High-Performance Mixed-Precision Linear Solver for FPGAs

Junqing Sun, *Student Member*, *IEEE*, Gregory D. Peterson, *Senior Member*, *IEEE*, and Olaf O. Storaasli

Abstract—Compared to higher precision data formats, lower precision data formats result in higher performance for computationally intensive applications on FPGAs because of their lower resource cost, reduced memory bandwidth requirements, and higher circuit frequency. On the other hand, scientific computations usually demand highly accurate solutions. This paper seeks to utilize lower precision data formats whenever possible for higher performance without losing the accuracy of higher precision data formats by using mixed-precision algorithms and architectures. First, we analyze the floating-point performance of different data formats on FPGAs. Second, we introduce mixed-precision iterative refinement algorithms for linear solvers and give an error analysis. Finally, we propose an innovative architecture for a mixed-precision direct solver for reconfigurable computing. Our results show that our mixed-precision algorithm and architecture significantly improve the performance of linear solvers on FPGAs.

Index Terms—Reconfigurable computing, supercomputing, floating point, data formats, iterative refinement.

1 Introduction

1.1 Floating Point on FPGAs

Because of their intrinsic parallelism, pipelining ability, and flexible architecture, field-programmable gate arrays (FPGAs) have shown great potential in accelerating floating-point computations. Previous work concluded that the peak floating-point performance on FPGAs has surpassed that of CPUs [2]. To exploit the floating-point capability of FPGAs, many researchers and commercial vendors have developed floating-point intellectual property (IP) cores on FPGAs. Xilinx, a large FPGA vendor, has included pipelined floating-point operators in its ISE tools [16]. Their data format and pipeline depth can both be customized. To reduce hardware resource cost and achieve high performance, some operators can be built from either combinational logic slices or more efficient embedded circuits such as built-in 18×18 multipliers. Related floating-point IP cores can also be found in academic research groups [17], [18].

For these floating-point operators on FPGAs, lower precision data formats require fewer resources and also result in higher circuit frequency. As we discuss later, the resource cost increases linearly for adders and quadratically for multipliers for the commonly used floating-point IP cores [16]. Our test results reveal that the floating-point performance for lower precision data formats greatly exceeds that of higher precision data formats. However, high accuracy is often required by scientific applications. For example, molecular dynamics applications demand a certain accuracy for convergence [30].

1.2 Reconfigurable Linear Algebra

Linear algebra is widely used in scientific computations. Because of its intensive floating-point computational loads and high communication-to-computation ratio, linear algebra usually cannot achieve good performance on traditional computers. Utilizing FPGAs for high-performance linear algebra computation has been explored with promising speedups achieved over CPUs [22]. Previous efforts [20] and our design experience [13], [19] reveal that the performance and resource cost of reconfigurable linear algebra design highly depend on the size, frequency, and pipeline latency of floating-point IP cores.

Lower precision designs can achieve higher performance by accommodating more floating-point operators, increasing the clock frequency, and shortening the pipeline latency. Memory bandwidth is another key factor affecting the performance of FPGA accelerators for linear algebra [21]. For instance, the actual performance in [19] and [20] is usually limited by the memory bandwidth. Using lower precision arithmetic directly enables higher performance for these designs because of reduced memory bandwidth needs.

1.3 Mixed-Precision Iterative Refinement

The motivation of this paper is to exploit lower precision data formats whenever possible for high performance and achieve higher precision by iterative refinement in higher precision. Iterative refinement algorithms for linear solvers have been used to improve solutions' accuracy for many years. While the traditional algorithms use the same data format, we plan to use mixed data formats to make use of the high performance of lower precision formats and achieve the accuracy of higher precision data formats.

- *J. Sun and G.D. Peterson are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996. E-mail: {jsun5, gdp}@utk.edu.*
- *O.O. Storaasli is with the Future Technologies Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1 Bethel Valley Rd., PO Box 2008, MS-6173, Oak Ridge, TN 37831-6173. He is also with the Joint Institute for Computer Science, University of Tennessee, Knoxville, TN 37996. E-mail: Olaf@ornl.gov.*

Manuscript received 24 July 2007; revised 14 Jan. 2008; accepted 27 Mar. 2008; published online 20 May 2008.
Recommended for acceptance by W. Najjar.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-07-0379.
Digital Object Identifier no. 10.1109/TC.2008.89.

1 : *Initialize* x_0
2 : ***for*** $k = 1, 2, \cdots$
3 : $r_k \leftarrow b - Ax_{k-1}$
4 : *solve* $Ad_k = r_k$
5 : $x_k \leftarrow x_{k-1} + d_k$
6 : ***end for***

Fig. 1. Iterative refinement for linear equations.

```
Factorize A to LU: PA=LU.
Solve LUx=Pb
while (r is too big & maximum loop not reached)
        r=b-Ax
        Solve Ly=Pr
        Solve Uz=y
        x=x+z
end
```

Fig. 2. Direct solver using iterative refinement.

Using mixed precision requires floating-point components in different data formats. This can be realized by implementing different precision floating-point IP cores on a single FPGA chip. An alternative approach is to have different precision arithmetic on different FPGAs or on both FPGAs and CPUs. The former approach requires that the FPGAs have a big enough capacity to accommodate parallel computational engines for multiple data formats. An issue with the second approach is that the data movement within FPGAs or between FPGAs and CPUs should not be too frequent to affect the performance. We use the latter approach for this paper.

1.4 Paper Overview

The rest of the paper is organized as follows: Section 2 introduces mixed-precision algorithms, Section 3 analyzes performance improvements and resource savings of mixed-precision algorithms on FPGAs, our mixed-precision reconfigurable direct solver design is described in Section 4, Section 5 maps our design onto the Cray-XD1 supercomputer and gives test results, and Section 6 discusses the performance of mixed-precision design on FPGAs. Finally, we draw conclusions and suggest future work.

