Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver

WEI ZHANG, VAUGHN BETZ, and JONATHAN ROSE, University of Toronto

FPGAs have the potential to serve as a platform for accelerating many computations including scientific applications. However, the large development cost and short life span for FPGA designs have limited their adoption by the scientific computing community. FPGA-based scientific computing and many kinds of embedded computing could become more practical if there were hardware libraries that were portable to any FPGA-based system with performance that scaled with the size of the FPGA. To illustrate this idea we have implemented one common super-computing library function: the LU factorization method for solving systems of linear equations. This paper describes a method for making the design both portable and scalable that should be illustrative if such libraries are to be built in the future. The design is a software-based generator that leverages both the flexibility of a software programming language and the parameters inherent in an hardware description language. The generator accepts parameters that describe the FPGA capacity and external memory capabilities. We compare the performance of our engine executing on the largest FPGA available at the time of this work (an Altera Stratix III 3S340) to a single processor core fabricated in the same 65nm IC process running a highly optimized software implementation from the processor vendor. For single precision matrices on the order of 10,000 $\times$ 10,000 elements, the FPGA implementation is 2.2 times faster and the energy dissipated per useful GFLOP operation is a factor of 5 times less. For double precision, the FPGA implementation is 1.7 times faster and 3.5 times more energy efficient.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Algorithms implemented in hardware

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: FPGA, linear system solver, acceleration, portable, scalable, LU decomposition

ACM Reference Format:
Zhang, W., Betz, V., and Rose, J. 2012. Portable and scalable FPGA-based acceleration of a direct linear system solver. ACM Trans. Reconfig. Technol. Syst. 5, 1, Article 6 (March 2012), 26 pages.
DOI = 10.1145/2133352.2133358 http://doi.acm.org/10.1145/2133352.2133358

1. INTRODUCTION

As the logic and computational capacity of FPGAs have grown, FPGAs have become a potential platform for accelerating many computations including those in scientific and embedded applications. The high level of parallelism and abundant flexibility available in the FPGA fabric offer the promise of significant speed-up. A number of vendors offer platforms that enable a processor to offload computation to an FPGA-based accelerator including XtremeData [XtremeData, Inc 2008], SRC [SRC Computers 2008], and Cray [Cray Inc 2008]. However, adoption of these FPGA accelerators by the computing community has been limited because the creation of an FPGA design is

The authors are grateful to Altera and an NSERC CRD grant for funding their research, as well as an NSERC postgraduate scholarship.
Authors' address: Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada; email: jonathanscottrose@gmail.com.

DOI 10.1145/2133352.2133358 http://doi.acm.org/10.1145/2133352.2133358

difficult and time consuming and outside the skill set of the typical software engineer. In addition, once a design has been created for one specific FPGA chip and board, the same design cannot be easily transferred to another. The design is typically locked onto the original FPGA-based platform in two ways: it has a specific off-chip memory architecture and a specific on-chip computational fabric architecture. Consequently as newer, more powerful FPGA systems become available the design rapidly becomes outdated.

In contrast, software is highly portable. Once a software application is completed, it could easily be upgraded to new and faster machines and obtain significantly better performance; in modern times this also requires that the code be inherently parallelizable across cores to permit performance scaling. In this case software engineers can develop and maintain rich libraries that solve important problems. Common software libraries for scientific computing include matrix manipulation packages such as BLAS [Blackford et al. 2002], SAT solvers, and linear program solvers. Scientific and other computing users need not be highly skilled in creating optimized code because they can simply use the functions in these libraries. In hardware, IP cores do allow some design reuse, but at a much lower level of abstraction than with high level software libraries.

One method that attempts to make FPGA programming more accessible is to employ high-level languages and synthesis tools that map software directly to an FPGA. Examples include Handel-C [Agility Design Solutions, Inc 2008], Catapult C [Mentor Graphics 2008], and AutoPilot [AutoESL 2008]. However, this approach is often not adequate to create an efficient hardware design from complex code as the programmer typically has to write the code in a stylized manner with the final architecture of the system in mind to obtain good performance.

In this work, we propose an alternative solution for making FPGA-based computation more accessible: the creation of a computational "library" that is portable to any FPGA platform with minimal effort. The second key feature of the library is that its performance scales with the capabilities and resources of the FPGA. Given an FPGA with more capacity and faster elements, the library performance should improve without extra effort from the designer. The creation of a *portable* and *scalable* library, would drastically reduce the development cost and increase the life span of the design, thus making it more attractive to designers of compute-intensive applications. The goal of this paper is to illustrate the how the concepts of portability and scalability could be implemented for one example common computation and to measure the resulting performance, energy consumption, cost and design effort.

Our focus application is solving systems of linear equations, as this is a very common problem and the computation time is high for large systems. We are limiting our scope to dealing with only nonsingular matrices, which has a nonzero determinant and only one solution. There are two main classes of linear equation solvers: iterative and direct [Hager 1988]. Iterative solvers begin with an initial guess for the solution vector and then refine it until the error is sufficiently small. Direct solvers manipulate the matrix and solution vector until the solution can be easily computed. For a nonsingular matrix, a direct solver will always compute the solution. Iterative solvers often require less computation but do not guarantee convergence for all types of matrices or require the same order of computation as direct solvers to guarantee convergence.

Both iterative and direct solvers are widely used. Direct solver are typically used for dense matrices, which are matrices mainly nonzero coefficents, and iterative solvers are typically used for sparse matrices, which are matrices that have a lot of zero coefficients. Prior work [Zhuo and Prasanna 2005] on iterative solvers has not resulted in significant speed-up over processors due to the large memory bandwidth requirements; thus, we focus on direct methods and solving dense nonsingular matrices. We have created a generator that automatically creates a portable and scalable FPGA compute engine

for the LU factorization method [Dongarra et al. 1998] to solve a linear system. The generator and engine are highly parameterized to permit any size of matrix (up to the external memory capacity) and to make use of any size of FPGA.

An earlier version of this work appeared in Zhang et al. [2008]. In that work we presented only the performance of single-precision floating point compute engines from our generator and compared it to the performance from software. In this article, we provide performance results for both the single and double precision floating point compute engines and compare it to software for both. We also study the effect of various key generator parameters on performance and describe competing trends that influence performance as the limit of resources on the FPGA is reached. In addition, we compare the performance results of two different generations of FPGA and show how performance scales across FPGA generations. Finally, we discuss the environment we created to enable the development of a portable and scalable engine, and the extra design effort required to achieve portability and scalability. These extended discussions and results are based on Zhang [2008].

This article is organized as follows. Section 2 provides background on the LU factorization method for solving linear systems and summarizes previous work using FPGAs to accelerate matrix operations. Section 3 outlines the architecture of our design. Section 4 describes how we achieve portibility and scalability via the use of parameters that describe the FPGA capabilities, and Section 5 describes the tool flow that can generate implementations for a wide range of these parameters, and the extra development time needed to create a parameterized design. Section 6 analyzes the effect on performance of the various parameters. Section 7 discusses the experimental results for both single precision and double precision engines. Section 8 outlines limitations of the engine and future work and Section 10 concludes.

2. SOLUTIONS OF SYSTEMS OF LINEAR EQUATIONS

A system of linear equations is often represented in a matrix and vector form as $Ax = b$. The coefficients of the variables in each linear equation are represented in each row of an N $\times$ N matrix (A) multiplied by the N-element vector of unknown variables (x). A solver must determine the values of x for which the product generates the N-dimensional constant (b). The LU factorization method directly solves for x by breaking the coefficient matrix into two matrices, forming $LUx = b$ [Hager 1988]. One of those matrices, called L, is a lower triangular matrix which has the diagonal elements equal to 1 and all elements above the diagonal equal to 0; the other matrix, called U, is an upper triangular matrix which has the elements below the diagonal equal to 0. If we set $y = Ux$, a forward substitution can be performed to compute y from $Ly = b$. Then a backward substitution can be performed to compute x from $Ux = y$. The most time consuming computation in this algorithm is the factorization of the coefficient matrix, which is the determination of the matrices L and U such that $A = LU$, as this requires $O(N^3)$ operations.

2.1. Simple LU Factorization

A pseudocode for a simple LU factorization algorithm is given in Algorithm 1 [Hager 1988]. There are two kinds of operations that must to be performed: the first is the division of all the elements below the diagonal in the column, $a_{k+1,k}$ to $a_{N,k}$, by the diagonal element, $a_{k,k}$. The second is the multiplication of column elements, $a_{k+1,k}$ to $a_{N,k}$, by row element, $a_{k,j}$, and the subsequent subtraction of the result from the column elements below the row element, $a_{k+1,j}$ to $a_{N,j}$. The multiplication and subtraction is repeated for j from $k+1$ to N. All the operations are repeated for the next diagonal element until the last diagonal element is reached.

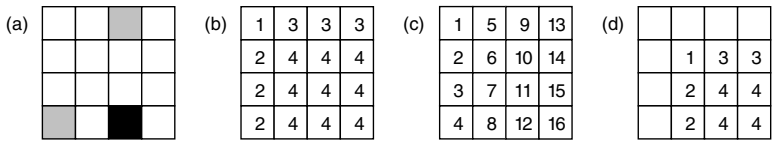

Fig. 1. (a) blocks required, in gray, to update the black block; (b) type of computation for each matrix block in the first block pass; (c) order of blocks updated in the first pass; (d) computation performed in the second block pass.

