Exploiting Mixed Precision Floating Point Hardware in Scientific Computations

Alfredo BUTTARI [a] Jack DONGARRA [a;b;d] Jakub KURZAK [a] Julie LANGOU [a]
Julien LANGOU [c] Piotr LUSZCZEK [a] and Stanimire TOMOV [a]
[a] *Department of Computer Science, University of Tennessee Knoxville*
[b] *Oak Ridge National Laboratory*
[c] *University of Colorado at Denver and Health Sciences Center*
[d] *University of Manchester*

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

Keywords. Iterative refinement, factorization, Krylov methods

Introduction

In numerical computing, there is a fundamental performance advantage in using the single precision, floating point data format over the double precision one. Due to more compact representation, twice the number of single precision data elements can be stored at each level of the memory hierarchy including the register file, the set of caches, and the main memory. By the same token, handling single precision values consumes less bandwidth between different memory levels and decreases the number of cache and TLB misses. However, the data movement aspect affects mostly memory-intensive, bandwidth-bound applications, historically have not drawn much attention to mixed precision algorithms.

In the past, the situation looked differently for computationally intensive workloads, where the load was on the floating point processing units rather than the memory subsystem, and so the single precision data motion advantages were for the most part irrelevant. With the focus on double precision in the scientific computing, double precision execution units were fully pipelined and capable of completing at least one operation per clock cycle. In fact, in many high performance processor designs single precision units were eliminated in favor of emulating single precision operations using double precision circuitry. At the same time, a high degree of instruction level parallelism was being achieved by introducing more functional units and relatively complex speculation mechanisms, which did not necessarily guarantee full utilization of the hardware resources.

That situation began to change with the widespread adoption of short vector, Single Instruction Multiple Data (SIMD) processor extensions, which started appearing in the mid 90s. An example of such extensions are the Intel MultiMedia eXtensions (MMX) that were mostly meant to improve processor performance in Digital Signal Processing (DSP) applications, graphics and computer games. Short vector, SIMD instructions are a relatively cheap way of exploiting data level parallelism by applying the same operation to a vector of elements at once. It eliminates the hardware design complexity associated with the bookkeeping involved in speculative execution. It also gives better guarantees for practically achievable performance than does runtime speculation, provided that enough data parallelism exists in the computation. Most importantly, short vector, SIMD processing provides the opportunity to benefit from replacing the double precision arithmetic with the single precision one. Since the goal is to process the entire vector in a single operation, the computation throughput doubles while the data storage space halves.

Most processor architectures available today have been augmented, at some point, in their design evolution with short vector, SIMD extensions. Examples include Streaming SIMD Extensions (SSE) for the AMD and the Intel line of processors; PowerPC's Velocity Engine, AltiVec, and VMX; SPARC's Visual Instruction Set (VIS); Alpha's Motion Video Instructions (MVI); PA-RISC's Multimedia Acceleration eXtensions (MAX); MIPS-3D Application Specific Extensions (ASP) and Digital Media Extensions (MDMX) and ARM's NEON feature. The different architectures exhibit large differences in their capabilities. The vector size is either 64 bits or, more commonly, 128 bits. The register file size ranges from just a few to as many as 256 registers. Some extensions only support integer types while others operate on single precision, floating point numbers, and yet others process double precision values.

Today, the Synergistic Processing Element (SPE) of the CELL processor can probably be considered the state of the art in short vector, SIMD processing. Possessing 128-byte long registers and a fully pipelined fused, multiply-add instruction, it is capable of completing as many as eight single precision, floating point operations each clock cycle. When combined with the size of the register file of 128 registers, it is capable of delivering close to peak performance on many common computationally intensive workloads.

Table 1 shows the difference in peak performance between single precision (SP) and double precision (DP) of four modern processor architectures. Following the recent trend in chip design, all of the presented processors are multi-core architectures. However, to avoid introducing the complexity of thread-level parallelization to the discussion, we will mainly look at the performance of individual cores throughout the chapter. The goal here is to focus on instruction-level parallelism exploited by short vector SIMD'zation.

