Implementation of a Mixed-Precision in Solving Systems of Linear Equations on the CELL Processor *

Jakub Kurzak[1], Jack Dongarra[1,2]

[1] *Department of Computer Science, University Tennessee, Knoxville, Tennessee 37996*

[2] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 37831*

November 23, 2006

Abstract

This paper describes the design concepts behind implementations of mixed-precision linear algebra routines targeted for the Cell processor. It describes in detail the implementation of code to solve linear system of equations using Gaussian elimination in single precision with iterative refinement of the solution to the full double precision accuracy. By utilizing this approach the algorithm achieves close to an order of magnitude higher performance on the Cell processor than the performance offered by the standard double precision algorithm. Effectively the code is an implementation of the high performance LINPACK benchmark, since it meets all the requirements concerning the problem being solved and the numerical properties of the solution.

Keywords

LINPACK, HPL, Cell Broadband Engine, iterative refinement, mixed-precision algorithms

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

1 Introduction

1.1 Motivation

Initially this work was motivated by the fact that many processors today exhibit higher single precision performance than double precision performance due to SIMD vector extensions. In fact the technology has been around since the late 90s. Examples include 3DNow! extensions for AMD processors, SSE extensions for both Intel and AMD processors and VMX/AltiVec extensions for PowerPC processors. Today in most cases these extensions offer a factor of two performance advantage for single precision versus double precision calculations. The advent of the Cell processor[1–4] introduced yet more dramatic performance difference between single precision floating point unit[5] and double precision floating point unit[6] with the ratio of 14 for the *synergistic processing element* (SPE)[7, 8] and the overall ratio of more than 10 for the entire processor. With the ratio of such magnitude, it is an extremely attractive idea to exploit single precision operations whenever possible and resort to double precision at critical stages, while attempting to provide the full double precision accuracy.

1.2 Iterative Refinement

Iterative refinement is a well known method for improving the solution of a linear system of equations of the form $Ax = b$[9]. The standard approach is to use the technique of Gaussian elimination. First, the coefficient matrix A is factorized using LU decomposition into the product of a lower triangular matrix L and an upper triangular matrix U. Commonly, partial row pivoting is used to improve numerical stability resulting in the factorization $PA = LU$, where P is the row permutation matrix. The system is solved by solving $Ly = Pb$ (*forward substitution*), and then solving $Ux = y$ (*backward substitution*). Due to the roundoff error, the solution carries an error related to 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, x, at each iteration, which yields the basic iterative refinement algorithm outlined on Figure 1.

Here mixed-precision iterative refinement approach is presented. The factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients. 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 the single precision speed. The only operations 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

REPEAT

$r = b - Ax$

$z = L\backslash(U\backslash Pr)$

$x = x + z$

UNTIL x is "accurate enough"

Figure 1: Iterative refinement of the solution of a system of linear equations using Matlab^{TM} notation.

$A_{(32)}, b_{(32)} \leftarrow A, b$

$L_{(32)}, U_{(32)}, P_{(32)} \leftarrow$SGETRF$(A_{(32)})$

$x^{(1)}_{(32)} \leftarrow$SGETRS$(L_{(32)}, U_{(32)}, P_{(32)}, b_{(32)})$

$x^{(1)} \leftarrow x^{(1)}_{(32)}$

REPEAT

$r^{(i)} \leftarrow b - Ax^{(i)}$

$r^{(i)}_{(32)} \leftarrow r^{(i)}$

$z^{(i)}_{(32)} \leftarrow$SGETRS$(L_{(32)}, U_{(32)}, P_{(32)}, r^{(i)}_{(32)})$

$z^{(i)} \leftarrow z^{(i)}_{(32)}$

$x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$

UNTIL $x^{(i)}$ is "accurate enough"

Figure 2: Solution of a linear system of equations with mixed-precision iterative refinement.

to single precision for the LU factorization. At the same time, the original matrix in double precision has to be retained for the residual calculation. By the same token, the method requires 1.5 times the storage of the strictly double precision method. The mixed-precision iterative refinement algorithm is outlined on Figure 2. More details of the algorithm, including error analysis, can be found in[10].