ALGORITHM 1: Pseudo-code for a simple LU factorization

```
for k = 1 to N − 1 do //for each diagonal element
    for i = k + 1 to N do //for each element below it
        a_{i,k} = a_{i,k}/a_{k,k}; //normalize
    end
    for j = k + 1 to N − 1 do //for each column right of current diagonal element
        for i = k + 1 to N do //for each element below it
            a_{i,j} = a_{i,j} − a_{i,k} × a_{k,j};
        end
    end
end
```

Partial pivoting is a technique that can be used in LU factorization to improve numerical stability, particularly for matrices that have very small diagonal values. It can also allow the algorithm to solve certain matrices that have a zero in the diagonal element used for normalization. The technique involves exchanging rows of the matrix so the diagonal element has the maximum absolute value before it is used to normalize the column. To reduce the complexity of the design, our LU factorization algorithm does not do any pivoting. It is a feature we plan on adding to the design as a future work.

2.2. Block LU Factorization

For the simple LU factorization method described previously, all of the elements in the matrix must be accessible during the computation. For many scientific computing problems, the matrix size (N) is at least 10,000 $\times$ 10,000 single-precision numbers, which requires roughly 0.4GBytes of memory. This is far too large to store on a chip, either an FPGA or a processor's cache, and therefore, the matrix must be stored in off-chip memory. Thus all practical approaches must deal with the fact that off-chip memory bandwidth is limited. The common approach to deal with this is to performed the computation in a "blocked" manner: to bring on-chip subsections of the coefficient matrix A, each of size $N_b \times N_b$ and perform as many computations on that data as possible to minimize the number of times the data have to be fetched from off-chip memory.

There are three common variants of the block LU factorization [Dongarra et al. 1998]; and we will employ the "right-looking" version; in this method, the current block being updated uses elements from the leftmost and topmost block in its row and column, as shown in Figure 1(a). With this blocking method, there are four kinds of computations that can be performed on the blocks.

Case 1. all three blocks (current, left-most, and topmost) are the same physical block.
Case 2. the current block is the same as the leftmost block.
Case 3. the current block is the same as the topmost block.
Case 4. all three blocks are different.

Figure 1(b) shows an example matrix in which blocks are labeled with each case. The computation for these blocks are similar to the simple LU factorization algorithm described in Algorithm 1, except the loop indices are different and some elements obtained from the left-most and top-most blocks are required. The pseudocode for all the cases is shown in Algorithm 2, where $a_{i,j}$, $l_{i,j}$, and $u_{i,j}$ represent elements in the current block, left-most block and top-most block respectively. For case 1, the operations performed are the same as the simple LU factorization, except N is replaced by N_b. The two cases where current block is also the left-most block (case 1 and 2) need to perform division operations. The other two cases (case 3 and 4) perform only the multiplication and subtraction, as an earlier block operation will have already normalized the necessary elements. For a large matrix, case 4 is the most common and dominates the computation time. The blocks are updated in the order shown in Figure 1(c). After the first block pass in which every block is updated, the blocks in the first block column and block row have the final solution. The remaining blocks, which were all the case 4 blocks in the first block pass, are updated again, repeating these computations as if it is a new matrix, as shown in Figure 1(d). This process repeats until no blocks are left, requiring N/N_b block passes.

ALGORITHM 2: Code for all 4 cases of block LU factorization

```
Case 1: same as simple LU factorization with N_b instead of N; //See Algorithm 1
Case 2: for k = 1 to N_b do //for each diagonal element in top-most block (u)
    for i = 1 to N_b do //for each element below it in current block (a)
        a_{i,k} = a_{i,k} / u_{k,k}; //normalize
    end
    for j = k + 1 to N_b do //for each column right of current diagonal element
        for ) do //for each element below it in current block (a)
            i = 1 to N_b
        end
        a_{i,j} = a_{i,j} − a_{i,k} × u_{k,j};
    end
end
Case 3: for k = 1 to N_b do //for each column in left-most block (l)
    for j = 1 to N_b do //for each column in current block (a)
        for i = k + 1 to N_b do //for each element below it
            a_{i,j} = a_{i,j} − l_{i,k} × a_{k,j};
        end
    end
end
Case 4: for k = 1 to N_b do //for each column in left-most block (l)
    for j = 1 to N_b do //for each column in top-most block (u)
        for i = 1 to N_b do //for each element below it in current block (a)
            a_{i,j} = a_{i,j} − l_{i,k} × u_{k,j};
        end
    end
end
```

2.3. Prior Work

Prior research on implementing linear equation solvers in FPGAs has generally focused on acceleration of iterative solvers. Zhuo and Prasanna [2005], deLorimier and DeHon [2005], Morris and Prasanna [2007], and Lopes and Constantinides [2008] built iterative solvers using the conjugate gradient method [Dongarra et al. 1998]. Morris and Prasanna [2007] report a speed-up of 2.4 using the Virtex II 6000 over a

2.8 GHz Xeon processor. They also implemented a Jacobi iterative solver, which achieved a speed-up of 2.2 using the same hardware. In all these prior works, except for Zhuo and Prasanna [2005], these solvers imposed a limit on the matrix size based on the on-chip memory capacity of the FPGA. Since the input matrix can be stored on the FPGA, the memory bandwidth required can be amortized across all the iterations of the algorithm. For Zhuo and Prasanna [2005], blocks of the matrix are loaded and computations performed on them before another block is brought on chip. The performance is limited by the memory bandwidth as only N^2 computation can be performed on each block, which contains N^2 data, thus requiring a large amount of memory access per unit of computation.

Zhuo and Prasanna [2006] implemented the same LU factorization method employed in our work. It reported a speed-up of about 1.2 in double precision using a Virtex-II Pro XC2VP100 over a 2.2 GHz Opteron (with a performance of just under 4 GFLOPS). This work also imposed a limit on the matrix size; a blocking version to remove the matrix size limit was proposed in Daga et al. [2004] and implemented in Zhuo and Prasanna [2008]. The design used the original LU factorization component and an additional matrix multiply component to perform the blocking algorithm, but due to the bottleneck imposed by the matrix multiplier, overall performance decreased. For matrices of size up to 1000 in double precision, LU factorization had a performance of 2.8 GFLOPS on a Virtex-II Pro XC2VP100. This was slower than software running on a 2.2 GHz Opteron which had a performance of 3.3 GFLOPS. [Zhuo and Prasanna 2008] also identified parameters in the operation that allow them to scale their design, similar to what we have done. However, not much is mentioned in terms of portability, such as how to handle different external memory or the challenges of moving to other FPGAs. Many previous works do not mention external memory, and some simply provide a bound on the required memory bandwidth. In contrast, this paper explicitly considers external memory and outlines how portability and scalability can be achieved for different FPGAs with different external memories.

Another method to improve performance is to use a lower precision during part of the algorithm than the standard single or double precision. Sun et al. [2008] implemented a mixed precision direct solver. A lower precision LU factorization is performed on the FPGA and then an iterative refinement algorithm is performed on the CPU to get the solution with a sufficient accuracy. They were able to get 2 to 3 times better performance when using the mixed precision method over a double precision design.

3. HARDWARE IMPLEMENTATION

3.1. High-Level Design Overview

Our goal is to create a highly parameterized LU factorization hardware design for an FPGA (and a software generator for automatically creating these designs) for floating-point matrices. The matrices most in need of solution acceleration are very large; thus, a key feature of our approach will be to employ large off-chip memories to store the input matrix. We will use the block LU factorization method described in Section 2.2, where blocks of the large matrix are brought into on-chip memory and processed separately to make most efficient use of off-chip memory bandwidth.

We will assume that the matrix is square and restrict the on-chip matrix blocks to be square. The result of the LU factorization will be stored in the same location as the input matrix on the off-chip memory.

Figure 2 shows a high level diagram of the design, which performs two main functions: The first is called *data marshalling*, which is the loading and storing of matrix blocks onto the FPGA from the external memory. The second function is the actual computation on each set of blocks brought into the FPGA.

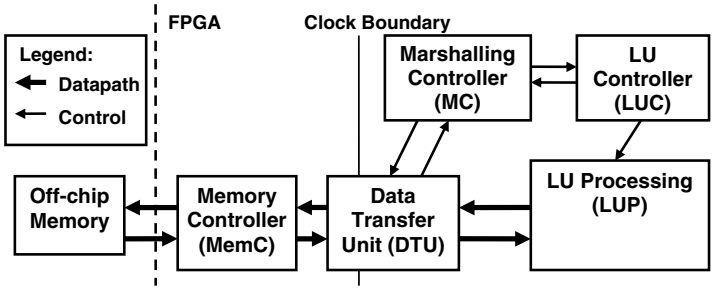

Fig. 2. Block diagram of Linear System Solver.

The data marshalling is handled by the Data Transfer Unit (DTU) and the Memory Controller (MemC) modules, as shown in Figure 2. The computation is performed by the LU Processing (LUP) module which is controlled by the LU Controller (LUC) module. The Marshalling Controller (MC) is responsible for issuing commands to these modules and to provide synchronization between the major tasks. The MC controls which blocks of memory to load and store and the series of operations to perform on the loaded blocks to complete the LU factorization.

There are two clocks in this design: one clock controls the speed of the external memory and the directly connected part of the data marshalling hardware; the second clock controls the computation itself. Separating these clocks is important for portability; it is unlikely that the speed of the off-chip memory and that of the on-chip memory and computation units will scale at the same rate as new generations of FPGAs and memories are developed.

This design illustrates one aspect of the design devoted to portability: it can be used for different FPGA boards with different memory units. By separating out the memory controller from the unit that requests memory blocks, we have made it more portable, as it should be fairly easy to connect in different memory controllers. (We expect that memory controllers from the same vendor would present almost identical interfaces to the inside of the chip, and similar interfaces across vendors).