Although short vector, SIMD processors have been around for over a decade, the concept of using those extensions to utilize the advantages of single precision performance in scientific computing did not come to fruition until recently, due to the fact that most scientific computing problems require double precision accuracy. It turns out, however, that for many problems in numerical computing, it is possible to exploit the speed of single precision operations and resort to double precision calculations at few stages

[1] The DP unit is not fully pipelined, and has a 7 cycle latency.

Table 1. Floating point performance characteristics of *individual cores* of modern, multi-core processor architectures.

Architecture	**Clock**	**DP Peak**	**SP Peak**
	[GHz]	**[Gflop/s]**	**[Gflop/s]**
AMD Opteron 246	2.0	4	8
IBM PowerPC 970	2.5	10	20
Intel Xeon 5100	3.0	12	24
STI Cell BE	3.2	1.8[1]	25.6

of the algorithm to achieve full double precision accuracy of the result. The techniques described here are fairly general and can be applied to a wide range of problems in linear algebra, such as solving linear systems of equations, least square problems, singular value and eigenvalue problems. Here we are going to focus on solving linear systems of equations, both dense and sparse, non-symmetric and symmetric, using direct methods, as well as iterative, Krylov subspace methods.

In this paper, we are going to focus on solving linear systems of equations, non-symmetric and symmetric, dense (Section 1) and sparse, using direct methods (Section 2), as well as iterative, Krylov subspace methods (Section 3).

1. Direct Methods for Solving Dense Systems

1.1. Algorithm

Iterative refinement is a well known method for improving the solution of a linear system of equations of the form Ax D b [1]. The standard approach to the solution of dense linear systems is to use the LU factorization by means of Gaussian elimination. First, the coefficient matrix A is factorized into the product of a lower triangular matrix L and an upper triangular matrix U using LU decomposition. Commonly, partial row pivoting is used to improve numerical stability resulting in the factorization PA D LU, where P is the row permutation matrix. The solution for the system is obtained by first solving Ly D Pb (*forward substitution*) and then solving Ux D y (*back substitution*). Due to the round-off error, the computed solution x carries a numerical error magnified by the condition number of the coefficient matrix A. In order to improve the computed solution, an iterative refinement process is applied, which produces a correction to the computed solution at each iteration, which then yields the basic iterative refinement algorithm (Algorithm 1). As Demmel points out [2], the non-linearity of the round-off error makes the iterative refinement process equivalent to the Newton's method applied to the function $f.x/$ D $b \quad Ax$. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct to the working precision. Iterative refinement is a fairly well understood concept and was analyzed by Wilkinson [3], Moler [4] and Stewart [1].

The algorithm can be modified to use a mixed precision approach. The factorization PA D LU and the solution of the triangular systems Ly D Pb and Ux D y are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients. The most computationally expensive operations, including the factorization of the coefficient matrix A and the forward and backward substitution, are performed using single precision arithmetic and take advantage of its higher speed. The only opera-

Algorithm 1 The iterative refinement method for the solution of linear systems

```
$x_0 \quad A^{\;1}b$
$k$ D 1
repeat
    $r_k \quad b \quad Ax_{k\;1}$
    $z_k \quad A^{\;1}r_k$
    $x_k \quad x_{k\;1}$ C $z_k$
    $k \quad k$ C 1
until convergence
```

tions that must be executed in double precision are the residual calculation and the update of the solution. It can be observed, that all operations of $O.n^3/$ computational complexity are handled in single precision, and all operations performed in double precision are of at most $O.n^2/$ complexity. The coefficient matrix A is converted to single precision for the LU factorization and the resulting factors are also stored in single precision. At the same time, the original matrix in double precision must be preserved for the residual calculation. The mixed precision, iterative refinement algorithm is outlined in Algorithm 2; the $.32/$ subscript means that the data is stored in 32-bit format (i.e., single precision) and the absence of any subscript means that the data is stored in 64-bit format (i.e., double precision). Implementation of the algorithm is provided in the LAPACK package by the routine DSGESV.