1.3 LINPACK Benchmark

The *high performance LINPACK benchmark* (HPL)[11] is the most widely used method for measuring performance of computer systems. The computational problem posed by the HPL benchmark is a solution of a system of linear equations, where the coefficient matrix is real, general and dense with random uniform distribution of its elements. Since performance gains can be achieved by sacrificing the correctness of the solution, as a guard against such practices, constraints are imposed on the numerical properties of the solution. In general

terms, the answer is correct if it has the same relative accuracy as standard techniques, such as the Gaussian elimination with partial pivoting used in the LINPACK package, when performed in double precision. To be more precise, the following scaled residuals are computed:

$$r_n = \frac{\|Ax - b\|_\infty}{\|A\|_1 \cdot n \cdot \epsilon},$$

$$r_1 = \frac{\|Ax - b\|_\infty}{\|A\|_1 \cdot \|x\|_1 \cdot \epsilon},$$

$$r_\infty = \frac{\|Ax - b\|_\infty}{\|A\|_\infty \cdot \|x\|_\infty \cdot \epsilon},$$

where ϵ is the relative machine precision. A solution is considered numerically correct when all of these quantities are of order $O(1)$. In calculating the floating-point execution rate, the formula $2n^3/3 + 2n^2$ is used for the number of operations, regardless of the actual number.

The mixed-precision iterative refinement algorithm was implemented on the Cell processor as a *proof of concept* prototype, with the goal to pave the way for a wider range of algorithms in numerical linear algebra. At the same time, the code meets all requirements of the *high performance LINPACK benchmark*, most notably the constraint on the accuracy of the solution. By the same token, the code can be used to evaluate the performance of the Cell processor in comparison to other architectures.

1.4 Comments on the CBE Design

The most significant architectural feature of the Cell processor is its multicore design based on one PowerPC core, referred to as PPE, and eight *synergistic processing elements* (SPEs). The most interesting characteristic of the Cell is that it blurs the line between shared memory and distributed memory systems. The main memory still plays the role of the central repository for code and data, yet the SPEs can only execute code in the *local store*[12] of 265KB, and only operate on data in the *local store* with all code and data motion handled explicitly via DMA transfers, in a message passing fashion. At the same time, the communication is non-blocking in its very nature, greatly facilitating overlapping of communication and computation. Great effort has been invested throughout the years in optimizing code performance for cache-based systems, in most cases leading to the programmers reverse engineering the memory hierarchy. By requiring explicit data motion, the memory design of the Cell takes the guesswork out of the equation and delivers predictable performance.

The SPEs are inherently vector units capable of very fast single precision arithmetic. This work is motivated in particular by the single to double performance ratio of the Cell processor. An SPE can issue a single precision vector fused multiplication-addition operation per clock cycle. A vector of 128B contains four 32-bit single precision values, which means that each SPE can execute

8 operations per cycle. At the same time, a vector contains only two 64-bit double precision values. Moreover, due to space and power constraints, double precision operations are not fully pipelined. It takes one cycle to issue a double precision operation, and the operation requires a stall for another six cycles. As a result, one vector can be processed every seven cycles. The factor of two and the factor of seven combined make the ratio of single to double precision performance equal to 14 for the SPEs. This means that for a 2.4 GHz system the single precision peak of the eight SPEs is 153.6 Gflop/s and the double precision peak is 11 Gflop/s. The VMX engine of the PPE can in theory deliver single precision performance equal to that of an SPE. At the same time, the double precision arithmetic is fully pipelined on the PPE and can complete one fused multiplication-addition operation per clock cycle. If the PPE performance is also considered, then the overall performance is 172.8 Gflop/s for single precision and 15.8 Gflop/s for double precision. For the 3.2 GHz system single precision peak of the SPEs is 204.8 Gflop/s and double precision peak is 14.6 Gflop/s. The values are 230.4 Gflop/s and 21 Gflop/s if the PPE is included.

Finally, it should be noted that the SPE floating point unit only implements truncation rounding, flushes denormalized numbers to zero, and handles NaNs as normal numbers[5], which can potentially cause numerical problems. No numerical problems were encountered for input matrices with random uniform distribution of elements. Nevertheless, the issue deserves further attention.

2 Design and Implementation

2.1 Overview

At the top level the algorithm is driven by a FORTRAN 77 routine, named DGESIRSV after its LAPACK[13] double precision counterpart DGESV and, in principle, offering the same functionality, but using a mixed-precision approach. The routine is planned to be also included in the LAPACK library, possibly with slight modifications. Development of more mixed-precision routines is planned to address a wider range of problems in linear algebra including linear systems and least squares problems as well as singular value and eigenvalue problems.

The mixed-precision routine is build on top of existing LAPACK and BLAS[14] routines and, in turn, LAPACK is designed to rely on a BLAS implementation optimized for a specific hardware platform to deliver the desired performance. Due to availability of both LAPACK and a reference implementation of BLAS in source code, the functionality can be delivered immediately on the Cell processor by compiling the necessary components for execution on the PPE. The lack of existence of a FORTRAN 77 compiler in the SDK can be addressed by either using the F2C utility[15] or compilation on the Cell hardware using existing PowerPC Linux compilers, GNU G77 or IBM XLF, although only the first one is publicly available at this moment. Also, the reference BLAS can be replaced with a more optimized implementation. Possibilities include ATLAS[16], GOTO BLAS[17] and ESSL[18], with the first two being freely available at this

time. All these implementations are engineered to make efficient use of the memory hierarchy and the vector/SIMD extension of the PPE[19], and, as a result, are much faster than the reference BLAS. At the same time, by utilizing only the PPE, they are capable of delivering only a tiny fraction of the overall performance of the Cell processor. Due to unavailability of an implementation of BLAS parallelized between the SPEs at this moment, the performance of the code has to be engineered from scratch.