3.2. Ordering of Blocks and Setup of Computation

As we developed our design, we noticed that when there are many processing elements (on the order of 50–100), the time required to transfer the matrix from off-chip memory onto the chip was on the same order of the computation itself. To achieve the best performance, it is thus necessary to simultaneously fetch data while performing the computation. This requires on-chip "double-buffering" of the matrix data in which one memory buffer is occupied with off-chip communication while the other is used in the computation.

The basic computation involves updating (i.e., performing all the computations for) a single block of the matrix. To update any given block, up to three blocks are required, as discussed in Section 2.2: the current block being computed, the topmost block in the same column and the left-most block in the same row. Following the order of updating shown in Figure 1(c), the topmost block is the same for the all blocks in the same column and we can reuse this block if the next block is in the same column. Therefore, this block only has to be loaded once per block column computation. At the beginning of a new column, this top-most block is also the current block. Thus in total, we only ever need to load a maximum of two blocks to perform any block computation, as we do not have to load the top-most block explicitly. This reduces the external memory bandwidth required to sustain the computation; a total of 5 matrix blocks in on-chip

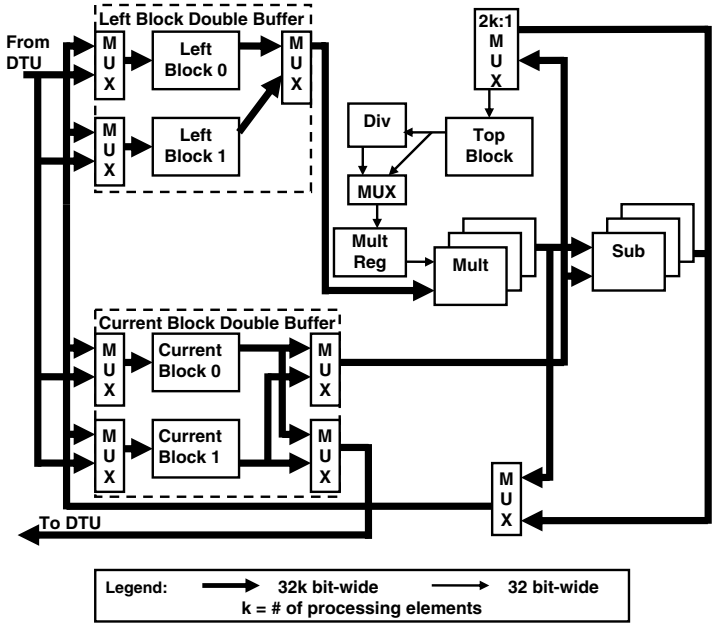

Fig. 3. Diagram of the LU Processing computation module.

memory is required to enable computation on one set of blocks while simultaneously preloading the next set.

The key inputs to the compute engine are: the size of the matrix, N, the starting memory address of the matrix in external memory and a start signal. The output is written back to the external memory over the original input matrix, and a done signal is asserted. Our compute engine generator, however, is highly parameterized to enable the portability and scalability described in the introduction. These parameters will be discussed in Section 4. We now proceed to describe the computation and data marshalling functions in turn.

3.3. Computation

In this section, we will describe the hardware needed to implement the LU factorization computation. As described in Section 2.2, there are four different block operations that have to be performed. The LU Processing module contains the data path units that perform all four block operations. A diagram of the structure of the LU Processing module is shown in Figure 3. As described in the previous section, the computation requires three input blocks, labeled top block, left block, and current block in the figure. Recall that the engine must load two of the blocks for the subsequent computation as part of the double buffering, and so the left and current blocks have "0" and "1" versions in the figure. The top block is only updated while computing a block in a new column.

Most of the area of the LU processing module (and indeed the total design) is made up of the processing elements. The key engine parameter is **k**, the number of such processing elements. Each processing element contains a multiplication and a subtraction floating-point unit. The multiplication units use data from the left block and top block. The subtraction units use data from the current block and the outputs from the multiplication units. The output from the subtraction units are written to the current block and occasionally to the top block. The multiplexers (labeled mux) shown in Figure 3

are needed to route the data among these memory block units, and are controlled by the LU Controller. The LU Controller ensures all data dependences in the algorithm are respected.

While the computation largely consists of multiplications and subtractions, some blocks perform one division on all the elements while other blocks perform no division. Rather than creating many parallel dividers that are infrequently used, we compute the reciprocal of the divisor (with just one divider), and use the multiplier units to compute the division. Overall, only one division per matrix column is performed.

The floating point units are generated using Altera's MegaCore IP functions [Altera Corporation 2008]. The multiplication and subtraction units are fully pipelined and have a throughput of one result per cycle. Since division is performed infrequently, it is implemented as a multicycle computation to prevent it from becoming the critical path of the engine.

The on-chip memory blocks must supply enough data to keep the **k** processing elements busy. The left block and current block need to supply a matrix element to each multiplication and subtraction operator respectively. Therefore the data width of the left block and current block has to be 32 times the number of processing elements (**k**) to support 32-bit (single precision) arithmetic. This data width is doubled for double precision. Typically, **k** is on the order of 100, therefore these on-chip memories are very wide, on the order of 3200 data bits for single precision. This is only possible because of the high bandwidth of on-chip FPGA memories. For the top memory block, one matrix element is sent to all the multipliers or the one divider, and therefore the top block has a data width of 32 bits for single precision.

3.4. Data Marshalling

The data marshalling function (the transfer of blocks to and from external memory) is performed by the Memory Controller (MemC), Data Transfer Unit (DTU) and Marshalling Controller (MC) as illustrated in Figure 2. The coefficient matrix A is stored in column major format in the off-chip memory to match the data storage and flow on-chip. The blocks that need to be loaded on-chip are broken into contiguous sets of memory addresses, which the MC issues as load and store instructions to the DTU. Each instruction consists of the external memory address, the on-chip memory address, size of transfer, a load signal and a store signal.

The MemC is a DDR2 memory controller generated from the DDR2 SDRAM High Performance Memory Controller in Altera's MegaCore IP functions [Altera Corporation 2008]. This unit receives read or write commands up to the burst length of the DDR2 and converts it into appropriate DDR2 off-chip interface. The DTU takes an arbitrary size of memory transfer and breaks it up into suitably sized read or write commands to the MemC.

The Data Transfer Unit (DTU), shown in more detail in Figure 4, along with MemC are the key modules that enable the portability of our compute engine generator to different FPGA-based boards regardless of the type of off-chip memory. We have designed the DTU to allow the operating speed of the off-chip memory to be independent of the speed and bandwidth (number of bits of data width) of the on-chip memory. The speed and external bus width are two key parameters to the compute engine generator describing the nature of the off-chip memory. The decoupling is accomplished through the use of the FIFOs illustrated in Figure 4. The left hand side of the FIFOs operate at the external memory clock speed, while the right hand side operates at the compute engine's clock speed.

The FIFOs in the DTU are generated using Altera's multiclock FIFOs MegaCore IP functions [Altera Corporation 2008], which have parameterized input and output data width. One side of the FIFO has to match the data width of the MemC and external

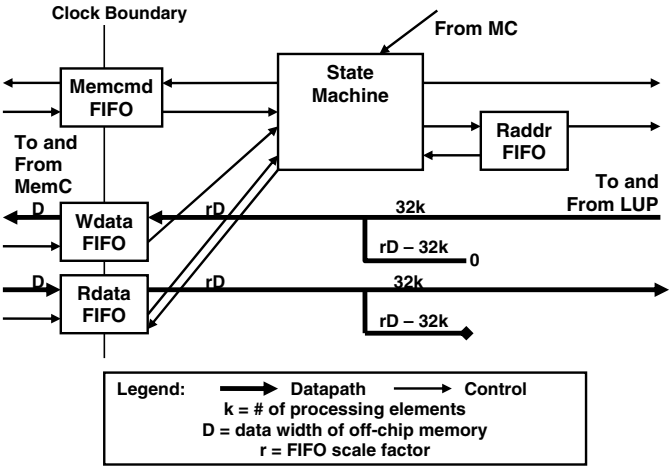

Fig. 4. Diagram of the Data Transfer Unit.

memory itself, D bits; while the other side of the FIFO has to match the data demand of the processing elements, which is $32\times$ **k** bits for single precision. The FIFO's input or output datawidth can be scaled by a factor, r, but r is limited to powers of 2 and thus, it is unlikely the two sides will match after scaling.

In the case that it matches, all data read from external memory can be written to on-chip memory and vice versa. However, when it does not match, we have to deal with the extra bits. One option is the extra bits contain useful data and we will add resource to use them in the next read or write. However, this solution requires shifting the next data to line up to the end of the extra bits, which is expensive to do on an FPGA. Instead the extra bits are ignored. Since the on-chip memory resource is more scarce/valuable than external memory, the external memory is padded with zeroes. We scale up the FIFO so that the data width coming from or going to the external memory is larger than on-chip memory.

Similarly, the size of the matrix will not always match the blocking size that is used in the engine. To simplify the data marshalling task, we pad the end of the column so that it is a multiple of the block size and each column starts some multiple of the block size from the previous column. These extra padded sections of columns are not loaded or stored. The cost of having internal padding is an increase in the total memory needed in the external memory to store the input matrix, which we assume is sufficient to store any input matrix. Currently, the user is required to prepare the input matrix by adding the necessary padding. A future work would be to implement a script to automatically pad the input matrix.