Algorithm 2 Solution of a linear system of equations using mixed precision, iterative refinement. (SGETRF and SGETRS are names of LAPACK routines).

```
$A_{.32/}; b_{.32/} \quad A; b$
$L_{.32/}; U_{.32/}; P_{.32/} \quad$ SGETRF$.A_{.32/}/$
$x^{.1/}_{.32/} \quad$ SGETRS$.L_{.32/}; U_{.32/}; P_{.32/}; b_{.32/}/$
$x^{.1/} \quad x^{.1/}_{.32/}$
$i \quad 0$
repeat
    $i \quad i$ C 1
    $r^{.i/} \quad b \quad Ax^{.i/}$
    $r^{.i/}_{.32/} \quad r^{.i/}$
    $z^{.i/}_{.32/} \quad$ SGETRS$.L_{.32/}; U_{.32/}; P_{.32/}; r^{.i/}_{.32/}/$
    $z^{.i/} \quad z^{.i/}_{.32/}$
    $x^{.i\text{C}1/} \quad x^{.i/}$ C $z^{.i/}$
until $x^{.i/}$ is accurate enough
```

Higham [5] gives error bounds for the single and double precision, iterative refinement algorithm when the entire algorithm is implemented with the same precision (single or double, respectively). He also gives error bounds in single precision arithmetic, with refinement performed in double precision arithmetic [5]. The error analysis in double precision, for our mixed precision algorithm (Algorithm 2), is given by Langou et al. [6].

The same technique can be applied to the case of symmetric, positive definite problems. Here, Cholesky factorization (LAPACK's SPOTRF routine) can be used in place of

LU factorization (SGETRF), and a symmetric back solve routine (SPOTRS) can be used in place of the routine for the general (non-symmetric) case (SGETRS). Also, the matrix-vector product Ax can be implemented by the BLAS' DSYMV routine, or DSYMM for multiple right hand sides, instead of the DGEMV and DGEMM routines for the non-symmetric case. The mixed precision algorithm for the symmetric, positive definite case is presented by Algorithm 2. Implementation of the algorithm is provided in the LAPACK package by the routine DSPOSV.

Algorithm 3 Solution of a symmetric positive definite system of linear equations using mixed precision, iterative refinement. (SPOTRF and SPOTRS are names of LAPACK routines).

```
$A_{.32/}; b_{.32/} \quad A; b$
$L_{.32/}; L^T_{.32/} \quad \text{SPOTRF}.A_{.32/}/$
$x^{.1/}_{.32/} \quad \text{SPOTRS}.L_{.32/}; L^T_{.32/}; b_{.32/}/$
$x^{.1/} \quad x^{.1/}_{.32/}$
$i \quad 0$
repeat
  $i \quad i \text{ C } 1$
  $r^{.i/} \quad b \quad Ax^{.i/}$
  $r^{.i/}_{.32/} \quad r^{.i/}$
  $z^{.i/}_{.32/} \quad \text{SPOTRS}.L_{.32/}; L^T_{.32/}; r^{.i/}_{.32/}/$
  $z^{.i/} \quad z^{.i/}_{.32/}$
  $x^{.i\text{C}1/} \quad x^{.i/} \text{ C } z^{.i/}$
until $x^{.i/}$ is accurate enough
```

1.2. Experimental Results and Discussion

To collect performance results for the Xeon, Opteron and PowePC architectures, the LAPACK iterative refinement routines DSGESV and DSPOSV were used, for the non-symmetric and symmetric cases, respectively. The routines implement classic, blocked versions of the matrix factorizations and rely on the layer of Basic Linear Algebra Subroutines (BLAS) for architecture specific optimizations to deliver performance close to the peak. As mentioned before, in order to simplify the discussion and leave out the aspect of parallelization, we have decided to look at the performance of individual cores on the multi-core architectures.