2 Mixed-Precision Algorithms

2.1 Iterative Refinement

Iterative refinement algorithms were introduced in the 1960s [23]. Demmel et al. provide thorough information on iterative refinement algorithms in [24]. We consider the iterative refinement algorithm shown in Fig. 1. Once an equation is solved at step 1, the solution can be refined through an iterative procedure. In each of the iterations, the residual is computed based on the solution at the previous iteration (step 3), a correction is computed in step 4 by using the computed residual, and finally, this correction is applied in step 5 for the updated solution.

2.2 Mixed-Precision Direct Solver

Experimental analyses and product descriptions reveal that current computer architectures have higher floating-point performance for single-precision operations than double-precision operations in many cases [3]. Buttari et al. investigated single/double mixed-precision algorithms for linear solvers [3]. Their results show that mixed-precision algorithms can exploit single-precision floating-point performance and also achieve double-precision accuracy on several architectures such as traditional CPUs and cell processors [3]. Strzodka and Göddeke discussed using mixed-precision algorithms for iterative solvers [4], [5], [6]. Their approach separated the loops of iterative solvers into two parts: lower precision inner loops and higher precision outer loops. The results from inner loops are used as preconditions for the outer loops. They used software to emulate the possible inner and outer loops required by mixed-precision algorithms and compared the results to double-precision algorithms.

Because of their ability to support flexible data formats, it is natural to improve the performance of linear algebra on FPGAs [13] by using mixed-precision algorithms. Although some previous work pointed out the possibility of implanting mixed-precision linear algebra on FPGAs [3], [5], no real architecture has been developed before. To the best of our knowledge, this paper is the first mixed-precision linear solver design on FPGAs and the first report for the experimental performance of mixed-precision algorithms on reconfigurable computers.

In contrast to [5], which used different data formats for loops of iterative solvers, we explore a mixed-precision algorithm for direct solvers. Suppose matrix A can be factored as $PA = LU$ with partial pivoting, where L is a lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix used for pivoting. The direct solver with iterative refinement is shown in Fig. 2, where the first two steps are very similar to traditional direct solver algorithms, and the refinement loops are taken to improve the accuracy based on the available solution. Demmel pointed out that the iterative refinement process is similar to Newton's method applied to $f(x) = b - Ax$. If all the computations were done exactly, it would converge in one step [14].

The key idea of mixed-precision algorithms here is that the factorization $PA = LU$ and the triangular solver $LUx = Pb$ are computed in lower precision, while the residual and update of the solution are computed in higher precision [3]. Wilkinson [7] and Moler [9] showed that this algorithm produces a solution correct to the working precision, provided that the matrix A is not too ill-conditioned. Demmel et al. [10] pointed out that the behavior of the mixed-precision method depends strongly on the accuracy with which the residual is computed. The error analysis for mixed-precision iterative refinement shows that this approach can achieve the same accuracy as double-precision arithmetic, provided that the matrix is not too badly conditioned [3].

The potential performance gain of using the mixed-precision algorithm comes from faster computation for factorization, which is $O(n^3)$ and dominates the runtime of the algorithm in Fig. 2. The other steps, including the triangular solver, residual computation, and solution update, are just $O(n^2)$. Furthermore, smaller data formats usually reduce the memory bandwidth requirement.

TABLE 1
Average Refinement Iterations for a Customized Format (s16e7)

Problem size (n)	Average condition number	Average iterations	Failure (iterations>30)
128	913	4	0
256	1818	5.1	0
512	4017	6.1	0
1024	6196	6.3	0
2048	9407	9.3	1/100
4096	22425	13.3	2/100

2.3 Error Analysis

Previous work addressed the error analysis of iterative refinement techniques. Higham derived error bounds for fixed-precision iterative refinement [25]. For single/double mixed-precision iterative refinement executing the refinement in double-precision arithmetic, [25] derives error bounds in single precision. Stewart gives an error analysis of iterative refinement [11]. Buttari et al. derived the results from [11] and give error bounds in double precision for a single/double mixed-precision algorithm with iterative refinement performed in double-precision arithmetic [3]. The result in [3] reveals that a mixed-precision algorithm can achieve the same accuracy as with higher precision, provided that the matrix is not too badly conditioned.

Data formats utilized in our design are much more flexible than merely single and double precision, so we extend the results in [3] for iterative refinement methods performed to general higher/lower mixed-precision arithmetic. We consider mixed-precision iterative refinement algorithms that execute steps 3 and 5 in Fig. 1 in higher precision ε_{high} but other steps in lower precision ε_{low}. If the matrix A is not too-ill conditioned with respect to the lower precision arithmetic, that is $\psi_k(n)\kappa(A)\varepsilon_{low} < 1$, from the results in [3], we have

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

where α_B and β_B are of the form

$$\alpha_B = \psi_B(n)\kappa(A)\varepsilon_{low}, \tag{2}$$

$$\beta_B = \rho_B(n)\varepsilon_{high}, \tag{3}$$

and $\psi_k(n)$, $\psi_B(n)$, and $\rho_B(A)$ are small functions of n explicitly defined in [3]. α_B depends on $\kappa(A)$ and ε_{low}, which are the condition number of the matrix A and the implemented lower precision. α_B indicates the convergence rate. β_B depends on the higher precision used ε_{high} and determines the limiting accuracy of the algorithm.

At convergence, the following exists:

$$\begin{aligned}\lim_{k\to\infty} \frac{\|b - Ax_k\|}{\|A\| \cdot \|x_k\|} &= \beta_B(1 - \alpha_B)^{-1} \\ &= \frac{\rho_B(n)}{1 - \psi_B(n)\kappa(A)\varepsilon_{low}}\varepsilon_{high}.\end{aligned} \tag{4}$$

This indicates that normwise accuracy is achieved as with the higher precision.