Nevertheless, code compiled for execution on the PPE only was used as a starting point for iterative development of the optimized version. The initial hope was that only Level 3 BLAS would have to be replaced with vectorized code parallelized between the SPEs. The emphasis in LAPACK is on implementing most of the computational work in Level 3 BLAS routines. As a result, it frequently is the case that Level 2 BLAS routines only contribute $O(n^2)$ factor to algorithms of $O(n^3)$ complexity and optimal performance of the Level 2 BLAS is not crucial. At the same time, on many multiprocessor systems parallelization of the Level 2 BLAS routines not only does not result in a speedup, but often yields a slowdown. This turned out not to be the case on the Cell, where the parallelization of Level 2 BLAS proved not only to be beneficial, but in most cases also necessary in order not to degrade the performance of the whole algorithm. By the same token, only Level 1 BLAS routines could remain implemented in the PPE BLAS and for simplicity the reference BLAS implementation from Netlib was chosen to provide this functionality.

2.2 SPE Parallelization

The basic model for developing the SPE-parallel version of the optimized routines is *master-worker*, with the PPE playing the role of the *master*, and the SPEs as the *workers*. The PPE manages the execution of the overall algorithm relying on the SPEs to deliver computational services. The PPE is responsible for launching and terminating the *workers*. The SPE execution cycle consists of waiting for a request, performing the requested task and sending back a response, which can be a positive acknowledgment, an error message or a return value.

At the time of the creation of the SPE threads the main memory address is passed to the global control block, which is then pulled by each SPE to its *local store* by a DMA transfer. The control block contains global execution parameters and main memory addresses of synchronization variables as well as effective addresses of the *local store* of each SPE to facilitate direct DMA transfers between *local stores* when it is desired. After this initial exchange of information each SPE waits for commands sent from the PPE to its inbound mailbox. The commands are integral values representing particular BLAS routines. Next, the SPE fetches the list of arguments specific for a given routine from the main memory through a DMA transfer from a location specified in the global control block. The list contains what would typically be BLAS function call arguments including input array sizes and their memory locations. Then the SPE proceeds to the computational task. When the task is finished, the SPE acknowledges

the completion to the PPE by sending a response through a DMA which is holding on a barrier with the last data transfer. The cycle continues until the PPE decides to terminate the servers by sending a termination command, at which point the SPEs finish their execution by simply returning from the *main* function.

Work partitioning is done by one- or two-dimensional decomposition of the input arrays, commonly referred to as tiling, and cyclic processing of the tiles by the SPEs. Each SPE processes a set of tiles, by pulling them from the main memory, performing the calculations, and writing the result back to main memory. Assignment of the tiles is managed either statically or dynamically, with dynamic assignment used in the LU factorization and static assignment used for all other operations. In this case the decision was arbitrary and the use of one approach versus the other should be further investigated. For many operations there are no dependencies between the tiles processed by different SPEs, and, as a result, no communication or synchronizations between the SPEs is necessary. In certain cases it is possible to remove existing dependencies by providing each SPE with an auxiliary space to store intermediate results, which are later combined by the PPE to form the final result. The implementation of matrix vector product in double precision is an example of this approach. When communication and synchronization is required, like in the case of panel factorization in LU decomposition, it is implemented by direct *local store* to *local store* DMA exchanges.

The majority of the routines in the code are built around the idea of overlapping computation and communication by pipelining of operations, which is facilitated by the DMA engines attached to the SPEs. Most of the routines follow the pattern depicted on Figure 3 with differences in the number and shape of buffers used. In many situations it is sufficient to use the technique of *double buffering*, where, for a given data stream, one tile is processed when another is being transferred. Good example of such operation is matrix multiplication $C = A \times B$, where *double buffering* can be applied to the tiles of each matrix. In this case, in each step of the algorithm, one tile of A and B can be read in, and one tile of C can be written back. The concept of *triple buffering* can be utilized when the data has to be read in, modified and written back, as it is in the case of calculating $C = C - A \times B$. Here *double buffering* is still used to bring in the tiles of A and B. However, calculation of a tile of C has to be overlapped with fetching of the next tile of C, as well as returning the tile resulting from the previous step of the loop. In this case three buffers are rolled instead of two buffers being swapped. It is also possible to use just two buffers for the tiles of C by using the same buffer for reading and writing and ordering the operations with a barrier, a solution actually utilized in the code.

