Norris/heart1	3557	1387773	$O(10^3)$
kivap001	86304	1575568	$O(10^2)$
kivap004	42204	755416	$O(10^3)$
kivap005	25054	436468	$O(10^4)$
kivap006	42204	755416	$O(10^3)$
kivap007	56904	1028800	$O(10^3)$
Sandia/mult_dcop_01	25187	193276	$O(10)$
Nasa/nasa4704	4704	104756	$O(10^6)$
Nasa/nasasrb	54870	2677324	$O(10^7)$
Nemeth/nemeth01	9506	725054	$O(10)$
Nemeth/nemeth26	9506	1511760	$O(1)$
FEMLAB/poisson3Db	85623	2374949	$O(10^3)$

continued on the next page

	Size	Nonzeroes	Cond. Num. Est.
Cunningham/qa8fk	66127	1660579	$O(10^{16})$
Simon/raefsky3	21200	1488768	$O(10^{2})$
Bova/rma10	46835	2374001	$O(10)$
FEMLAB/sme3Db	29067	2081063	$O(10^{6})$
Norris/torso2	115967	1033473	$O(1)$
ATandT/twotone	120750	1224224	$O(10^{2})$
Simon/venkat01	62424	1717792	$O(10)$
Wang/wang4	26068	177196	$O(10^{3})$
GHS_psdef/wathen120	36441	565761	$O(1)$