3 Mixed Precision on FPGAs

3.1 Iterative Refinement

Given that FPGAs enable more flexible data formats than traditional processors, it is valuable to find the data formats optimal for both accuracy and performance. More specifically, lower precision data formats help to increase the circuit frequency and reduce resource cost and bus bandwidth requirements. On the other hand, using lower precision for LU factorization might require more refinement iterations and may even result in a failure to converge. As an example, we test the convergence and iteration loops of a mixed-precision direct solver using double precision (s52e11, with 52-bit mantissa and 11-bit exponent) and a customized format (s16e7). The refinement stops when either the solver achieves the accuracy of the double-precision algorithm or there are more than 30 iterations. The latter case is considered to be a failure of convergence.

Table 1 shows the test results for 100 random matrices whose entries are generated by Gaussian random number generators. The required number of refinement iterations depends on the matrix condition numbers [3]. When the matrix size increases, we observe that the condition numbers increase, and more refinement iterations are required. Note that for large problems, the refinement takes a very small percentage $(O(n^2)/O(n^3))$ of the overall time, so a minor increase in the number of iterations will have little performance impact. We observe one failure out of 100 tests when the problem size is 2,048 and two failures when the problem size is 4,096. In Table 2, we test the number of iterations required for different data formats. We keep the same number of exponent bits as in double precision and vary the size of the mantissa in Table 2. The number of iterations increases from right to left and from top to bottom, which corresponds to reduced precision and matrices with bigger condition numbers. Due to the large problem size and short mantissa, convergence failures frequently appear at the lower left and are marked by hyphens.

TABLE 2
Average Refinement Iterations for Different Data Formats

Problem Size \ Mantissa Bits	12	16	23	31	48	52
128	8.9	4	2	1	1	0
256	11.1	5.1	2.1	1	1	0
512	19.7	6.1	2.5	1	1	0
1024	28	6.3	2.6	1	1	0
2048	-	9.3	3	1.3	1	0
4096	-	13.3	3.1	1.43	1	0

TABLE 3
Characteristics of a Multiplier on XC4LX160-10 (Using DSP48s)

Data Formats	DSP48s	Frequency (MHz)	Latenc y	GFLOPs
s52e11 (double)	16/96	237	21	1.42
s51e11	16/96	238	21	1.43
s50e11	9/96	245	19	2.61
s34e8	9/96	289	14	3.08
s33e8	4/96	292	9	7.01
s23e8 (single)	4/96	339	9	8.14
s17e8	4/96	370	9	8.88
s16e8	1/96	331	6	31.78
s16e7	1/96	352	6	33.79
s13e7	1/96	336	6	32.26

3.2 Resource Cost

We find that the floating-point performance of FPGAs can be increased significantly by using lower precision data formats. One reason is that lower precision data formats reduce the resource cost, and therefore, more floating-point operators can be implemented. At the same time, lower precision data formats reduce the memory space and bus bandwidth. This is crucial to linear algebra applications, which usually require frequent data movement. Finally, using lower precision data formats also reduces the latency of floating-point units, which is an important factor for the performance of linear algebra on FPGAs [19].

Modern FPGAs utilize embedded circuits for higher performance. One case is the embedded DSP48 blocks in the Xilinx Virtex-4 FPGA families. Because each DSP48 can be configured as an 18×18 multiplier (including a sign bit), data formats wider than 18 bits require multiple embedded units. Therefore, designs using embedded multipliers might result in significant resource saving by selecting suitable data formats such as those highlighted in Table 3. All frequency reports here come from Xilinx place-and-route tools, with the Place-and-Route Effort Level high. If we assume that all the DSP48s are configured as multipliers, the floating-point performance (in gigaflops) can be computed by multiplying the number of multipliers available and the frequency.

Tables 4 and 5 show the characteristics of implementing one multiplier or one adder by using slices. To compute the FPGA gigaflops performance for adders and multipliers, we assume that only 70 percent of the slices can be configured as multipliers or adders, the rest are reserved for other circuits and routing. This is a reasonable assumption according to previous linear algebra designs on FPGAs [13].

4 Architecture Design

4.1 LU Decomposition

LU decomposition is a widely used matrix factorization algorithm. It transforms a square matrix A into a lower triangular matrix L and an upper matrix U with $A = LU$.

TABLE 4
Characteristics of a Multiplier on XC4LX160-10 (Using Slices)

Data Formats	Slices	Frequency (MHz)	Latency	GFLOPs
s52e11 (double)	1392/67584	184	9	6.25
s51e11	1368/67584	184	9	6.36
s50e11	1326/67584	191	9	6.81
s34e8	656/67584	199	8	14.35
s33e8	644/67584	207	8	15.21
s23e8 (single)	388/67584	286	8	34.87
s17e8	274/67584	265	7	45.75
s16e8	237/67584	283	7	56.49
s16e7	233/67584	257	7	52.18
s13e7	185/67584	343	7	87.71

TABLE 5
Characteristics of an Adder on XC4LX160-10

Data Formats	Slices	Frequency (MHz)	Latency	GFLOPs
s52e11 (double)	778/67584	235	12	14.29
s51e11	772/67584	239	12	14.65
s50e11	754/67584	245	12	15.37
s34e8	531/67584	278	12	24.77
s33e8	510/67584	268	12	24.86
s23e8 (single)	380/67584	287	11	35.73
s17e8	314/67584	278	11	41.88
s16e8	301/67584	309	11	48.57
s16e7	293/67584	266	11	42.95
s13e7	244/67584	287	10	55.65

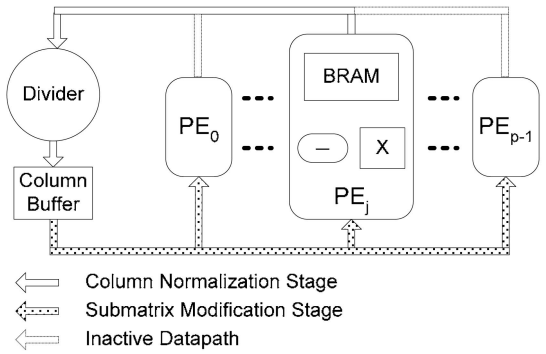

Fig. 3. LU decomposition without pivoting.