2.3 Local Store Usage

One of the most prominent features of the Cell processor is the *local store*, which provides limited space of 256KB for both data and code. This enforces tiling of the matrix operations and raises the question of the optimal tile size. For a

```
Prologue
Receive tile 1
Receive tile 2
Compute tile 1
Swap buffers
Loop body
FOR I=2 TO N-1
    Send tile I-1
    Receive tile I+1
    Compute tile I
    Swap buffers
END FOR
Epilogue
Send tile N-1
Compute tile N
Send tile N
```

Figure 3: Basic model of overlapping communication and computation with tiling and pipelined processing of tiles.

number of reasons the size of 64 by 64 elements in single precision was chosen. In particular, for this size, an optimized matrix multiplication kernel can achieve within 98% of the peak of an SPE. Also, the matrix multiplication operations are implemented using tiles of this size. This provides a transfer to computation ratio, which allows to fully overlap communication with computation. At the same time, the size of a tile is 16KB, which is the maximum size of a single DMA transfer. By the same token, if *block layout*[20, 21] is used (§2.4), a whole tile can be transferred in one DMA transfer. Moreover, the size of a tile is a multiplicity of 128B, which is the size of a cache line. This means that, if a matrix is aligned at a 128B boundary, then each of its tiles is aligned on a 128B boundary, what is beneficial for the performance of DMA transfers. Lastly, a cache line aligned DMA of size 16KB perfectly balances memory accesses to all 16 memory banks, allowing for maximum utilization of the memory bandwidth. Not without significance is the fact that the tile size is a power of two, which can simplify efficient implementations of recursive formulations of many linear algebra algorithms.

The tile size of 64 by 64 is perfect for implementing matrix multiplication $C = A \times B$, in particular when A, B and C are of considerable size and relatively square. The most time consuming part of the LINPACK benchmark is the update to the trailing matrix in the LU factorization in single precision, $C = C - A \times B$. Although in principle the operation is a matrix multiplication, it would better be described as a block outer product, since C is of size $m \times n$, A is of size $m \times NB$ and B is of size $NB \times n$, NB being the block size. Unfortunately, this operation is much more demanding in terms of communication. It could

only achieve the peak in theory if bus utilization was perfect. In practice it achieves 80% of the peak, so in the future bigger tile sizes should be taken into consideration.

A second question is the number of tile buffers to be allocated. Again, the most demanding operation here is the update to the trailing matrix in the LU factorization in single precision, $C = C - A \times B$. In, order to update a tile of matrix C, an SPE has to read in a tile of C, a tile of A and a tile of B, perform the computation and write back the updated tile of C. If buffer usage is maximized for the sake of communication overlapping, the tiles of A and B are *double buffered* and the tiles of C are *triple buffered* (§2.2), which means that the total of seven buffers are required. Alternatively, reading in a tile of C and writing it back after the update can be separated with a barrier, in which case tiles of C are *double buffered* and only a total number of six buffers is required. The implementation actually allocates eight buffers for a couple of reasons. Obviously, it is beneficial for the number of buffers to be a power of two. For some operations it may be advantageous to temporarily transpose a tile, in which case an auxiliary buffer may be necessary. Larger buffer space can be taken advantage of when certain operations can be executed entirely in the *local store*, without the need to write back intermediate results to the main memory. It also allows more DMA requests to be queued for memory intensive operations, like the conversion from *standard* to *block layout*. On the other hand, eight tiles of 16KB sum up to 128KB, which constitutes half of the *local store* and going beyond that would be a serious limitation for the space for code.

Finally, the last issue is the tile size in double precision, which cannot be the same as tile same size in single precision. The minimum of six buffers is required to implement matrix multiplication efficiently. Six buffers of size 64 by 64 in double precision would consume 192KB of the *local store*, leaving dangerously little space for the code. The choice was made to use the closest smaller power of two of 32, in which case, as in single precision, the 128B memory alignment property also holds for each tile, each tile can be transferred in a single DMA and utilization of memory banks is fully balanced when *block layout* is used. In the general case, the use of a smaller tile introduces inefficiencies due to bigger communication overhead and a worse ratio of memory accesses to floating point operations. In this case, however, these inefficiencies are negligible due to an order of magnitude lower speed of double precision arithmetic. Since the double precision buffers are aliased to the single precision buffers, 16 double precision buffers are available. Although such number is not requited for the matrix multiplication, same as for single precision, they prove useful for operations, which can take place entirely in the *local store* and for memory intensive storage and precision conversions.

