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

Wei Zhang
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario
Email: weizer@eecg.toronto.edu

Vaughn Betz
Altera Corporation
Toronto, Ontario
Email: vbetz@altera.com

Jonathan Rose
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario
Email: jayar@eecg.toronto.edu

***Abstract*—FPGAs are becoming an attractive 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. We believe that FPGA-based scientific computing would become far more practical if there were hardware libraries that were portable to any FPGA with performance that could scale with size of the resources of the FPGA. To illustrate this idea we have implemented one common super-computing library function: the LU factorization method for solution of systems of linear equations. This paper discusses issues in making the design both portable (which is primarily about the ease of use of memory external to the FPGA) and scalable. The compute engine is automatically generated to match the FPGA capabilities and external memory through the use of a wide range of parameters. We compared the performance of the engine on the largest FPGA available, (an Altera Stratix III 3S340) to a single processor core fabricated in the same 65nm IC process, and show that it performs LU factorization 2.2 times faster on matrices on the order of 10,000 x 10,000 elements, and that the energy dissipated per useful GFLOP operation is a factor of 5 times less. We also note that, unless the very best software libraries from processor manufacturers are used, the FPGA results would be far better than these quoted.**

I. Introduction

As the logic and computational capacity of FPGAs have grown, FPGAs have become an attractive platform for accelerating many computations including scientific 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 [1], SRC [2], and Cray [3]. However, adoption of these FPGA accelerators by the scientific computing community has been limited because the creation of an FPGA design is difficult and time consuming and outside the skill set of the typical scientific computing user. 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 locked onto that FPGA-based platform because it typically has a specific memory architecture that soon becomes outdated.

In contrast, software is highly portable. Once a software application is completed, it can easily be upgraded to new and faster machines and obtain significantly better performance. This permits software programmers to develop and maintain rich libraries that solve important problems. Scientific 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 [4], Catapult C [5], and SystemC [6]. 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 present an alternative solution for making FPGA-based computation more accessible by creating a computational "library" that is portable to any FPGA platform with minimal effort. The key second feature of the library is that its performance should also scale 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. By creating a *portable* and *scalable* library, we can drastically reduce the development cost and increase the life span of the design, thus making it more attractive to scientific computing users.

Common software libraries for scientific computing include matrix manipulation packages such as BLAS [7], SAT solvers, and linear program solvers. If an equivalent library existed for FPGAs, it could enable broader adoption of FPGA acceleration. This research lays out a framework to create such a hardware library, by illustrating the design issues and efforts needed to build *one* such library member.

Our focus application is the solution of systems of linear equations, as this is a very common problem and the computation time is high for large systems. There are two main classes of linear equation solvers: iterative and direct. Iterative solvers either require less computation but do not guarantee converge for all types of matrices or require the same order of computation as direct solvers to guarantee convergence;

thus both iterative and direct solvers are widely used. 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. Prior work [8] 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 matrices. We have created a generator that automatically creates a portable and scalable FPGA computer engine for the LU factorization method [9] 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.

This paper is organized as follows. Section II provides background on the LU factorization method for solving linear systems and summarizes previous work on using FPGAs to accelerate matrix operations. Section III outlines the architecture of our design, including the broad space of parameters for which the tool can generate implementations. Section IV discusses the experimental results and Section V concludes.

II. 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 x 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, thus forming $LUx = b$. 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.

A. Simple LU Factorization

A pseudo-code for a simple LU factorization algorithm is given in Algorithm 1. 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 product 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.

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
    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 for
    end for
end for
```

B. Block LU Factorization

For the simple LU factorization method described above, 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 x 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 limitation to off-chip memory bandwidth. 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_bxN_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 [9]; and we will employ the "right-looking" version - in this method the current block being updated uses elements from the left-most and top-most block in its row and column as shown in Figure 1(a). With this blocking method there are four kinds of computations on the blocks: Case 1: all three blocks (current, left-most, and top-most) are the same physical block. Case 2: the current block is the same as left-most block. Case 3: the current block is the same as the top-most 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 pseudo-code 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) only perform 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.

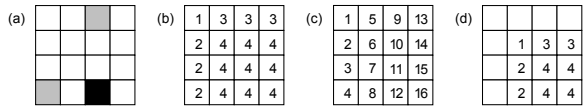

Fig. 1. (a) blocks required, in grey, to update the black block; (b) type of computation for each matrix block in the first block pass; (c) order of blocks updated in the first pass; (d) computation performed in the second block pass.

The remaining blocks, which were all the case 4 blocks in the first block pass, are updated again, repeating the above 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
   for j = k + 1 to N_b do {for each column right of current diagonal element}
      for i = 1 to N_b do {for each element below it in current block (a)}
         a_{i,j} ← a_{i,j} − a_{i,k} × u_{k,j}
      end for
   end for
end for

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 for
   end for
end for

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 for
   end for
end for
```