To illustrate the overhead of padding, we will compute the off-chip memory overhead for an example compute engine. In this example, the compute engine has 120 processing elements (k = 120) with a block size of 120 ($N_b = 120$) and an off-chip memory controller data width of 128 bits (D = 128). Thus, the processing elements need 3,840 bits and the closest match we can get with the FIFO ratio is 4,096 bits, with a scale ratio of 32 (r = 32). So instead of needing only 3,840 bits to store one packet of a transfer, 4,096 bits are required and so there is a 6.7% increase in off-chip memory storage from the on-chip and off-chip memory data width mismatch.

The memory overhead due to block padding is very small for large matrices. For example, consider a matrix of size N = 10,000, and a block size N_b of 120, resulting in the matrix being contained in an array of 84×84 blocks. The last block in each block

column contains 40 columns of valid data, but since we restrict all blocks to have the same number of columns, N_b, we will allocate 120 columns of memory storage for it. So effectively, instead of storing a matrix of size 10,000 $\times$ 10,000, we actually store a matrix of size 10,080 $\times$ 10,000. This results in a 0.8% increase in off-chip memory. Combining the overheads of memory transfer padding and block padding leads to a total off-chip memory overhead of 7.5% in this example, illustrating that the memory overhead of padding is quite managable.

3.5. Verification

To verify the functionality of the compute engine, we compared the result from a simulation of the synthesizable verilog to a software version of LU factorization (a basic naive implementation based on Algorithm 1, which does not have pivoting). Since there are many elements to compare, we used the error norm as a metric for comparison [Diersch 2008]. First we use two different optimization levels in the gcc compiler (level 2 and 3) to create two different programs from the same code that only differ in the order of operations. The idea is the error norm between the two versions of the software provides a control or standard against to compare the error norm between the FPGA and software implementations. Using randomly generated matrices of size 70 to 100 with values ranging from -1000 to 1000 as test cases, we computed the resultant LU factorized matrix for the two software versions and the FPGA. By comparing the difference of the factorized matrix using the Frobenius L2 integral Root Mean Square error norm, we found that the error norm between the two software version was similar to the error norm between the FPGA and software.

4. PORTABILITY AND SCALABILITY

Portability and scalability is accomplished by having the compute engine adapt to the available resources of the FPGA and the specific external memory system on an FPGA board. We define portability as having the ability to move to a new FPGA and external memory system with minimal human effort. We define scalability as having the ability to automatically take advantage of speed, capacity and memory bandwidth improvements in the new FPGA and memory system. We achieve portability and scalability by (1) defining parameters for the portions of an engine that should change as the FPGA or external memory technology changes, and (2) creating a software compute engine *generator* which can create an HDL design implementation that matches the desired parameters. We implemented our generator as software (written in the C language) that generates HDL code in Verilog based on the parameters. More detail on the generator can be found in Section 5.1. The compute engine consists of automatically created HDL code from the generator and cores generated from the Altera MegaCore IP functions.

The parameters used in the generator are divided into two categories. The first is the core parameters that must be supplied by the user, and Table I shows a subset of these core parameters. The first 6 parameters in the table control the amount of resources used on the FPGA, which ultimately dictates the achieved performance of the engine. The performace of the engine typically increases as more resources are used but there are some counterveiling effects that influence performance as well, which we will discuss in Section 6. By changing these parameters, the user can create the most suitable engine that matches the resources available on the FPGA. The last 4 parameters in the table modify the external memory interface, making the engine portable by allowing the use of different external memory systems. In addition to affecting how the vendor-supplied off-chip memory controller is implemented, they affect the FIFOs in the DTU to ensure that the correct commands are issued to the

Table I. Core Parameters Used to Generate LU Factorization

Name	Description
k	Number of processing elements
precision	Single or double precision
AdderLatency	The latency of adder unit
MultLatency	The latency of floating point multiplier unit
DivLatency	The latency of floating point divide unit
OnChipRamBlockSize	The size of the on chip memory blocks
DDRWidth	The datawidth of the DDR2 memory interface
DDRAddrWidth	The width of the DDR2 address line
DDRRowAddrWidth	The width of the DDR2 row address line
DDRBurstLen	The burst length of the DDR2 memory interface

Table II. Advanced Parameters Used in Our Generator

Name	Variable	Description
ExtraOnChipRamBlock-InputPortDelay	RI	Extra registers added to input port of on chip current and left blocks
ExtraOnChipRamBlock-OutputPortDelay	RO	Extra registers added to output port of on chip current and left blocks
ExtraOnChipTopBlock-InputPortDelay	TI	Extra registers added to input port of on chip top block
ExtraOnChipTopBlock-OutputPortDelay	TO	Extra registers added to output port of on chip top block

memory controller and data is transferred to the on-chip memory at the appropriate times.

The second category of parameters, which we refer to as *advanced* parameters, describe less visible internal design parameters of the engine. The user does not have to input these parameters since they do not affect functionally but rather influence performance; default values of these parameters, which are provided by the generator, can be used to obtain a functional engine. These parameters can be (optionally) employed by the user to exert more fine-grained control over the timing optimization of the engine. For better performance, the user can modify these parameters to suit their FPGA platform. Table II lists the most important advanced parameters that can influence the performance of the engine. These parameters add one or more registers in series to the inputs and outputs of on-chip memory blocks, which can increase maximum operating frequency at the cost of increased area consumption.

For compute engines with many processing elements, the floating point units will likely be placed across the entire FPGA and thus the on-chip memory blocks will have to connect to regions that span much of the FPGA. This leads to many of the critical paths in the compute engine occuring in routes to and from the input and output ports of the on-chip memory blocks. By adding one or more registers in series in these paths, and engaging appropriate synthesis techniques, these long distance routes can be pipelined effectively and the design speed improves. Simply adding one set of registers may not sufficiently reduce delay since many paths which fan out across the chip will have to connect to these registers. Consequently it is important to duplicate these registers so that they can be spread out across the FPGA and pipeline the routing delay to the logic they feed. We rely on the physical synthesis process in the Altera CAD tools [Altera 2008] to automatically duplicate these registers, to determine which paths should connect to which duplicate register, and to choose where to place each duplicate. It is important to note that without this synthesis method, the maximum operating frequency degrades significantly [Zhang et al. 2008].

There is a trade-off here, though; too many such registers use excessive resources on the FPGA, and may cause congestion and actually decrease maximum operating

Table III. Clock Frequency for Compute Engine (57 Double-Precision PEs) with Different Advanced Parameters on Stratix III 3SL340 FPGA

RI	RO	TI	TO	Clock Frequency
2	1	4	2	125 MHz
2	1	6	4	155 MHz
3	1	6	4	170 MHz
3	2	7	2	115 MHz

frequency. Table III shows some experimental data that illustrates this with a design consisting of 57 double precision processing elements (PEs) synthesized, placed and routed on a Stratix III 3SL340F1760C3 FPGA. Each row of the table gives the frequency achieved with a different setting of these advanced parameters; as more pipeline registers are added to the engine, the clock frequency improves as critical path delays are reduced. However, the clock frequency decreases for the last set of parameters in the table because of congestion on the FPGA. This kind of experimentation is needed to determine the best number of registers to use, although we set default values based on our own experiments. This points out that, as is common on all platforms, to obtain the very best performance requires more in-depth design effort.

5. CHALLEGES TO ACHIEVING PORTABILITY AND SCALABILITY

In this section, we discuss the challenges that were faced in creating a portable and scalable design and the solutions we used. We will also quantify the cost (in effort) and benefits of implementing a portable and scalable design as compared to a normal hardware design.

5.1. Compute Engine Generator

One of the key goals of this work was to understand the effort to create a custom computing system that will easily and quickly customize the design to the target hardware, thereby creating portability and scalability. We found that Hardware Description Languages (HDL), such as Verilog, were not sufficiently powerful programming languages to fully adapt the compute engine in all the ways needed as described in Section 4. For example, Verilog cannot easily create some variable sized structures, especially when the size of one structure depends in a complex way on an input parameter or a set of parameters. Consequently, we implemented a generator in software (written in the C language) which generates HDL code in Verilog, based on the user input parameter values. The C code mainly consist of fprintf statements that output HDL statements into a file. The code employs if-else and for-loop statements to alter specific HDL statements based on the parameters. This higher functionality in C allows us to easily create variable sized structures based on the specified parameter values. Verilog does have the ability to use parameter variables to represent constant values, which were used as much as possible to minimize changes in generated HDL. This feature also improves the readability of the HDL code as it highlights the parameters used in specific modules and shows how they affect the HDL code.

The following example portions of the full design illustrate how the generator produces customized engines. Consider the 2**k**-to-1 multiplexer used in the LU Processing module to store values into the top block shown in Figure 3. The size of this mux changes as a function of the **k** parameter, with the number of inputs to the mux based on the following equation: $2k \times \ precision$ (where precision is either 32 bits for single or 64 bits for double). The generator C code, shown in Figure 5(a), automatically creates the desired mux size based on the **k** and precision parameters. Two for-loops are used to create the 2**k**-to-1 mux. Each for-loop generates a k-to-1 mux and combined together, they form a 2**k**-to-1 mux. The generated HDL code for $\mathbf{k} = 4$ and precision $= 32$ (single precision) is shown in Figure 5(b).

```
(a)
fprintf(fp, " case (topWriteSel)\n");
for (i = 0; i < k; i++) {
      lowerIdx = (k-i-1)*precision;
      upperIdx = (k-i)*precision-1;
      fprintf(fp, " %i:\n", i);
      fprintf(fp, " topWriteDataLU = ramReadDataLU[%i:%i];\n",
            upperIdx, lowerIdx);
}
fprintf(fp, " default:\n");
fprintf(fp, " topWriteDataLU = ramReadDataLU[PRECISION-1:0];\n");
fprintf(fp, " endcase\n");
fprintf(fp, " else\n");
fprintf(fp, " case (topWriteSel)\n");
for (i = 0; i < k; i++) {
      fprintf(fp, " %i:\n", i);
      fprintf(fp, " topWriteDataLU = addResult[%i];\n", k-i-1);
}
fprintf(fp, " default:\n");
fprintf(fp, " topWriteDataLU = addResult[0];\n");
fprintf(fp, " endcase\n");

(b)
if (topSourceSel == 0)
      case (topWriteSel)
      0:    topWriteDataLU = ramReadDataLU[127:96];
      1:    topWriteDataLU = ramReadDataLU[95:64];
      2:    topWriteDataLU = ramReadDataLU[63:32];
      3:    topWriteDataLU = ramReadDataLU[31:0];
      default:   topWriteDataLU = ramReadDataLU[PRECISION-1:0];
      endcase
else
      case (topWriteSel)
      0:    topWriteDataLU = addResult[3];
      1:    topWriteDataLU = addResult[2];
      2:    topWriteDataLU = addResult[1];
      3:    topWriteDataLU = addResult[0];
      default:   topWriteDataLU = addResult[0];
      endcase
```

Fig. 5. (a) shows C code from Generator; (b) shows automatically created Verilog code for single-precision.