Figures 1-8 show the performance of the single-core serial implementations of Algorithm 2 and Algorithm 3 on the architectures in Table 2.

These figures show that the mixed precision, iterative refinement method can run very close to the speed of the full single precision solver while delivering the same accuracy as the full double precision one. On the AMD Opteron, Intel Woodcrest and IBM PowerPC architectures (see Figures 1- 6), the mixed precision, iterative solver can provide a speedup of up to 1.8 for the unsymmetric solver and 1.5 for the symmetric one, if the problem size is big enough. For small problem sizes, in fact, the cost of even a few iterative refinement iterations is high compared to the cost of the factorization and thus, the mixed precision, iterative solver is less efficient than the full double precision one.

Table 2. Hardware and software details of the systems used for performance experiments.

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

For the Cell processor (see Figures 7 and 8), parallel implementations of Algorithms 2 and 3 have been produced in order to exploit the full computational power of the processor. Due to the large difference between the single precision and double precision floating point units (see Table 1), the mixed precision solver performs up to 7 and 11 faster than the double precision peak in the unsymmetric and symmetric, positive definite cases respectively. Implementation details for this case can be found in [7, 8].

Figure 1. Performance of mixed precision, iterative refinement for unsymmetric problems on Intel Woodcrest.

Figure 2. Performance of mixed precision, iterative refinement for symmetric, positive definite problems on Intel Woodcrest.

Figure 3. Performance of mixed precision, iterative refinement for unsymmetric problems on AMD Opteron246.

Figure 4. Performance of mixed precision, iterative refinement for symmetric, positive definite problems on AMD Opteron246.

Figure 5. Performance of mixed precision, iterative refinement for unsymmetric problems on IBM PowerPC 970.

Figure 6. Performance of mixed precision, iterative refinement for symmetric, positive definite problems on IBM PowerPC 970.

Figure 7. Performance of mixed precision, iterative refinement for unsymmetric problems on CELL Broadband Engine.

Figure 8. Performance of mixed precision, iterative refinement for symmetric, positive definite problems on CELL Broadband Engine.

2. Direct Methods for Solving Sparse Systems

2.1. Algorithm

The mixed precision, iterative refinement methods apply to sparse operations as well as to dense operations. In fact, even for sparse computations, single precision operations are performed at a higher rate than double precision ones. The reason for this difference is different than in the dense case. As already pointed out, in the dense case the difference is due to the fact that vector units in the processors can exploit a higher level of parallelism in the single precision computations than in the double precision ones.

Sparse computations are very difficult to vectorize due to their nature (mostly because of the very irregular memory access patterns and because of the heavy use of indirect addressing). Even in the case where they can be vectorized, this optimization does not have a significant effect on performance because sparse operations are inherently memory bound, which means that the number crunching phase is much cheaper than the cpu-memory communication phase. Despite all this, single precision computations can be performed at a speed that is up to 2 as fast as in double precision, since the amount of data that is moved through the memory bus is halved. The mixed precision, iterative refinement technique is thus applicable to the solution of sparse linear systems, which is commonly achieved with either direct of iterative methods.

Most sparse direct methods for solving linear systems of equations are variants of either multifrontal [9] or supernodal [10] factorization approaches. Here, we focus only on multifrontal methods. For results on supernodal solvers see [11]. There are a number of freely available packages that implement multifrontal methods. We have chosen for our tests the software package called MUMPS [12–14]. The main reason for selecting this software is that it is implemented in both single and double precision, which is not the case for other freely available multifrontal solvers such as UMFPACK [15–17].