2.4 Block Layout and Large Pages

Traditionally the matrices are stored in the main memory in a column-major or row-major order, where all column elements, or row elements respectively, are stored continuously in memory, which is further referred to as *standard*

layout. Column-major order is assumed unless stated otherwise. Optimized linear algebra routines use block algorithms in order to implement most of their operations in Level 3 BLAS, and frequently access submatrices of sizes being multiples of the block size. At the same time, most matrix operations on the Cell have to be implemented with tiling, due to the limited size of the *local store*. By the same token, the data in memory is accessed by blocks of fixed size most of the time.

The communication mechanism of the Cell offers a convenient way of accessing tiles in the main memory by using DMA lists, which in principle can be as fast as DMA transfers of continuous memory blocks. That is, however, only if TLB misses and optimal usage of memory banks do not come into play. Pulling a tile from the main memory using a DMA list is an example of strided memory access and, unfortunately, due to the two issues mentioned, its performance largely depends on the stride, which in this case is the *leading dimension* of the input matrix. The memory subsystem of the Cell has 16 banks interleaved on cache line boundaries, and addresses 2KB apart access the same bank. For a tile of 64 by 64 in single precision the transfer of each DMA list element accesses two banks. The worst case scenario is, when the leading dimension of the matrix is 2KB or 512 single precision elements. In this case, each DMA list element accesses the same two banks, and only those two banks are accessed for the transfer of the entire tile. The fact that more than one SPE can be issuing requests to the same memory banks may further aggregate the situation. One possible approach is to simply try to avoid the troublesome matrix sizes. In general this is not a satisfactory solution though.

The second problem is that, with the standard page size of 4KB, accesses to strided data are likely to access different memory pages and generate many TLB misses, which may turn out to be fatal in case of relatively small TLBs of the SPEs (256 entries, vs. 1024 entries for the PPE[22]). For instance if the leading dimension of the matrix is larger than the page size, which typically is 4KB or 1024 single precision elements, then each DMA list element accesses a different page, and can potentially generate a page fault. As large numbers of pages are accessed, TLB thrashing occurs, resulting in performance degradation.

The solution to the first problem is storage of the matrices in *block layout*. Here blocks of 64 by 64 single precision elements are stored continuously in the memory and row-major order is used within the blocks as well as on the block level. The same storage is used for double precision with blocks of size 32 by 32. In this case each single DMA operates on either a 16KB or a 8KB continuous memory blocks uniformly distributing accesses to all 16 memory banks with the additional benefit, that a single tile can be read or written with a single DMA instead of a DMA list.

Since the input matrices are stored in the *standard* column-major *layout*, conversion operations are required. Due to the fact that the iterative refinement algorithm requires the conversion of the coefficient matrix from single precision to double precision, this operation is performed first. Then the single precision matrix is translated to *block layout* with 64 by 64 blocks and the double precision matrix is translated to *block layout* with 32 by 32 blocks. The conversion from

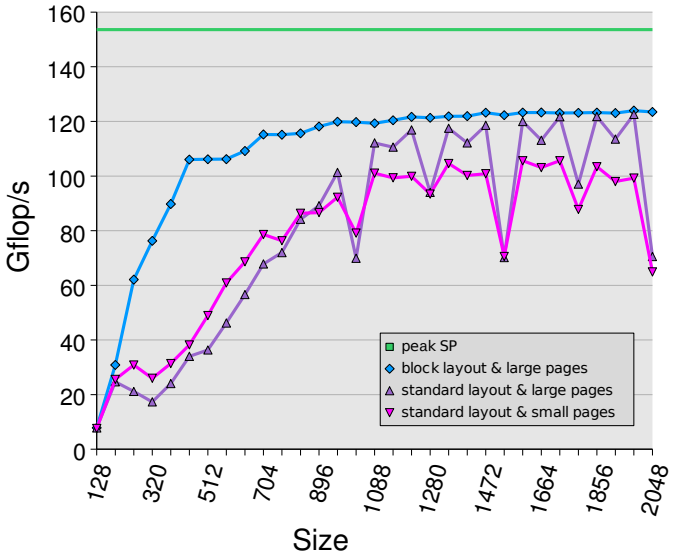

Figure 4: Performance of block outer product $C = C - A \times B$, where C is of size $N \times N$, A is of size $N \times 64$ and B is of size $64 \times N$, on a 2.4 GHz Cell BE.

single to double, as well as the two conversions from *standard* to *block layout*, are performed in parallel by all SPEs. Also, both the transposition in the layout conversion step and the precision conversion are subject to vectorization.

The solution to the TLB thrashing problem is the use of large pages. Here pages of 16MB are used. There is the question of to what extent *block layout* can solve the problem of page faults when small pages are used. It could potentially solve the TLB performance problem in the main algorithm. Unfortunately, there still remain the operations performing the conversion from *standard* storage to *block layout*. The experience shows that the use of small pages can degrade the performance of these conversions by an order of magnitude, effectively making them prohibitively expensive. The issue can be further investigated. Figure 4 shows the performance impact of using large pages and *block layout* on the calculation of block outer product in single precision.