Another example of customization in the engine is the shift registers used to delay control signals in the LU Controller module. The control signals used to store the output of the floating point units are passed through variable length shift registers to create the necessary delay to match the latency of the floating point units, which are determined by the AdderLatency, MultLatency, and DivLatency parameters. Typically, one for-loop is used in the generator C code to create the desired size of the shift registers for one control signal. Each iteration of the for-loop increases the size of the shift registers by one and the number of iterations is a function of the latency parameters. In cases that the delay could be zero, an if-else statement is use to handle this special case. It is possible to do this in Verilog using a for loop, but given that computing the required (or in some cases, most optimized) control signal latency often required complex expressions, it was easier to do in C.

There have been other studies that use generators to create specific purposed FPGA designs. Mencer et al. [1998] created a tool that used C++ along with specific computational operators (adders, multipers and special artihmetic units for encryption) to create custom FPGA designs. In Moore et al. [2007], standard signal processing functions in C++ were used to generate specific FPGA designs. Liang and Jean [2003] implemented a generator that will create generatized template matching operations typically used for image processing.

5.2. IP Cores

Besides custom generated HDL, our design also employs IP cores provided by the tool vendor, Altera. By using these cores, it saves us development time as we do not need

to create our own cores. However, it also restricts overall portability because these cores are limited to FPGAs from that specific vendor. In order to use an FPGA from another FPGA vendor the design would have to change to use that vendor's IP cores. For example, Xilinx does provide all the cores that are required for the LU Engine; however, there are differences between the interfaces used by Altera and Xilinx. For the floating point operation cores, the only difference between the two vendors is the naming conventions for the input and output ports. A simple wrapper can be used to provide an unified interface to the design.

The more complex off-chip memory controller core has more differences between the two vendors, and thus requires more work to adapt. Both vendors' controllers have similar interface ports. They all need a port for the address, write data, read data, and control signals to indicate a read or write command or when read data is valid, etc. However, there can be slight differences in the control ports. For example, Altera has a separate port for read requests, write request and a port to change the number of bytes that is written or read per command, while Xilinx groups the read and write request into one command port. There is no additional port to indicate number of bytes per command because this is a fixed number. The main difference between the memory controller cores comes from how the write command is performed. For a write command, Xilinx requires the user to preload the write data into a FIFO in the controller before the write command is issued. For Altera's controller in native mode, the write command is issued first and the controller will send a write data request signal when the next chunk of write data is needed; the user then has to send the data to write port by the next cycle. This difference will require a bridging block to be implemented in addition to a wrapper to handle naming differences.

In the future, it would be useful if there was a standard convention for these IP cores. Ideally, a commonly established module name and interface (with a fixed protocol) can be used in the HDL to instantiate these cores and the vendor can internally implement them in their own way. This approach would faciliate creating portable and scalable design as it shifts the task of writing these IP core wrappers to the vendors, who write the IP cores already.

5.3. Separating External Memory Interface

External high-speed memory is essential in many applications, and so supporting portability across different external memory configurations is an important requirement in meeting our design goals. We want the design to be able to use different external memory to ensure portability while avoiding any detrimental effects on performance. We achieve this by (1) using variable sized FIFOs to handle different data width requirements between external memory and internal memory and (2) running the external memory controller in an independent clock domain. Thus, there is a clean divide between the computation and the external memory. With this interface, the user can use off-chip SDRAM of various data widths and speeds and have very little impact on the computational section of the design, which is dependent on the resources available on the FPGA.

By having different clock domains for the external memory interface and the main computation units, each part of our design can run at its own maximum frequency. One does not have to slow the computation to match the clock speed of the external memory or vice versa, and since memory interface and on-FPGA clock speeds will probably increase at different rates, this flexibility is very important to the scalability of our engine. This flexibility allows the user to change either the FPGA or the external memory independently of each other. It comes at the cost of the extra complexity of a multi-clock design, with some clocks that are asynchronous to each other.

5.4. Costs and Benefits of Portability and Scalability

By creating a portable and scalable design, one trades a reduction in future development costs (when the design is reused) for upfront development costs. While it will take longer to create a portable and scalable design than a design that is specific to one board and one chip, the benefit is that the former can be adapted to a different device and board architecture with significantly less effort than reimplementing anew.

We estimate that for our LU-factorization engine writing a portable and scalable design required roughly 1.75x the design time that would have been needed for a board-and-chip specific design due to the parameterizing the design and the increased complexity of the FSM. We believe that if we increase the portability of the design so that it could target the FPGA families of two vendors (rather than the current one), the design time would grow to 2.5x that of a board-and-chip specific design, due to the extra effort to create wrappers to make some vendor-specific IP cores compatible. The floating point units are very straightforward to make multivendor, while the memory controller will require more work. In addition to the development time increase, it took roughly 2.5x the time to verify the portable and scalable design vs. the verification time for a board-and-chip specific design because of the many different features in the design.

In terms of optimization, the portable and scalable design took less time, we estimate about 0.95x of the specific design, because one can use the generator to create design alternatives automatically using new parameter values. This makes it easier and quicker to try various optimizations.

The major benefit of having portable and scalable design comes from reduced development cost when implementing the same design on other hardware. To scale up the design and move to a bigger FPGA in the same family, all that is required is to change a few parameters (those that are related to the computation blocks) and generate the new design, which will take just seconds. To optimize the design to maximize performance will take longer but based on our experience we estimate it will still be roughly 50 times faster than manually re-coding and re-optimizing a design. Re-targeting to a different FPGA architecture which still supports the same IP cores is also simple: one simply changes the necessary parameters and generates the design. Depending on the level of architectural change, some minor re-optimization may be required, but our experience in migrating across Stratix series FPGAs suggests the design time will still be approximately 40x less than a re-write. If the same IP cores cannot be used, it is still fairly easy so long as the appropriate IP cores with “wrappers” that expose a vendor-independent interface have been created during the generator design. In this case we expect the design time to still be reduced by approximately 40x vs. a rewrite, but more up-front effort in creating this level of generality in the engine is required, as described earlier.

6. OPTIMIZATION OF ENGINE DESIGN

The generator can create many engines for a specific FPGA platform by changing the parameters described in Section 4. Some of the parameters are determined based on the target FPGA platform to allow portability such as the parameters involved with the off-chip memory. The remaining parameters scale the performance achieved and the amount of resources used on the FPGA. The number of processing elements ($\mathbf{k}$) and latency of floating point units are key core parameters that affect the performance of the computation engine. The advanced parameters can be used to reduce the critical path delay and improve clock frequency. Using these parameters, the user can increase the resource usage on the FPGA to obtain better performance. This succeeds until the FPGA becomes too congested and performance begins to decrease.

In this section, we will explore how these parameters impact performance. For this exploration, we targeted the largest FPGA available at the time of this work, the Stratix III 3SL340F1760C3 FPGA. The specific design employs 256MB of DDR2 SDRAM with a 64-bit wide data path as the off-chip memory. The optimization discussed below is for a single precision compute engine.

6.1. Methodology

We measure the performance of the compute engine by the number of billions of floating point operations it can execute per second (GFLOPS). The maximum GFLOPS achievable for our compute engine can be calculated by multiplying the maximum number of floating point operations performed per cycle with the number of cycles per second. However, there is overhead required to set up the computation and data dependencies in the LU factorization algorithm that prevent the engine from achieving this maximum performance. Thus we measure the performance in useful GFLOPS, which counts only operations that are used to solve the linear system.

The useful GFLOPS can be calculated by dividing the total number of floating point operations needed to solve the LU factorization by the total runtime. The runtime is determined as the product of the total number of cycles required to execute the computation and multiplying it by the cycle time of the compute engine. The cycle count is determined through simulation of the circuit with the Modelsim simulator. The cycle time is determined by timing analysis after synthesis, placement and routing by Altera's Quartus II CAD tool, version 7.2.

6.2. Number of Processing Elements

The key parameter that scales the performance of the engine (and increases resource usage) is the number of processing elements, $\mathbf{k}$. There are competing trends that impact performance as the number of processing elements is increased. As $\mathbf{k}$ increases, more computation can be performed per cycle and so the number of cycles required to compute decreases. However, as an FPGA fills up with more processors, a number of effects conspire to reduce the maximum clock frequency as discussed below.