The elements of A, L, and U are denoted as $a_{x,y}$, $l_{x,y}$, and $u_{x,y}$, respectively. As shown in the following steps, the Dolittle algorithm for LU decomposition does the elimination column by column from left to right. It results in a unit lower triangular matrix and an upper triangular matrix that can use the storage of the original matrix A [15]. For simplicity, we first assume that no pivoting is needed. The architecture with pivoting will be discussed later. This algorithm requires $2n^3/3$ floating-point operations. Partial pivoting adds only a quadratic term and can thus be neglected here.

Step 1: column normalization. The elements $a_{x,0}$ in the first column below the diagonal element $a_{0,0}$ are divided by $a_{0,0}$.

Step 2: submatrix modification. The product of $l_{x,0}$ and the row vector $a_{0,x}$ (also $u_{o,x}$) is computed and subtracted from each row of the submatrix $a_{x,y}$, where $(1 \leq x, y \leq n-1)$.

Step 3. Steps 1 and 2 are recursively applied to the new submatrix generated in step 2. During the kth iteration, $l_{x,k}$ and $u_{k,y}$ $(k+1 \leq x, y \leq n-1)$ are generated. The iterations stop when $k = n-1$.

As shown in Fig. 3, our design consists of a divider, a column buffer, and p PEs. In each PE, there is a multiplier, an adder, and local memory. The maximum number of PEs, and their local memory size are limited by the available resources of the FPGA.

Our design uses four stages to complete the LU decomposition: "*matrix input*," "*column normalization*," "*submatrix modification*," and "*completion*" (Fig. 4). As introduced in the Dolittle LU decomposition algorithm, stages 2 and 3 are executed iteratively until the submatrix becomes a scalar. To fill the deep pipelines of floating-point units, our design uses a streaming architecture. Data are streamed from the memory, through the arithmetic engines for computation, and stored back to the memory in each stage. The data path configurations are different for different stages. To maximally reuse the expensive floating-point units and memory, these components are connected by high-speed switches.

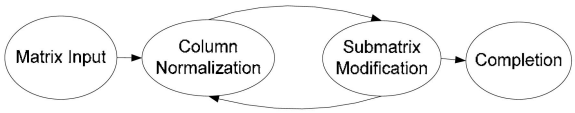

Fig. 4. LU design operation stages.

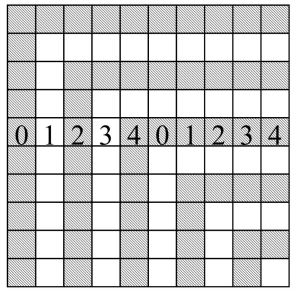

Fig. 5. Matrix mapping on FPGAs.

At the "matrix input" stage, the input data and address ports of the PE local memory are connected to the PEs' local memory input ports. These local memories appear to the host as a big memory block by address mapping. The matrix is striped to PEs by column, with column $p * j$ stored in PE j as shown in Fig. 5, where p is the total number of PEs. Because the submatrices become smaller in the iterative stages, such a storage format ensures that the submatrices are evenly distributed among the PEs for parallel computation. Without loss of generality, we assume that the matrix size is an integer multiple of p.

At the "*column normalization*" stage, col_0 is streamed out of its local PE storage through the divider to compute $a_{k,0}/a_{0,0}$ $(0 < k \leq n-1)$. The computed results are l_0 and are stored in the col buffer and the PE's local memory at the same time. In the "*submatrix modification*" stage, $a_{x,y} - l_{x,0}a_{0,y}$ needs to be computed for $1 \leq x, y \leq n-1$. l_0 streams through all the multipliers in the PEs simultaneously, while its address flows through the PEs' local memory to address $a_{x,y}$ and $a_{o,y}$. Because multiple columns are stored in one PE, this data flow must circulate multiple times until all the columns are updated. The "*column normalization*" and "*submatrix modification*" stages can be overlapped to shorten the overall runtime. For the "*submatrix modification*" stage, the data flow configuration inside a PE is shown in Fig. 6.

Design analysis. The proposed design completes LU decomposition in approximately $n^3/3p$ steps. Just n additional words are needed for the storage besides the original matrix.

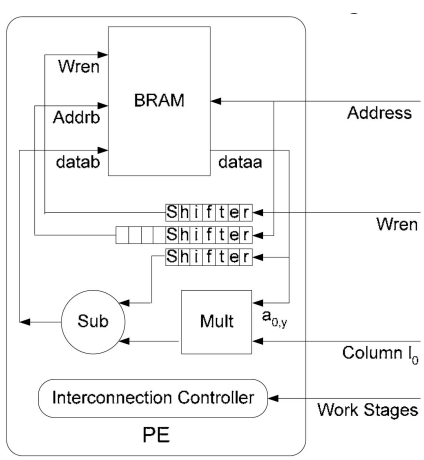

Fig. 6. Data flow configuration at submatrix modification stage.