2.5 More on Optimizations

Manual vectorization with inlined assembler[23] and C intrinsics[24] and manual loop unrolling are heavily used for performance optimization of the code. The code relies largely in its performance on optimized math kernels for processing of the tiles. The best example is the matrix multiplication implementing the

functionality of the BLAS routine SGEMM. The code is manually unrolled and relies heavily on inlined assembler statements. It is also manually tuned to maximize the amount of dual issue and achieves a dual issue ratio above 90%. In many cases inlined assembler and manual optimization for dual issue are not used. Nevertheless, vectorization with C language intrinsics and manual loop unrolling is prevalent in the code, and even applied to such auxiliary operations as precision conversions and DMA list creation.

For both performance, as well as correctness of the DMA transfers, all memory allocations are made with alignment to the cache line size of 128B, and most of control data structures are rounded up in size to 128B by padding with empty space.

Also, for performance reasons the code does not pay particular attention to possible numerical problems, which is further commented on in the following section.

2.6 Limitations

Although in principle the top level FORTRAN 77 routine accepts multiple right hand sides, the underlaying Cell-specific code only supports a single right hand side.

The code requires that the input coefficient matrix is in standard FORTRAN 77 style column-major layout. Due to the use of *block layout* the size has to be a multiplicity of the block size of 64. The translation from *standard* to *block layout* is included in program timing and in the calculation of the Gflop/s number, and turns out not to pose a significant performance problem. However, at this moment the code excessively allocates memory due to the fact that the coefficient matrix is stored in both double and single precision and in both *standard* and *block layout*. This considerably limits the size of problems which can be solved with a given amount of main memory. Also, the code requires large page support or otherwise the performance is unacceptable, mainly due to slow speed of the layout and precision conversion operations. It is assumed that pages of size 16MB are used.

The number of numerical problems are neglected at this time due to performance reasons. Obviously, a smaller range of numbers is representable in single precision than in double precision, and a check for overflow would be desirable, but, at the same time, introduce unacceptable performance overhead. Overflow is also possible when calculating norms of vectors and matrices, and for the same reasons it is not checked for. For instance the LAPACK DLANGE routine is basically implemented as DDOT.

If for these or other reasons the result does not meet the required bound on the backward error, as a fall-back strategy, the factorization is performed entirely in double precision, and, at this moment, by calling to the PPE BLAS. More details on the numerical behavior of the algorithm can be found in[10].

3 Results

The results presented here were collected on a 2.4 GHz Cell blade using only one of the two processors located on the board. The Gflop/s numbers reported here mean the actual number of floating point operations over time for the codes running exclusively single or double precision calculations. For the mixed-precision iterative refinement code, the Gflop/s number means performance relative to the double precision case. In other words, it is the speed required by the double precision code to deliver the same results in the same amount of time.

Due to very suboptimal use of memory, the largest system which could be run was of size 3712 by 3712. A uniform random matrix was used as the coefficient matrix. The relative norm-wise backward error of $O(10^{-14})$ was achieved in four iterations of the iterative step. The system was solved in 0.37 second, with the relative speed of 84 Gflop/s, which is 5.4 times greater than the total double precision peak of the Cell, including all eight SPEs and the PPE, 7.7 times greater than the double precision peak of the eight SPEs only, and 9.9 times greater than the actual speed of solving the system entirely in double precision on the eight SPEs. Figure 5 shows the performance comparison between the single precision algorithm, the double precision algorithm and the mixed-precision iterative refinement algorithm.

Figure 6 shows the breakdown of the execution time for the mixed-precision algorithm. Individual routines are referred to using their equivalent BLAS or LAPACK names with the exception of the conversion from single to double precision (s2d) and double to single precision (d2s), the conversion from *standard* (LAPACK) *layout* to *block layout* in single precision (l2b) and the conversion from *standard layout* to *block layout* in double precision (l2b_d). The most time is spent in the factorization of the coefficient matrix in single precision, which is the desired behavior. The two operations which contribute the most to the overhead of iterative refinement are solution of the triangular system in single precision (sgetrs) and matrix-vector multiplication in double precision (dgemm/dgemv), which is to be expected. The overhead from all other routines, including layout and precision conversions, is minimal.

The code was also run on a 3.2 GHz Cell system. It achieved 98.05 Gflop/s, which is less than the expected gain from the faster clock comparing to the 2.4 GHz system. Due to limited availability of the 3.2 GHz hardware, it was not possible to address some of the performance issues. Nevertheless, as of the August 1st 2006 the 3.2 GHz Cell system is included in the LINPACK report [25], where it outperforms a number systems based on modern HPC processors.