To investigate the impact of parameter $\mathbf{k}$ on performance, we synthesized and simulated several engines with the same core parameters while varying $\mathbf{k}$. We further optimized each engine by adjusting the advanced pipelining parameters described in Section 4, which allows the physical synthesis tool to pipeline routes to the far-flung memories, to achieve the best performance. Note that because of the highly pipelined nature of the design, increasing the latency in this way does not significantly impact performance. As $\mathbf{k}$ increases more of the FPGA is occupied so there is less space available to build the registers needed for this route pipelining.

Table IV shows the performance in GFLOPS of several compute engines generated in this manner. The maximum number of single precision processing elements that can fit on the Stratix III 3SL340 FPGA is 144, limited by the number of hard 36x36 multipliers on the chip. The second column of Table IV provides the maximum post-placement and routing clock frequency achieved and the third column gives the achieved useful GFLOPS for large matrices. The fourth column shows the performance relative to the compute engine with 30 processing elements ($\mathbf{k} = 30$).

Figure 6 is a plot of the second and third column of Table IV versus the first column. It illustrates the effect of varying the number of processing elements on clock frequency and performance, and shows that performance increases with the number of processing elements up to approximately 128 PEs. Up to this point, the decrease in number of cycles for the computation (due to more processing elements) outweighs the slight decrease in clock frequency, resulting in improved performance. Past this point, the frequency decrease outweighs the cycle count improvements.

Table IV. Performance of Several Computing Engines on Stratix III 3SL340 FPGA

Number of Processing Elements	Maximum Clock Frequency	GFLOPS	Performance Ratio vs 30 PEs
30	240 MHz	14	1.00
60	220 MHz	25	1.8
90	215 MHz	38	2.7
120	185 MHz	43	3.1
128	180 MHz	45	3.2
136	100 MHz	26	1.9
144	95 MHz	26	1.9

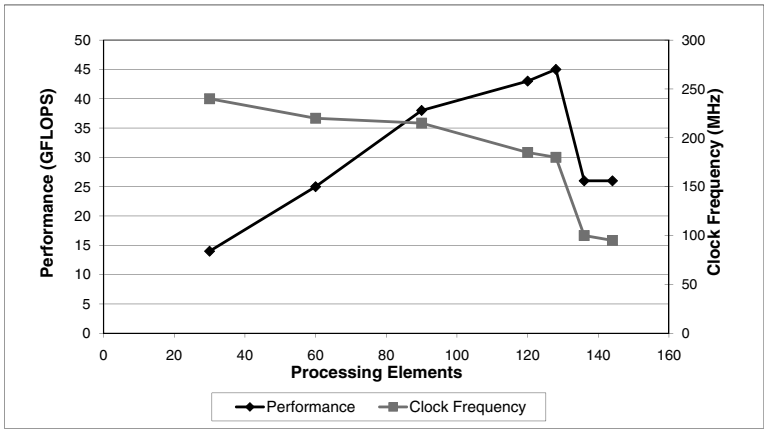

Fig. 6. The effect of varying the number of processing elements on performance and clock frequency on Stratix III 3SL340.

The decrease in clock frequency arises for a number of reasons: first, it becomes harder for the placement and routing tools to optimize the critical and near-critical paths when the chip is fully occupied, as there are fewer choices. Second, the routing paths in the engine simply become longer as more of the chip is used; both the control signals and data in on-chip memory have to be sent to all PEs. Thirdly, as **k** increases, the size of the 2**k**-to-1 multiplexer in the LU Processing module shown in Figure 3 becomes larger. A larger multiplexer requires more levels of logic to implement and thus the paths through this multiplexer are longer. These longer paths often become the critical paths in the engine. Finally, as the chip becomes more occupied, there is less ability to include more piplelining registers to ameliorate the delays of long routes. We have found that the maximum performance is achieved when no more than 90% of the FPGA is utilized.

6.3. Floating Point Unit Latency

The latency of the floating point units (FPUs) also affects the performance of the engine, and interacts with the pipelining of the interconnect delays described previously.

In the case that the device is not fully occupied (less than the 90% threshold described here) the best choice is to select a floating point unit that has the highest operating frequency and is therefore the most pipelined, and has the highest latency. Here, the FPGA resources can also be allocated to the appopriate amount of pipelining for the interconnect.

However, once the chip begins to become more fully occupied, it may be better to carefully trade off the pipelining of the FPU with the pipelining of the interconnect, as they both compete for scarce registers.

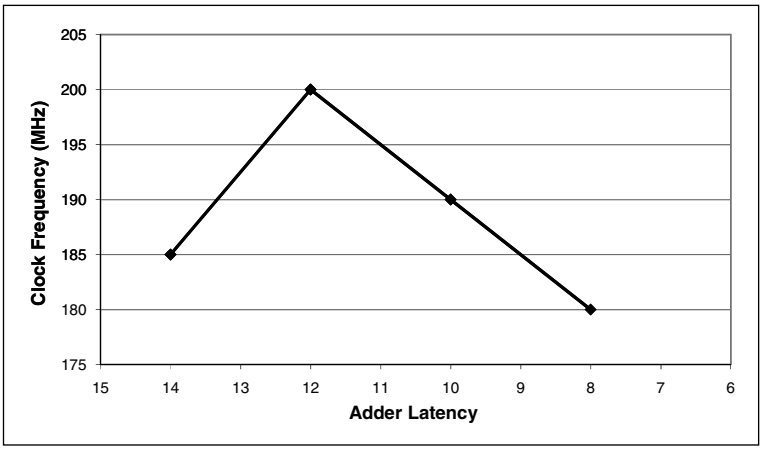

Fig. 7. Clock Frequency for Compute Engines with Different Adder Latencies on Stratix III 3SL340.

The floating point units have two key parts: a multiplier and an adder. The floating point multiplier unit in the Altera Megacore IP functions provide only 2 useful latency value choices, and thus does not play a significant role in this trade-off. However, the Altera floating point adder units have a large range of latency/pipelining values and so play a major role in this trade-off. Figure 7 illustrates the maximum clock frequencies achieved for a set of engines with 120 PEs and various adder latencies. For each engine, the routing interconnect pipelining was optimized (using our "advanced" parameters) as described in Section 6.2, to obtain the highest clock frequency. Given the same number of processing elements, the engine with the highest clock frequency has the best performance. As shown in the figure, the best performing engine has adder latency of 12. Comparing this engine to one with the maximum adder latency of 14, this engine used less area for the floating point units which was instead used for pipeline registers to improve other critical path delays. For engines with latencies less than 12, the adders in those engines become the critical paths, thus decreasing the overall clock frequency.

7. EXPERIMENTAL RESULTS

In this section, we present measurements of the overall performance of the engine on a modern FPGA, and compare them to a software version running on a processor fabricated in the same (65 nm) process technology. We targeted our compute engine to the largest FPGA available at the time, which was the Stratix III 3SL340F1760C3 FPGA. We assume that the FPGA is attached to off-chip DDR2 SDRAM of depth 256MB and data width of 64 bits. This is compared to two software versions: one from the Intel MKL library [Intel 2008] and a more basic code written by the author. The processor used is an Intel Xeon 5160 dual core 3.0GHz processor with 4MB of L2 cache and 8GB of RAM. The Intel MKL library is highly optimized multithreaded code specifically created for the Intel processor, which makes use of the vectorlike SSE2 and SSE3 instruction set. The more basic code is single-threaded and it is modeled after the pseudocode in Section 2.1, but includes blocking. We include a comparison to this more basic code to illustrate the difference between naive software and highly vendor-optimized software. It is important to use the latest software algorithms for a fair comparison of one's results to the state of the art.

Table V. Single Precision Performance on 65nm FPGA and Processor(s)

Platform	Clock Frequency	GFLOPS	Performance Ratio
FPGA: Stratix III 3SL340F1760C3	200 MHz	47	2.2
CPU: MKL on Xeon 5160 single core	3 GHz	21	1
CPU: MKL Xeon 5160 dual core	3 GHz	42	2
CPU: basic code on Xeon 5160 single core	3 GHz	1.1	0.052

Table VI. Double Precision Performance on 65nm FPGA and Processor(s)

Platform	Clock Frequency	GFLOPS	Performance Ratio
FPGA: Stratix III 3SL340F1760C3	170 MHz	19	1.7
CPU: MKL on Xeon 5160 single core	3 GHz	11	1
CPU: MKL on Xeon 5160 dual core	3 GHz	21	1.9
CPU: basic code on Xeon 5160 single core	3 GHz	0.55	0.05

7.1. Performance

The top-performing single precision compute engine employs 120 processing elements on the above FPGA, using adder units with latency of 12. It achieves a maximum operating frequency of 200MHz, after selecting the best combination of pipelining in the floating point unit and interconnect, as discussed in Section 6.3.

Table V gives the performance of several devices, each fabricated in the same 65nm process. The first column lists the platform (FPGA, single processor, or dual processor) and in the case of the processor, which version of the software code was used, MKL or basic. The table also gives the operating frequency of the hardware (either of our design or the processor clock speed), the performance in GFLOPS, and the performance normalized to that of the single core optimized MKL code.

Our FPGA implementation achieves a performance of 47 GFLOPS, which is 2.2 times greater than the single core Xeon running MKL. The FPGA is essentially tied with the dual core processor. This is a surprising result, as we expected to achieve far more significant speed gains. The Intel optimized code for the Xeon processor makes use of the SSE2 instruction set, which employs 4-way data parallelism on 32 bit single-precision quantities for multiplication and addition. We believe also that the optimized implementation uses a careful blocking scheme (similar to the one described in Section 2.2) to make the best use of the caches on the processor.