TABLE 6
LU Implementation without Pivoting

Design	Double (s52e11)	s31e8	s16e7
PE number	8	16	32
Max size	128	256	256
Achievable Frequency	120MHz	130MHz	140MHz
Slices	20422(86%)	19070(80%)	19575(82%)
BRAMs	68 (29%)	148 (63%)	97(41%)
MULT18X18	128 (55%)	64 (27%)	32(13%)

TABLE 7
LU Implementation with Pivoting

Design	Double (s52e11)	s31e8	s16e7
PE number	8	16	32
Max size	128	128	256
Achievable Frequency	120MHz	130MHz	140MHz
Slices	21044 (89%)	20356 (86%)	20907 (88%)
BRAMs	84 (36%)	84 (36%)	130 (56%)
MULT18X18	128 (55%)	64 (27%)	32(13%)

Proof. In iteration k $(0 \leq k < n-1)$, the submatrix size is $n-k$. The number of steps needed by the divider to compute $l_{k,x}$ $(k \leq x < n-1)$ is $(n-k-1)$, while that for multiplication and subtraction is $2(n-k-1)^2$. Therefore, the total divider operation is $\sum_{x=1}^{n-1} x = n(n-1)/2$, and that for multiplication and subtraction is $\sum_{x=1}^{n-1} 2x^2 = 2n^3/3 - n^2$. The multiplication and subtraction are overlapped and computed by p PEs in parallel, so the total time is roughly $n(n-1)/2 + (2n^3/3 - n^2)/2p \approx n^3/3p$. The addition and multiplication floating-point operations for LU decomposition are on the order of $O(2n^3/3)$. Our design utilizes the parallelism of p adder and multiplier floating-point units and achieves performance on the order of $O(n^3)$.□

We implement the LU design on the Xilinx XC2VP50-7 FPGA with ISE 8.2. By using a similar amount of slices, we implemented eight PEs for the double-precision design, 16 PEs for single precision, and 32 PEs for the customized format s16e7. The achievable frequency listed in Table 6 is the highest frequency for our design running on the Cray-XD1 supercomputer located at the Oak Ridge National Laboratory. The resource costs are reported from synthesis tools.

4.2 Design for Pivoting

When zeros exist on submatrix diagonals, elements will be divided by zeros in the LU decomposition algorithm discussed above. At the same time, because computers have to use certain precisions, relatively small values on submatrix diagonals will possibly cause big accumulated numerical errors [28]. Pivoting is a process performed on a matrix to increase numerical stability. The element $a_{0,0}$ used in the "*column normalization*" stage is called a pivot element. The row having the pivot element is called the pivot row.

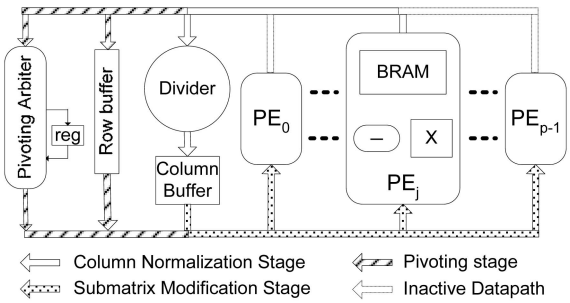

Fig. 7. LU decomposition with pivoting.

The system diagram with partial pivoting is shown in Fig. 7. During the "*pivoting*" stage, the first submatrix column is streamed out from the PEs to the pivoting arbiter, which compares the pivot element with other values in this column. If the pivot element is the biggest value in the column, no pivoting is required. Therefore, the state machine transfers to the next stage "*column normalization*" directly to avoid extra execution time. If the pivoting arbiter finds a value in column 0 bigger than the pivot element, a pivoting operation has to be performed. The biggest element in column 0 is the new pivot element, while the row that has the new pivot element will be the new pivot row. Because matrix A is stored in the PEs' local memories by column, a row of matrix A is distributed in all PEs. The values in the old and new pivot rows are exchanged in all PEs simultaneously. The pivoting buffer is used to temperately store the old pivot row when exchanging two rows. Table 7 summarizes the implementation results of LU decomposition with pivoting.

4.3 Mixed-Precision Hybrid Direct Solver

The FPGA-based LU decomposer we proposed can be used for the mixed-precision algorithm, as shown in Fig. 8. A lower precision version of matrix A is moved from the CPU main memory to the FPGA for LU decomposition. The CPU computes the solution using lower precision LU matrices but calculates the residual and updates the solution in higher precision.

The execution time of our design in Fig. 8 consists of four parts: the time for the LU decomposition, iterative refinement, forward/backward triangular solver, and communication. As we discussed before, the clock cycles required for our LU design is $O(n^3/3p)$, where p is the number of PEs. The frequency f of the LU decomposition design depends heavily on the data formats. The communication time is associated with the data movements between the FPGA and the CPU main memory, so it is determined by the matrix size (n^2), data width (w), and bus bandwidth

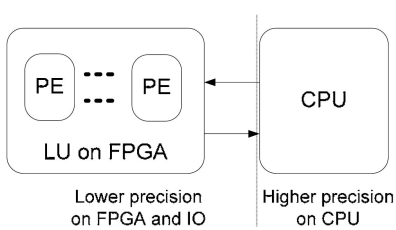

Fig. 8. Mixed-precision direct solver

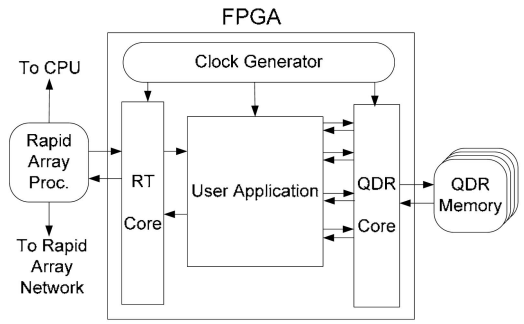

Fig. 9. Cray-XD1 FPGA organization [8].

(B_{bus}). Finally, the time for iterative refinement depends on the number of iterations (I_{ref}) and the time for each loop (T_{ref}). The total time can be described as

$$\begin{aligned} T_{mixed} &= T_{LU} + T_{comm} + T_{tri} + T_{refinement} \\ &= \frac{n^3}{3pf} + \frac{2n^2}{B_{bus}}w + T_{tri} + I_{ref}T_{ref}. \end{aligned} \tag{5}$$

Using lower precision data formats could significantly increase the number of PEs and their frequency as well as reduce the data width, so the first two terms will be greatly decreased. On the other hand, the number of refinement iterations would increase, as shown in Table 2. Because T_{ref} is relatively small, the impact from the third term is dominated by the first two terms. The computation for the triangular solver is $O(n^2)$, which is much less than that for the LU decomposition.