4 Conclusions

The results presented here show the huge potential of the mixed precision approach to the development of numerical algorithms in particular in the area of numerical linear algebra. The method is applicable to a wide range of algorithm for solutions of linear systems and least square problems as well as singular and

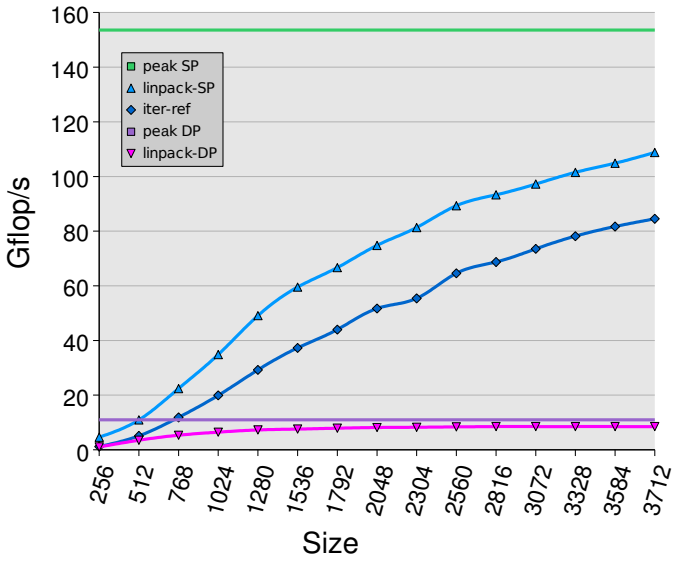

Figure 5: Performance comparison of solving linear system of equation $A \times x = b$ on a 2.4 GHz Cell BE using single precision, double precision and mixed-precision approach.

eigenvalue problems.

The results also prove the huge potential of the Cell processor for high performance numerical applications. The Cell architecture allows for much finer granularity of parallelism then the traditional architectures. It also encourages much more dynamic and asynchronous algorithm behavior and may be a good target for testing concepts like work-queue parallelization. By blurring the boundaries between shared and distributed memory systems the Cell has the potential to inspire new algorithmic discoveries in the area of numerical computing.

5 Future Plans

Despite the fact that significant effort was invested in the current implementation, the code suffers from numerous performance problems. The block size used here does not allow the block outer product operation to achieve the peak when parallelized between all eight SPEs. Wasteful memory usage limits the size of systems which can be solved. At the same time, the cost of panel factorization prevents the code from achieving good performance for systems of moderate sizes. Also, the triangular solve is not parallelized between SPEs at

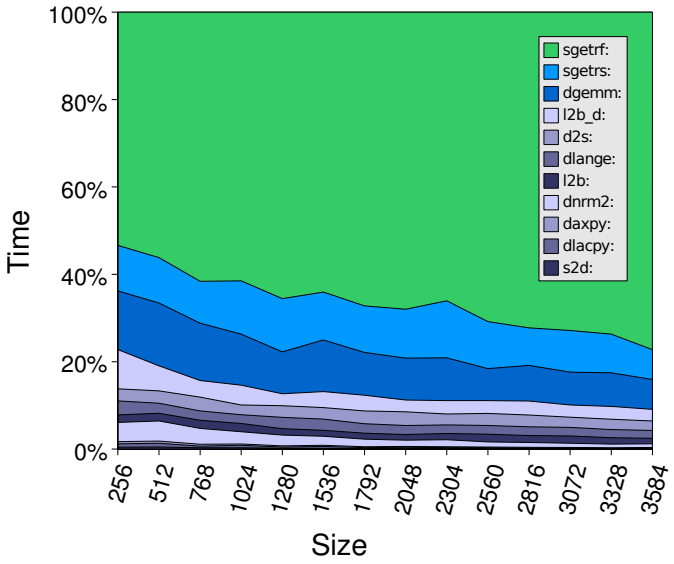

Figure 6: Breakdown of the execution time of solving linear system of equations using mixed-precision iterative refinement on a 2.4 GHz Cell BE.

this time. Although the reported performance is rather impressive, addressing the shortcomings listed above should yield substantial further performance increases. Finally, right now the code can only utilize a single Cell processor, and parallelization between multiple Cell systems with message passing is envisioned in the future.