The quality of this optimization is illustrated by the performance of the basic software as shown in Table V. The basic software is a factor of 19 times slower than the optimized MKL code running on the Xeon single core processor. We believe a key lesson of this research is that for FPGA-based compute acceleration it is crucial to compare to the best performing software on large-scale problem instances, as we have done here. We understand that significant effort is expended by Intel to produce this optimized library, perhaps not unlike our effort to create the FPGA-based design.

For the Stratix III 3SL340F1760C3 FPGA, we also determined the highest performance double-precision compute engine, which has 57 double-precision processing elements. In this case, this engine has the maximum number of double-precision PEs possible and is limited by the number of hard multipliers on the FPGA. The double precision multiplier uses 2.5 times more hard multipliers than the single precision multiplier, and twice as many registers. This engine achieves a maximum operating frequency of 170MHz. Table VI gives the performance in GFLOPS of several platforms in double precision, similar to Table V.

The double precision FPGA implementation achieves a performance of 19 GFLOPS, which is 1.7 times greater than the single core Xeon running the MKL code. This relative performance improvement is less than that for single precision, because of

Table VII. Single Precision Power Consumption and Energy Efficiency Comparison

Platform	Power (W)	Power Ratio	GFLOPS per Watt	Efficiency Ratio
FPGA: Stratix III 3SL340F1760C3	18	0.45	2.61	5
CPU: MKL on Xeon 5160 single core	40	1	0.525	1
CPU: MKL on Xeon 5160 dual core	80	2	0.525	1
CPU: basic code on Xeon 5160 single core	40	1	0.0275	0.052

the higher cost for the FPGA to perform double precision floating point operations. The double precision multiplication floating point unit uses 2.5 times the number of hard multipliers than the single precision unit, so less than half the number of processor elements can fit on the FPGA. In contrast, the SSE2 instruction set in the processor enables 2-way data parallelism on 64-bit double precision quantities for multiplication and addition, so it can perform half the number of operations per cycle in double precision than single precision. Hence the GFLOPS for double precision is approximately half that of the single precision. In addition, while the operating frequency remains the same for the processor, the operating frequency is lower for double precision on the FPGA.

7.2. Power Consumption

While it is true that, in supercomputing, performance is the key metric, in recent years the power consumed for computation has become a significant issue, not only in the portable world, but in the cost of electricity required to support super computers. Table VII shows the power consumption of single precision compute engines on each of the platforms listed in Table V. The second column lists the power consumption of each platform and the third column specifies the power ratio, which is the power of the platform divided by the power of the single core processor. The fourth column contains the energy efficiency in GFLOPS per Watt and the fifth column states the energy efficiency ratio normalized to the single core processor running MKL.

The power consumption of the 120 processing element FPGA design was measured using vectorless estimation in Altera's PowerPlay Power Analyzer. Although vectorless estimation is less accurate then using simulation to obtain signal toggle rates, it is generally within 30% of the actual power consumption. The power consumption of the Xeon dual core processor was determined from the specification on the Intel Web site [Intel Corporation 2008], which should be close to the actual power since the MKL code keeps the processor busy. The Xeon dual core processor requires 80W of power and we assume that a single core requires half the power.

As shown in Table VII, the FPGA implementation requires 2.2 times less power than the single-core Xeon processor. Furthermore, the performance in GFLOPS per watt, which is essentially the amount of energy used per computation, is 5 times better for the FPGA implementation than the processor. As the performance of the dual core is twice as fast as the single core but uses twice the power, the energy efficiency of the Xeon single and dual core processor is the same.

Table VIII shows the power consumption for double precision, similar to Table VII. The double precision FPGA implementation uses 2 times less power than the single core Xeon processor. In terms of GFLOPS per watt, it is 3.5 times better than the single core processor. Similar to the performance ratio, the energy efficiency ratio for double precision FPGA implementation is lower than the energy efficiency ratio for single precision FPGA implementation. The energy efficiency of the single and dual core processor is about the same for both cases.

Table VIII. Double Precision Power Consumption and Energy Efficiency Comparison

Platform	Power (W)	Power Ratio	GFLOPS per Watt	Efficiency Ratio
FPGA: Stratix III 3SL340F1760C3	20	0.5	0.95	3.5
CPU: MKL on Xeon 5160 single core	40	1	0.275	1
CPU: MKL on Xeon 5160 dual core	80	2	0.263	0.96
CPU: basic code on Xeon 5160 single core	40	1	0.0138	0.052

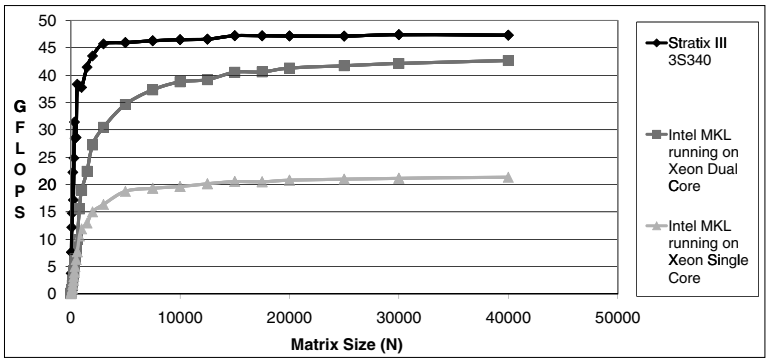

Fig. 8. Performance as a Function of Matrix Dimension.

Table IX. Performance on FPGA and Processor Platforms for Solving a 600x600 Single Precision Matrix

Platform	GFLOPS	Performance Ratio	GFLOPS per Watt	Efficiency Ratio
FPGA: Stratix III 3SL340F1760C3	38	4.9	2.1	11
CPU: MKL on Xeon 5160 single core	7.7	1	0.19	1
CPU: MKL on Xeon 5160 dual core	10	1.3	0.125	0.65

7.3. Matrix Size

We observed that many previous works that implemented linear equation solvers on FPGAs typically only use on-chip FPGA memory to store the matrix, which severely limits the size of the problems addressed, and therefore the overall applicability. Our implementation employs off-chip large-scale memory and therefore is much more widely applicable. Figure 8 illustrates the performance (in GFLOPS) for the various platforms as a function of the NxN matrix dimension, N. Here you can see that the performance for all platforms eventually levels out and reaches a maximum as matrix size increases. The performance comparison we used was for the larger matrix size that achieves these leveled off performance values. When comparing for small matrices, the performance of the software and FPGA implementation have not reached their maximum performance. As shown in the figure, the FPGA implementation ramps up faster and is able to reach its maximum achievable performance at smaller matrix sizes than software. Thus, there is a larger speedup for FPGA over software when solving small matrices.

We will illustrate this larger speed up by comparing the performance for solving a smaller matrix size. Since a Stratix III FPGA can store a maximum single precision matrix of 721×721 in on-chip memory, we will compare the performance of an FPGA and a processor when solving a 600x600 single precision matrix. Table IX shows the performance and energy efficiency in single precision for each platform listed in Table V. The second and fourth columns show the performance in GFLOPS and energy efficiency in GFLOPS per Watt. The third and fifth columns show the performance and energy efficiency ratio normalized to the single core processor. Since the FPGA can reach its

Table X. Performance on Stratix II 2S180 and Stratix III 3SL340

Platform	Precision	PEs	Clock Frequency	GFLOPS	Performance Ratio
Stratix II 2S180F1508C3	Single	64	170 MHz	21	1
Stratix III 3SL340F1760C3	Single	120	200 MHz	47	2.2
Stratix II 2S180F1508C3	Double	29	140 MHz	7.4	1
Stratix III 3SL340F1760C3	Double	57	170 MHz	19	2.6

maximum performance faster than software, the FPGA has a higher performance and efficiency ratio than it does for larger matrices. The FPGA achieves a performance of 38 GFLOPS while the software running on single core and dual core processor has GFLOPS of 7.7 and 10 respectively. Comparing the FPGA to the single core, the FPGA has 5 times higher performance and is 11 times more energy efficient. The performance of the software on the dual core processor increases more slowly than on a single core processor and therefore, the dual core processor actually has lower energy efficiency than the single core processor. The FPGA has roughly 4 times higher performance and is 17 times more energy efficient than the dual core.

Although there are some real-time applications that require accelerated solving of small matrices, an accelerator for small matrices is not very important for most applications since the overall computation time is already small. We are simply pointing out that not using leveled-off performance values is a common pitfall of FPGA acceleration research.

7.4. Comparing Different FPGAs

To demonstrate a limited form of portability and scalability of our generator and engine, we targeted the generator at the previous-generation Altera Stratix II 2S180F1508C3 FPGA but assumed it is attached to the same off-chip DDR2 SDRAM of depth 256MB and datapath width of 64 bits. We determined the top-performing single and double precision engine in a similar fashion as described in Section 7. The best single precision compute engine we achieved has 64 processing elements with adder latency of 12 and operates at 170MHz. For double precision, our best performing engine has 29 processing elements with adder latency of 12 and operates at 140MHz.

Table X compares the performance of the Stratix II 2S180 FPGA engine to the performance of the Stratix III 3SL340 engine as shown in Table V and VI for both levels of precision. The first and second columns show the target platform and precision respectively. The third and fourth columns list the number of processing elements and the clock frequency in the top-performing engine respectively. The fifth column shows the performance in GFLOPS and the sixth column shows the performance ratio normalized to the Stratix II 2S180 FPGA of the same precision. For single precision, the Stratix III 3SL340 FPGA has a 2.2 times performance improvement over Stratix II 2S180, and a 2.6 times performance improvement for double precision. Thus, the performance for both versions scales up by about the same amount. From a Stratix II FPGA to a Stratix III FPGA, the clock frequency increased by 18% for single precision and increased by 21% for double precision. The majority of the performance improvement comes from increasing the number of processing elements, which increased by 87.5% for single precision and 93% for double precision. This scalability feature in our generator is key in achieving significant performance improvement as one moves from one generation FPGA to the next.