Suppose matrix A is factored to a lower triangular matrix L and upper triangular matrix U, the linear system becomes $LUx = b$. It is equivalent to solving two linear equations $Ly = b$ and $Ux = y$. Since L and U are triangular matrices, y and x can further be solved by forward and backward substitutions, which are shown in the following equations:

$$\begin{aligned} y_0 &= b_0/l_{0,0}, \\ y_1 &= (b_1 - l_{1,0}y_0)/l_{1,1}\cdots, \\ y_{n-1} &= \left(b_{n-1} - \sum_{j=0}^{n-2} l_{n-1,j}y_j\right)/l_{n-1,n-1}, \end{aligned} \tag{6}$$

$$\begin{aligned} x_{n-1} &= y_{n-1}/u_{n-1,n-1}, \\ x_{n-2} &= (y_{n-2} - u_{n-2,n-1}x_{n-1})/u_{n-2,n-2}\cdots, \\ x_0 &= \left(y_0 - \sum_{j=n-1}^{1} u_{0,j}x_j\right)/u_{0,0}. \end{aligned} \tag{7}$$

Equations (6) and (7) could be implemented on FPGAs, but complicated control logic is required to achieve fine-grained parallelism. Furthermore, the division operation of each iteration cannot be easily parallelized and requires an expensive floating-point divider. On the other hand, the computational complexity for (6) and (7) is only $O(n^2)$, while that for the LU decomposition is $O(n^3)$. For an $n \times n$ matrix, the computational load for LU decomposition is n times that for forward and backward solvers. Therefore, we propose to explore LU decomposition on FPGAs but leave the forward and backward substitutions on the CPU, as shown in Fig. 8.

5 Test on Supercomputer

5.1 Design on Cray-XD1 Supercomputer

The Cray-XD1 supercomputer incorporates reconfigurable computing abilities with accelerators to deliver significant speedup of targeted applications [12], [31]. The basic architectural unit of the Cray-XD1 system is the Cray-XD1 chassis, which can contain one to six computation blades. Each computation blade includes two 64-bit AMD Opteron processors configured as a two-way symmetric multiprocessor (SMP) that runs Linux. Each compute processor can be assigned 1- to 8-Gbyte DDR. FPGAs can be included as coprocessors by adding an expansion module on the computation blade. As shown in Fig. 9, the processors, FPGAs, and memory are linked within a chassis and between chassis by a high-speed switch fabric called the RapidArray interconnect. Besides the main memory, each FPGA module contains four QDR II SRAMs for high-speed storage. The programmable clock enables the user to set the speed of the FPGAs [8]. The Cray-XD1 machine at ORNL (Tiger) has 12 chassis containing 144 Opteron processors and 6 Xilinx FPGAs (XC2VP50-7).

Fig. 10 shows our mixed-precision hybrid direct solver architecture on the Cray-XD1 supercomputer. The lower precision LU decomposer is mapped to the FPGA for high performance, while the forward/backward solver and iterative refinements are mapped to the Opteron processor

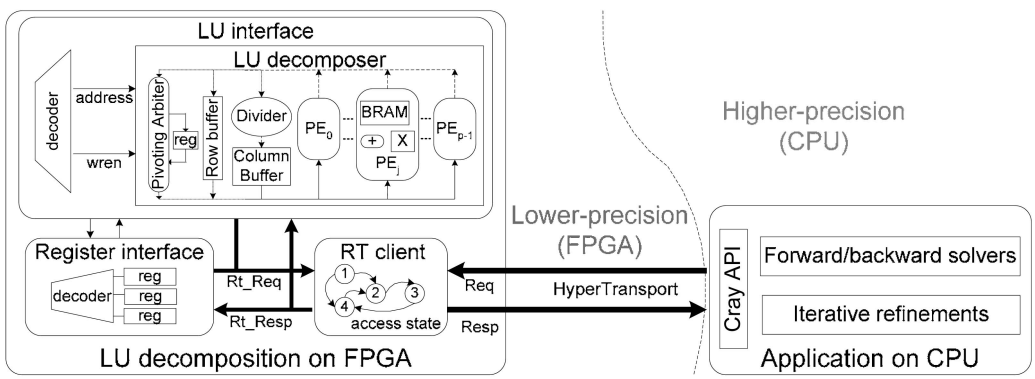

Fig. 10. Hardware structure on Cray-XD1.

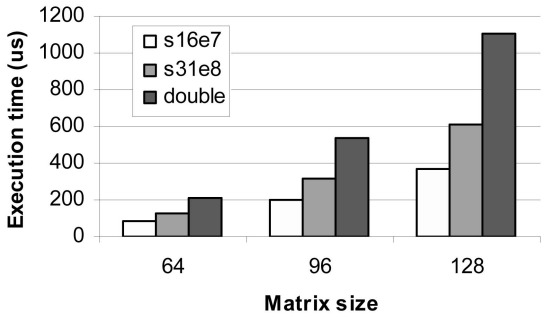

Fig. 11. Performance comparison of LU design.

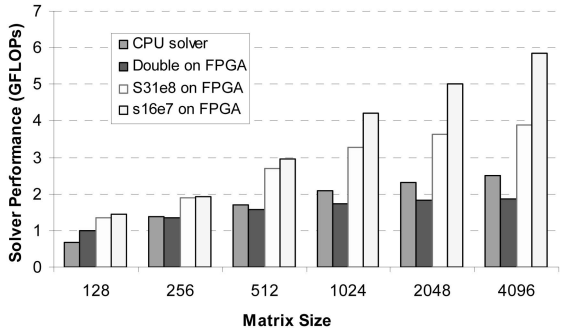

Fig. 13. Gigaflops performance of direct solvers.

in higher precision. The user application on the FPGA consists of three components: the RT client, the LU interface, and the register interface. The RT client is the bridge between the fabric and the user application. The basic structure of the RT client is primarily a state machine. It interprets requests and assigns jobs to different components according to the request addresses. The register interface provides the interface to read and write registers, which are used to control and monitor the status of the LU decomposer.