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 fill-in, which provides an estimate of the memory that will be allocated in the following steps. Also, pivoting is performed based on the structure of A C A^T, ignoring numerical values. Only integer operations are performed at this step.
2. Matrix Factorization: in this phase the PA D 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 D Pb and Ux D 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). The cost of the iterative refinement steps is lower than the advantage obtained by performing the system solution in single precision if the number of iterations is limited. The implementation of a mixed precision, iterative refinement method with the MUMPS package can thus be summarized as in algorithm 4.

Algorithm 4 Mixed precision, Iterative Refinement with the MUMPS package

```
system analysis
$LU \quad PA$                    $(\ _s)$
solve $Ly \mathrm{D} Pb$         $(\ _s)$
solve $Ux_0 \mathrm{D} y$        $(\ _s)$
$k \quad 1$
until convergence do:
    $r_k \quad b \quad Ax_{k \ 1}$        $(\ _d)$
    solve $Ly \mathrm{D} Pr_k$    $(\ _s)$
    solve $Uz_k \mathrm{D} y$     $(\ _s)$
    $x_k \quad x_{k \ 1} \mathrm{C} z_k$  $(\ _d)$
    $k \quad k \mathrm{C} 1$
done
```

At the end of each line of the algorithm, we indicate the precision used to perform this operation as either $_s$, for single precision computation, or $_d$, for double precision computation. Based on backward stability analysis, the solution x can be considered as accurate as the double precision one when

$$\mathrm{k}b \quad Ax\mathrm{k}_2 \quad \mathrm{k}x\mathrm{k}_2 \ \mathrm{k}A\mathrm{k}_{fro} \quad \mathrm{p}\overline{n}$$

where $\mathrm{k} \ \mathrm{k}_{fro}$ is the Frobenius norm. The iterative method is stopped when the double precision accuracy is achieved or a maximum number of iterations is reached.

2.2. Experimental Results and Discussion

The method in Algorithm 4 can offer significant improvements for the solution of a sparse linear system in many cases if:

1. the number of iterations is not too high.
2. the cost of each iteration is small as compared to the cost of the system factorization. If the cost of each iteration is too high, then a low number of iterations will result in a performance loss with respect to the full double precision solver. In the sparse case, for a fixed matrix size, both the cost of the system factorization and the cost of the iterative refinement step may substantially vary depending on the number of nonzeroes and the matrix sparsity structure.

The efficiency of the mixed precision, iterative refinement approach on sparse direct solvers is shown in Figures 9, 10 and 11. These figures report the performance ratio between the full single precision and full double precision solvers (light colored bars) and the mixed precision and full-double precision solvers (dark colored bars) for six matrices from real world applications. The number on top of each bar shows how many iterations are performed by the mixed precision, iterative method to achieve double precision accuracy.

The data in Figures 9, 10 and 11 have been measured using the architectures listed in Table 1 (except for the Cell processor) on a number of matrices from real world applications. These matrices are reported in Table 3 and are grouped into symmetric and

Figure 9. Mixed precision, iterative refinement with the MUMPS direct solver on an Intel Woodcrest 3.0 GHz system.

Figure 10. Mixed precision, iterative refinement with the MUMPS direct solver on an AMD Opteron 246 2.0 GHz system.

Figure 11. Mixed precision, iterative refinement with the MUMPS direct solver on an IBM PowerPC 970 2.5 GHz system.

unsymmetric ones because the MUMPS package uses different numerical methods for these two classes of matrices.

Table 3. Test matrices for sparse mixed precision, iterative refinement solution methods.

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

3. Iterative Methods for Solving Sparse Systems

Direct sparse methods suffer from fill-ins and, consequently, high memory requirements as well as extended execution time. There are various reordering techniques designed to minimize the amount of fill-ins. Nevertheless, for problems of increasing size, there is a point where they become prohibitively high and direct sparse methods are no longer feasible. Iterative methods are a remedy, since only a few working vectors and the primary data are required [18, 19].