Hopefully the early experiences with the iterative refinement code can guide the design of the BLAS for the Cell processor. A number of crucial design questions remain. Probably the most important is the one of *block layout*. Experience shows that *block layout* offers unquestionable performance advantages. At the same time, it seems unlikely that data layout can be hidden within BLAS and not exposed to higher software layers in one way or another. In particular LAPACK uses block operations and it will be necessary to synchronize the block sizes between LAPACK and BLAS. At the same time, LAPACK has no notion of *block layout* and code modifications would be required to facilitate it. The question remains if, and how, the *block layout* should be exposed to the user, and if conversion is required, how is it handled. It does not help the situation that different block sizes may be necessary for single and double precision and in both cases the question of the optimal block size remains unanswered. A related issue is the one of introducing constraints to the BLAS and possibly also LAPACK implementations. In both BLAS and LAPACK significant inef-

ficiencies are caused by matrix sizes not being a multiplicity of the block size. In case of the Cell the impact can reach such proportions that introduction of constraints in the input array sizes can be justifiable. At this time it is quite certain that a library like BLAS will not be able to fit in the local store in its entirety. Code motion at runtime will be necessary and, although the concept is simple in principle, the question remains if it should be resolved internally in BLAS or exposed to the higher level library. Also, reliance on parallel BLAS can prevent interesting algorithmic improvements and it may be desirable to bypass the standard BLAS API and directly utilize the underlaying high performance kernels. The question remains of utilizing the PPE of the Cell, which is capable in matching the performance of an SPE in single precision, and is much more powerful than a single SPE in double precision. Implementations of the PPE BLAS already exist, although as of today none of them is well tuned for the hardware. The availability of BLAS in all three instances of single SPE BLAS, SPE-parallel BLAS and PPE BLAS would provide the application developer with an extremely flexible and powerful tool and not only facilitate quick port of libraries like LAPACK, but also enable the pursuit of new algorithms in numerical linear algebra and other computational disciplines.

Although it is hard to predict the future hardware road map for the Cell processor, improvements to the double precision performance of the processor would be very welcome, as long as single precision performance is not sacrificed. One interesting concept is the Cell+ architecture [26].

6 Acknowledgments

The authors would like to thank IBM for providing hardware and support, in particular the people at IBM Austin Research Laboratory for sharing their expertise with the Cell processor. Special thanks to Sidney Manning for his enthusiasm towards the project. The authors would also like to thank Mercury Computer Systems for providing access to the 3.2 GHz Cell hardware.

References

[1] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 184–185, 2005.

[2] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture*, 2005.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and

D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. & Dev.*, 49(4/5):589–604, 2005.

[4] IBM. *Cell Broadband Engine Architecture, Version 1.0*, August 2005.

[5] H. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. H. Dhong. A fully-pipelined single-precision floating point unit in the synergistic processor element of a CELL processor. In *Symposium on VLSI Circuits*, pages 24–27, 2005.

[6] J. B. Kuang, T. C. Buchholtz, S. M. Dance, J. D. Warnock, S. N. Storino, D. Wendel, and D. H. Bradley. A double-precision multiplier with fine-grained clock-gating support for a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 378–379, 2005.

[7] B. Flachs, S. Asano, S. H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processorn. In *IEEE International Solid-State Circuits Conference*, pages 134–135, 2005.

[8] O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Murakami, H. Noro, H. Oh, S. Onishi, J. Pille, J. Silberman, and S. Yong. The circuits and physical design of the synergistic processor element of a CELL processor. In *Symposium on VLSI Circuits*, pages 20–23, 2005.

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

[10] 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. to appear.

[11] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[12] S. H. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. Silberman, A. Kawasumi, and H. Yoshihara. A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 486–487, 2005.

[13] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[14] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001.

[15] S. I. Feldman, D. M. Gay, M. W. Maimone, and Schryerr N. L. A Fortran-to-C converter. Computing Science Technical Report 149, AT&T Bell Laboratories, 1990.

[16] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.*, 27(1-2):3–35, 2001.

[17] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-02-55, Department of Computer Sciences, University of Texas at Austin, 2002.

[18] IBM. *Engineering and Scientific Subroutine Library for Linux on POWER, Version 4 Release 2.2*, November 2005.

[19] IBM. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual, Version 2.06*, August 2005.

[20] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel and Distrib. Systems*, 14(7):640–654, 2003.

[21] N. Park, B. Hong, and V. K. Prasanna. Analysis of memory hierarchy performance of block data layout. In *Proceedings of the International Conference on Parallel Processing*, 2002.

[22] IBM. *Cell Broadband Engine Programming Handbook, Version 1.0*, April 2006.

[23] IBM. *SPU Assembly Language Specification, Version 1.3*, October 2005.

[24] IBM. *SPU C/C++ Language Extensions, Version 2.1*, March 2006.

[25] J. J. Dongarra. Performance of various computers using standard linear equations solver. Technical Report CS-89-85, Computer Science Department, University of Tennessee, 2006. www.netlib.org/benchmark/performance.ps.

[26] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *ACM International Conference on Computing Frontiers*, 2006.