5.2 Performance on Cray-XD1

We are interested in the performance comparison of our design with 2.2-GHz Opteron processors on the Cray-XD1 machine. The software codes on CPUs run the same algorithm as the FPGA designs in double precision.

Our work accelerates the performance of direct solvers by mapping LU decomposition onto FPGAs and taking advantage of the high performance of lower precision arithmetic. Therefore, we first profile the performance of our LU decomposition designs with different data formats. As shown in Fig. 11, the LU decomposition time for lower precision designs is much less than that for higher precision designs.

We further test the performance of our mixed-precision direct solver design. As shown in (5), the execution time for our mixed-precision design consists of four components: LU decomposition, iterative refinement, forward/backward triangular solver, and communication. The profiled execution time for a 128×128 matrix is shown in Fig. 12. As expected, the time for both computation and communication is reduced rapidly for lower precision arithmetic. The time for backward/forward solvers and iterative refinements occupies just a small portion of the complete direct solver algorithm but appears to be relatively long in Fig. 12. The reason is that the time for LU decomposition is significantly reduced by using our FPGA accelerator.

6 Performance

Finally, we compare the performance of our design with CPUs. Pivoting is considered for both FPGA and CPU implementations. We test the performance of CPUs by using ATLAS [29], an optimized BLAS library, on a 2.2-GHz Opteron processor of the Cray-XD1 supercomputer. For large matrices, which cannot be accommodated by FPGA on-chip memory, we propose to use QDR memory in Fig. 9. A clock-cycle-accurate model [27] is built to predict the performance of our mixed-precision direct solver. As shown in Fig. 13, lower precision designs have higher performance by taking advantage of additional parallelism and higher frequency. The performance of the lower precision design s16e7 is about two to three times better than the double-precision design and CPUs.

For the platform used with this research, the Cray-XD1, the FPGA included on the nodes (Xilinx XC2VP50) is an older technology than the Opteron processors included on the nodes. Given the fact that the floating-point performance of FPGAs increases much faster than CPUs [2], we expect that the gigaflops performance of newer FPGAs to result in even better speedups.

For newer FPGAs, we are able to implement more computational logic at higher frequency. Fig. 14 predicts the

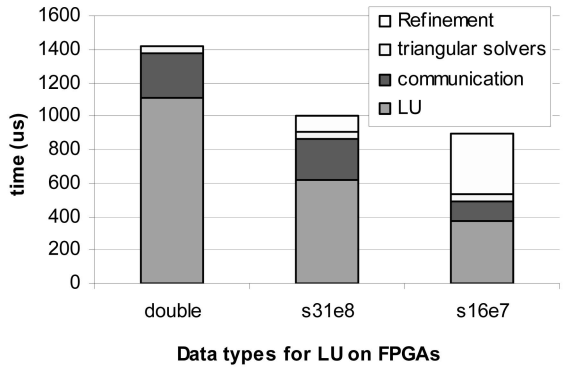

Fig. 12. Execution time for mixed-precision direct solvers.

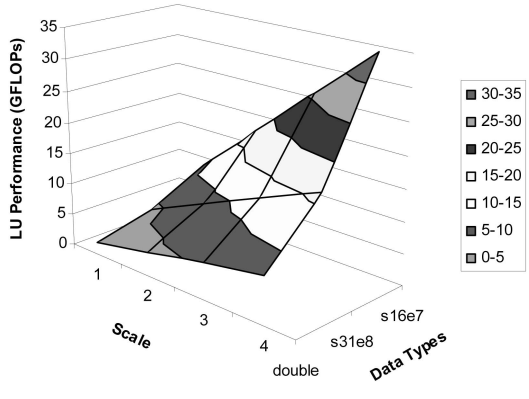

Fig. 14. Prediction of LU performance.

performance of our LU decomposer using our clock-cycle-accurate performance model [27], where "*scale*" describes a multiplicative scaling factor for the number dividers, and PEs can be accommodated compared to the XC2VP50 FPGA on the Cray-XD1. For example, with s16e7 data representation, about four times as many divider units and PEs can be accommodated by a Xilinx XC4VLX200 FPGA contemporary to the Opteron processor. Although in Fig. 14, we conservatively assume the same frequency as before, increased FPGA clock rates will further increase the LU decomposer performance.

7 Conclusions

This paper proposes a mixed-precision linear solver design on FPGAs to achieve both the high performance of lower precision arithmetic and the accuracy of higher precision at the same time. First, we introduce mixed-precision linear solver algorithms and give an error analysis. Second, we give the design of a mixed-precision LU decomposer and direct solver for FPGAs. To the best of our knowledge, this is the first work to implement a hardware architecture for the pivoting algorithm. Our work is also the first mixed-precision linear solver design on FPGAs. Finally, we implement our design on the Cray-XD1 supercomputer and compare the performance to CPUs. This paper discusses mixed-precision design with respect to floating-point performance; our future work will address power efficiency.

Acknowledgments

This project was supported by the University of Tennessee Science Alliance and the Oak Ridge National Laboratory Director's Research and Development program. The authors also would like to thank Jack Dongarra and Stanimire Tomov of the University of Tennessee for useful discussions on mixed-precision algorithms and the anonymous reviewers for their comments.