As an example, let us first consider the iterative refinement itself, described in Algorithm 1 as

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

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

$$AMu \text{ D } b\mathbf{;} \quad x \text{ D } Mu$$

iteratively. Related to the overall efficiency, M needs to be easy to compute, apply, and store. Note that these requirements were addressed in the mixed precision methods above by replacing M (coming from LU factorization of A followed by matrix inversion), with its single precision representation so that arithmetic operations can be performed more efficiently on it. Here, we go two steps further: we consider replacing not only M by an inner loop of incomplete iterative solver performed in single precision arithmetic [20] , but also the outer loop by more sophisticated iterative methods (e.g., Krylov type).

3.1. Mixed Precision, Inner-Outer Iterative Solvers

Note that replacing M by an iterative method leads to *nesting* of two iterative methods. Variations of this type of nesting, also known in the literature as an *inner-outer* iteration, have been studied, both theoretically and computationally [21–27]. The general appeal of these methods is that computational speedup is possible when the inner solver uses an approximation of the original matrix A that is also faster to apply (e.g., in our case, using single precision arithmetic). Moreover, even if no faster matrix-vector product is available, speedup can often be observed due to improved convergence (e.g., see [23], where Simoncini and Szyld explain the possible benefits of FGMRES-GMRES over restarted GMRES).

To illustrate the above concepts, we demonstrate the ideas with a mixed precision, inner-outer iterative solver that is based on the restarted Generalized Minimal RESidual (GMRES) method. Namely, consider Algorithm 5, where for the outer loop we take the flexible GMRES (FGMRES [19, 22]) and for the inner loop the GMRES in single precision arithmetic (denoted by GMRES$_{SP}$). FGMRES, a minor modification to the standard GMRES, is meant to accommodate non-constant preconditioners. Note that in our case, this non-constant preconditioner is GMRES$_{SP}$. The resulting method is denoted by FGMRES(m_{out})-GMRES$_{SP}$(m_{in}) where m_{in} is the restart for the inner loop and m_{out} for the outer FGMRES.

The potential benefits of FGMRES compared to GMRES are becoming better understood [23]. Numerical experiments confirm cases of improvements in speed, robustness, and sometimes memory requirements for these methods. For example, we show a maximum speedup of close to 15 on the selected test problems. The memory requirements for the method are the matrix A in CRS format, the nonzero matrix coefficients in single precision, 2 m_{out} number of vectors in double precision, and m_{in} number of vectors in single precision.

The Generalized Conjugate Residuals (GCR) method [26, 28] is comparable to the FGMRES and can replace it successfully as the outer iterative solver.

3.2. Numerical Performance

Similar to the case of sparse direct solvers, we demonstrate the numerical performance of Algorithm 5 on the architectures from Table 1 and on the matrices from Table 3.

Algorithm 5 Mixed precision, inner-outer FGMRES(m_{out})-GMRES$_{SP}$(m_{in})

```
for $i$ D 0; 1; ::: do
    $r$ D $b$   $Ax_i$
        D $h_{1;0}$ D jj$r$jj$_2$
    check convergence and exit if done
    for $k$ D 1; : : : ; $m_{out}$ do
        $\mathbf{v}_k$ D $r$ = $h_{k;k\ \ 1}$
        One cycle of GMRES$_{SP}$ .$m_{in}$/ in solving $Az_k$ D $\mathbf{v}_k$, initial guess $z_k$ D 0
        $r$ D $A$ $z_k$
        for j=1,. . . ,k do
            $h_{j;k}$ D $r^T\mathbf{v}_j$
            $r$ D $r$   $h_{j;k}$ $\mathbf{v}_j$
        end for
        $h_{kC1;k}$ D jj$r$jj$_2$
        if $h_{kC1;k}$ is small enough then break
    end for
    // Define $Z_k$ D $z_1$; : : : ; $z_k$; $H_k$ D f$h_{i;j}$g$_{1\ \ i\ \ kC1;1\ \ j\ \ k}$
    Find $W_k$ D $\mathbf{w}_1$; : : : ; $\mathbf{w}_k{}^T$ that minimizes jj$b$   $A$.$x_i$ C $Z_k$ $W_k$/jj$_2$
    // note: or equivalently, find $W_k$ that minimizes jj  $e_1$   $H_k$ $W_k$jj$_2$
    $x_{iC1}$ D $x_i$ C $Z_k$ $W_k$
end for
```