7.5. Comparison to GPU

Volkov and Demmel [2008] accelerated the LU factorization algorithm using a CPU and GPU system. They employed a 2.67GHz Core2 Duo E6700 combined with an Nvidia GeForce 8800GTX and they were able to achieve a performance of 179 GFLOPS for

single precision. For the same CPU combined with a GeForce GTX280, they were able to achieve a performance of 309 GFLOPS using single precision data. Comparing with our results, the GTX280 GPU result is 6.6x faster than our FPGA implementation; however if you take into account power consumption, the GTX280 has a maximum power consumption of 236W [NVIDIA Corporation 2011] which is a power efficiency of 1.31 GFLOPS per Watt. Compared to the FPGA implementation described here, FPGA achieve roughly 2 time better energy efficiency than the GPU.

8. CAVEATS AND FUTURE WORK

8.1. Multi-FPGA Design

Our design uses a blocking approach, where blocks are read from external memory into the FPGA to be computed and then written back. This approach can easily be modified to use multiple FPGAs to parallelize the computation. If all the FPGAs have access to the same external memory, then a scheme can be determined so that each FPGA will have its own set of blocks to compute and greatly increase performance. For example, in a system of two FPGAs, the first FPGA can compute the blocks in the odd columns of the matrix and the second FPGA will compute the blocks in the even columns. The second FPGA cannot start computation until the first FPGA has finished computing the blocks in the first column. The first FPGA can signal directly to the second FPGA to start computation or write a certain value in a fixed location in external memory that the second FPGA will periodically check. After all the blocks have been computed, the two FPGAs can start on the next iteration of the computation.

Because parallel computations cannot begin until the first FPGA has finished processing the first column of blocks, there is a small overhead involved in the multi-FPGA scheme. However, since this overhead is small, our design will still scale well when using multiple FPGAs.

8.2. Limitations and Improvements

While one of the key goals was to provide an engine that is portable, there are a few restrictions that limit its portability. To reduce development time, we used predesigned cores from Altera (that are vendor-specific and therefore not portable to other vendors' devices, but are portable within this vendor's families) to generate part of our compute engine: specifically the floating-point operators and DDR2 memory controller were Altera-specific. In most cases, a simple wrapper could be created to allow vendor independent portability.

One final limitation involves setting up and initiating the engine. In our current design, we require a host processor to be able to fill the external memory with the input matrix data and it must also initiate the computation on the FPGA. In some FPGA computation systems, the external memory for the FPGA has a dedicated connection to the FPGA and the host processor has no access to it. In such a case the host would have to use the FPGA itself to fill the data in external memory. In general, an additional "setup" hardware module is needed to handle all possible board configurations for complete portability. Future work should remove these restrictions to achieve complete portability. Even with these restrictions, the generator provides more portability and scalability than most designs to date.

There are some improvements that can be made to the design and generator that can improve usability. One key improvement to the linear equation solver algorithm would be to implement pivoting, which would greatly improve the numerical stability of the algorithm and allow more diverse range of linear systems to be solvable using the design. Other improvements involve automating some of the things that users have to do in order to use or maximize the performance of the engine. They involve developing

a script to automatically pad the input matrix with zeroes as mentioned in Section 3.4 and automating the process to find the best performing design as described in Section 6.

While our current design solves only systems of linear equations, the framework of the design can be used to create portable and scalable designs for other scientific algorithms. The data marshalling design can be reused for any algorithm that requires block-based matrix access. The computation blocks will have to be modified but similar parameters can be used to maintain portability and scalability. Future work should involve implementing other common scientific algorithms and expanding the hardware library.

The floating point units use more than half the logic (both LUTs and registers) in the LU engine. An FPGA that included floating point capability within an enhanced DSP block, as suggested in Beauchamp et al. [2006], should be able to fit more than double the number of PEs per device, and hence increase the FPGA performance on this application by over 2X. This may be a fruitful direction for FPGA architects to explore to increase the attractiveness of FPGAs in the scientific computing market.

9. RELEASE

We have released a version of the generator that works online (the user specifies the parameters into a Web page) and delivers compute engines as well as a downloadable version of the generator itself at `http://www.eecg.toronto.edu/~jayar/software/LUgen/`.

10. CONCLUSION

In this work, we have created a portable and scalable computational engine generator for the LU factorization method of solving systems of linear equations in either single or double precision that should make it easier to employ FPGAs in super computing applications. Using the generator, we obtained a performance of 47 GFLOPS on a Stratix III 3S340 FPGA and a performance of 21GFLOPS on a Stratix II 2S180 FPGA in single precision. In double precision, we reached performances of 19 and 7.4 GFLOPS for a Stratix III 3S340 FPGA and Stratix II 2S180 FPGA respectively. For both precisions, the performance scaled by more than twice when moving from Stratix II to III, largely due to the near doubling of functional units in the engine. We have also shown that this engine has significant performance and performance per watt advantages over a single core processor, and that the choice of the software package used for comparison to a processor is crucial. Using a home-grown software benchmark for our comparisons to processors would have increased our speed-up numbers by a factor of 20x vs. those we obtained with a very highly optimized library created by Intel. Compared to this highly optimized software version, our scalable single precision FPGA engine is 2.2 times faster than a single core processor (built in the same IC fabrication process) and 5 times more energy efficient. For double precision, the FPGA engine is 1.7 times faster than a single core processor and 3.5 times more energy efficient.

REFERENCES

AGILITY DESIGN SOLUTIONS, INC. 2008. Handel-c. `http://www.agilityds.com/products/c_based_products/dk_design_suite/handel-c.aspx`.

ALTERA. 2008. Netlist optimizations and physical synthesis. Tech. rep., Altera Corporation. `http://www.altera.com/literature/hb/qts/qts_qii52007.pdf`.

ALTERA CORPORATION. 2008. Intellectual property solutions. `http://www.altera.com/products/ip/ipm-index.html`.

AUTOESL. 2008. Auto pilot synthesis tool. `http://www.autoesl.com/`.

BEAUCHAMP, M. J., HAUCK, S., UNDERWOOD, K. D., AND HEMMERT, K. S. 2006. Embedded floating-point units in FPGAs. In *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA'06)*. ACM, New York, NY, 12–20.

Blackford, L. S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., and Whaley, R. C. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw. 28,* 2, 135–151.

Cray Inc. 2008. http://www.cray.com.

Daga, V., Govindu, G., Gangadharpalli, S., Sridhar, V., and Prasanna, V. K. 2004. Efficient floating-point based block LU decomposition on FPGAs. In *Proceedings of the International Conference on Engineering of Reconfigureable Systems and Algorithms*.

deLorimier, M. and DeHon, A. 2005. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. 75–85.

Diersch, H. J. G. 2008. Error norm. http://www1.wasy.de/deutsch/produkte/feflow/hilfe/general/theory/whitepapers/error_norms/enornorm.html.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. 1998. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA.

Hager, W. W. 1988. *Applied Numerical Linear Algebra*. Prentice Hall, Englewood Cliffs, NJ.

Intel. 2008. Intel math kernel library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

Intel Corporation. 2008. Intel Xeon processor 5160. http://processorfinder.intel.com/Details.aspx?sSpec=SLABS.

Liang, X. and Jean, J. S.-N. 2003. Mapping of generalized template matching onto reconfigurable computers. In *IEEE Trans VLSI Syst.*. 167–174.

Lopes, A. R. and Constantinides, G. A. 2008. A high throughput FPGA-based floating point conjugate gradient implementation. In *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*. Lecture Notes in Computer Science Vol. 4943, 75–86.

Mencer, O., Morf, M., and Flynn, M. J. 1998. PAM-Blox: High performance FPGA design for adaptive computing. In *Proceedings of the 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines*. 485–498.

Mentor Graphics. 2008. Catapult synthesis. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm.

Moore, N., Conti, A., Leeser, M., and King, L. S. 2007. Vforce: An extensible framework for reconfigurable supercomputing. *Comput. 40*, 39–49.

Morris, G. R. and Prasanna, V. K. 2007. Sparse matrix computations on reconfigurable hardware. *Comput. 40,* 3, 58–64.

NVIDIA Corporation. 2011. Geforce gtx 280. http://www.nvidia.com/object/product_geforce_gtx_280_us.html.

SRC Computers. 2008. http://www.srccomp.com.

Sun, J., Peterson, G. D., and Storaasli, O. O. 2008. High-performance mixed-precision linear solver for fpgas. *IEEE Trans. Comput. 57*, 1614–1623.

Volkov, V. and Demmel, J. W. 2008. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE Press, Los Alamitos, CA, 11.

XtremeData, Inc. 2008. http://www.xtremedatainc.com.

Zhang, W. 2008. Portable and scalable FPGA-based acceleration of a direct linear system solver. M.A.Sc. Thesis, University of Toronto.

Zhang, W., Betz, V., and Rose, J. 2008. Portable and scalable FPGA-based acceleration of a direct linear system solver. In *Proceedings of the International Conference on Field-Programmable Technology*. 17–24.

Zhuo, L. and Prasanna, V. 2008. High-performance designs for linear algebra operations on reconfigurable hardware. *IEEE Trans. Comput. 57,* 8, 1057–1071.

Zhuo, L. and Prasanna, V. K. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. 63–74.

Zhuo, L. and Prasanna, V. K. 2006. High-performance and parameterized matrix factorization on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–6.

Received October 2010; revised January 2011, April 2011; accepted July 2011