References

[1] J.W. Demmel, *Applied Numerical Linear Algebra*. SIAM Press, 1997.
[2] K.D. Underwood, "FPGAs versus CPUs: Trends in Peak Floating-Point Performance," *Proc. ACM/SIGDA 12th Int'l Symp. Field Programmable Gate Arrays (FPGA '04)*, Feb. 2004.
[3] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems," *Int'l J. High Performance Computer Applications*, vol. 21, pp. 457-466, 2007.
[4] R. Strzodka and D. Göddeke, "Mixed Precision Methods for Convergent Iterative Schemes," *Proc. Workshop Edge Computing Using New Commodity Architectures (EDGE '06)*, May 2006.
[5] R. Strzodka and D. Göddeke, "Pipelined Mixed Precision Algorithm on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components," *Proc. 14th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '06)*, May 2006.
[6] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating Double Precision FEM Simulations with GPUs," *Proc. 18th Symp. Simulation Technique (ASIM '05)*, Sept. 2005.
[7] J.H. Wilkinson, *The Algebraic Eigenvalue Problem*. Clarendon, 1965.
[8] Cray, Inc., *Cray XD1 FPGA Development*, 2005.
[9] C.B. Moler, "Iterative Refinement in Floating Point," *J. ACM*, vol. 2, pp. 316-321, 1967.
[10] J. Demmel, Y. Hida, W. Kahan, X.S. Li, S. Mukherjee, and E.J. Riedy, "Error Bounds from Extra Precise Iterative Refinement," Technical Report UCB/CSD-04-1344, LAPACK Working Note 165, Aug. 2004.
[11] G.W. Stewart, *Introduction to Matrix Computations*. Academic Press, 1973.
[12] Cray Inc., http://www.cray.com, 2008.
[13] J. Sun, G. Peterson, and O.O. Storaasli, "Sparse Matrix-Vector Multiplication Design on FPGAs," *Proc. 15th IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '07)*, Apr. 2007.
[14] J.W. Demmel, *Applied Numerical Linear Algebra*. SIAM Press, 1997.
[15] R. Barrett, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, second ed. SIAM, 1994.
[16] Xilinx Inc., http://www.xilinx.com, 2008.
[17] G. Govindu, R. Scrofano, and V.K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA '05)*, June 2005.
[18] X. Wang, M. Leeser, and H. Yu, "A Parameterized Floating-Point Library Applied to Multispectral Image Clustering," *Proc. Seventh Ann. Military and Aerospace Programmable Logic Devices (MAPLD '04) Int'l Conf.*, Sept. 2004.
[19] J. Sun, G. Peterson, and O.O. Storaasli, "Mapping Sparse Matrix-Vector Multiplication on FPGAs," *Proc. Reconfigurable Systems Summer Inst. (RSSI)*, 2007.
[20] L. Zhuo and V.K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," *Proc. ACM/SIGDA 13th Int'l Symp. Field-Programmable Gate Arrays (FPGA '05)*, pp. 63-74, Feb. 2005.
[21] K. Underwood, S. Hemmert, and C. Ulmer, "Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications," *Proc. ACM/IEEE Conf. Supercomputing (SC '06)*, Nov. 2006.
[22] L. Zhuo and V.K. Prasanna, "High Performance Linear Algebra Operations on Reconfigurable Systems," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2005.
[23] H. Bowdler, R. Martin, G. Peters, and J. Wilkinson, "Handbook Series Linear Algebra: Solution of Real and Complex Systems of Linear Equations," *Numerische Math.*, vol. 8, pp. 217-234, 1966.
[24] J. Demmel, M. Heath, and H. van der Vorst, "Parallel Numerical Linear Algebra," *Acta Numerica*, pp. 111-198, 1993.
[25] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, second ed. SIAM Press, 2002.
[26] J. Sun, "Obtaining High Performance via Lower-Precision FPGA Floating Point Units," *Proc. ACM/IEEE Conf. Supercomputing (SC '07)*, Nov. 2007.
[27] J. Sun, "High Performance Reconfigurable Computing for Linear Algebra: Design and Performance Analysis," PhD dissertation, Univ. of Tennessee, 2008.
[28] G.H. Golub and C.F. Loan, *Matrix Computations*, third ed. Johns Hopkins, 1996.
[29] R. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, pp. 3-35, 2001.
[30] Y. Gu, T. VanCourt, and M. Herbordt, "Improved Interpolation and System Integration for FPGA-Based Molecular Dynamics Simulations," *Proc. 16th Int'l Conf. Field Programmable Logic and Applications (FPL)*, 2006.
[31] Y. Bi, G.D. Peterson, L. Warren, and R. Harrison, "Hardware Acceleration of Parallel Lagged-Fibonacci Pseudo Random Number Generation," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms*, June 2006.

Junqing Sun received the BS and MS degrees from Tongji University, Shanghai, in 2001 and 2004, respectively, and the PhD degree, with a cumulative GPA of 4.0, from the University of Tennessee, Knoxville, in 2008. He is currently with the Department of Electrical Engineering and Computer Science, University of Tennessee. His research interests include computer architecture, digital systems, control and communication systems, and probability. He received the first place award of the ACM Student Research Competition at Supercomputing 2007. He is a stuedent member of the IEEE and the ACM.

Gregory D. Peterson received the BS, MS, and DSc degrees in electrical engineering and BS and MS degrees in computer science from Washington University, St. Louis. He is an associate professor in the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. He worked at the Air Force Research Laboratory for four years, where he received the Research and Development Award, recognizing him as the best research and design engineer in USAF for 1998. His research interests include parallel processing, electronic design automation, performance evaluation, and high-performance reconfigurable computing. He is a senior member of the IEEE and a member of the ACM.

Olaf O. Storaasli received the PhD degree in engineering mechanics from North Carolina State University. During his long career at NASA, he also served on the Graduate Faculties of George Washington and Christopher Newport Universities. In 2005, he joined the Future Technologies Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee. He is currently a distinguished research scientist exploring new algorithms and architectures to harness the power of field-programmable gate arrays (FPGAs) to accelerate future scientific supercomputing applications (including rapid solution of large matrix equation systems). He is also a research affiliate in the Joint Institute for Computer Science, University of Tennessee, Knoxville.