□

Figure 12. Mixed precision, iterative refinement with FGMRES-GMRES$_{SP}$ from Algorithm 5 *vs.* DP FGMRES-GMRES$_{DP}$.

Figure 12 shows the performance ratio of the mixed precision, inner-outer FGMRES-GMRES$_{SP}$ *vs.* the full, double precision, inner-outer FGMRES-GMRES$_{DP}$, i.e., here we compare two inner-outer algorithms that do the same, with the only difference being that their inner loop's incomplete solvers are performed in correspondingly single and double precision arithmetic.

□

Figure 13. Mixed precision, iterative refinement with FGMRES-GMRES$_{SP}$ from Algorithm 5 *vs* full double precision FGMRES-GMRES$_{DP}$.

Figure 13 shows the performance ratio of the mixed precision, inner-outer FGMRES-GMRES$_{SP}$ *vs.* double precision GMRES. This is an experiment that shows that inner-outer type iterative methods may be very competitive compared to their original counterparts. For example, we observe a speedup for matrix #4 of up to 15 which is mostly due to an improved convergence of the inner-outer GMRES *vs.* GMRES (e.g., about 9:86 of the 15-fold speedup for matrix # 4 on the IBM PowerPC architecture is due to improved convergence). The portion of the 15-fold speedup that is due exclusively to single *vs.* double precision arithmetic can be seen in Figure 12 (about 1:5 for the IBM PowerPC).

4. Conclusions

The algorithms presented focus solely on two precisions: single and double. We see them however in a 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 limiting 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 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.

It should be noted that this process can be applied whenever a Newton or "Newton-like" method is used. That is, whenever we are computing a correction to the solution as in

$$x_{i\mathrm{C}1} \;\mathrm{D}\; x_i \quad \frac{f\,.x_i/}{f^0.x_i/}$$

or

$$x_{i\mathrm{C}1} \quad x_i \;\mathrm{D}\; \frac{f\,.x_i/}{f^0.x_i/}$$

this approach can be used. We see solving optimization problems as a natural fit.

References

[1] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[2] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[3] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, 1963.

[4] C. B. Moler. Iterative refinement in floating point. *J. ACM*, 14(2):316–321, 1967.

[5] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[6] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. J. Dongarraa. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[7] J. Kurzak and J. J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency Computat. Pract. Exper.* to appear.

[8] J. Kurzak and J. J. Dongarra. Mixed precision dense linear system solver based on cholesky factorization for the CELL processor. *Concurrency Computat. Pract. Exper.* in preparation.

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

[10] 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.

[11] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanmire Tomov. Computations to enhance the performance while achieving the 64-bit accuracy. Technical Report UT-CS-06-584, University of Tennessee Knoxville, November 2006. LAPACK Working Note 180.

[12] 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.

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

[14] 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.

[15] Timothy A. Davis Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. 18:140–158, 1997.

[16] Timothy A. Davis. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. 25:1–19, 1999.

[17] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. 30:196–199, 2004.

[18] 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.

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

[20] Kathryn Turner and Homer F. Walker. Efficient high accuracy solutions with gmres(m). *SIAM J. Sci. Stat. Comput.*, 13(3):815–825, 1992.

[21] 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.

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

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

[24] 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.

[25] Y. Notay. Flexible conjugate gradients. *SIAM Journal on Scientic Computing*, 22:1444–1460, 2000.

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

[27] 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.

[28] H. A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. *Numerical Linear Algebra with Applications*, 1(4):369–386, 1994.