C. Prior Work

The work in [10] showed that a state-of-the-art FPGA has a higher peak floating-point operation performance for computing various basic linear algebra operations than a CPU, and that an FPGA also has a higher performance increase trend than CPU; therefore over time, the performance advantage should increase. However, the impact of new CPUs with multi-cores was not evaluated. There has been some prior research on implementing linear equation solvers in FPGAs. The work in [8], [11], [12] and [13] built iterative solvers using the conjugate gradient method [9]. The work in [12] reports 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 [8], 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 [8], 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 N^2 computation requires N^2 data, indicating a large amount of memory access per unit of computation.

The work in [14] 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. This work also imposed a limit on the matrix size; a blocking version to remove the matrix size limit was proposed, but not implemented, in [15]. The present work can solve systems of linear equations of any size up to the capacity of the off-chip memory of the system, which is an important feature as it is the largest matrices which most need accelerated solutions. 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.

III. Hardware Implementation

A. High Level Design Overview

Our goal is to create a highly parameterized LU factorization hardware design for single-precision 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 (we'll assume DDR2 SDRAM) to store the large input matrices. We will use the block LU factorization method described in Section II-B - 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 use the Altera Stratix III 3SL340 as the main FPGA vehicle for the computation, which has 16.7 Mbits of on-chip memory and can hold at most one 721 x 721 single precision matrix block. Furthermore, 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 external 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.

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

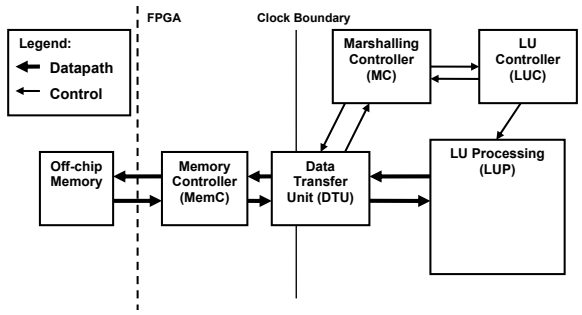

Fig. 2. Block Diagram Block-Based Linear System Solver

the LU Processing (LUP) 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 which series of operations are performed 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 connected part of the data marshalling unit; the second clock mainly controls the computation. Separating these clocks is important for scalability; it is unlikely the speed of the off-chip memory and that of the on-chip memory and computation units will scale at the same rate.

B. Ordering of Blocks and Setup of Computation

In creating our design, we observed that the time required to transfer the matrix from off-chip memory onto the chip was on the same order of the computation itself, when there are many processing elements. To gain the best computational time, it is thus necessary to simultaneously fetch data and compute on it. This requires on-chip "double-buffering" of the memory to allow one memory to do transfers 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 II-B: the current block being computed, the top-most block in the same column and the left-most block in the same row. Following the order of updating show in Figure 1(c), the top-most 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 column, which is at the start of a new column. 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 and thus, a total of 5 matrix blocks in on-chip memory is required to enable computation on one set of blocks while simultaneously pre-loading of 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 into the original input matrix in the main memory, 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 III-E. We now proceed to describe the computation and data marshalling functions in turn.

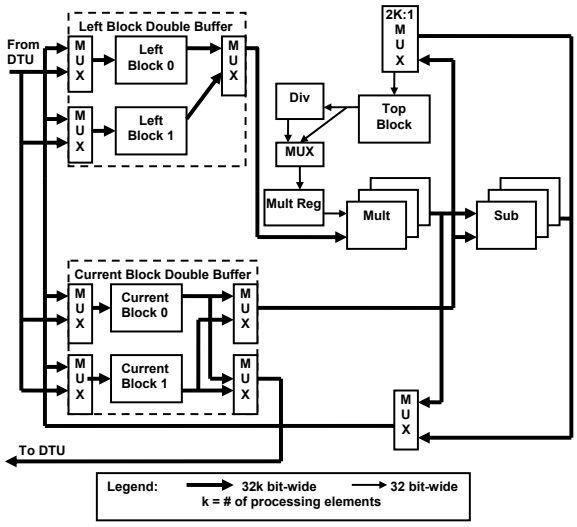

Fig. 3. Diagram of the LU Processing module which performs the computation.

C. Computation

In this section we will describe the hardware needed to implement the LU factorization computation. As described in Section II-B, 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. These are multiplication and subtraction floating-point units. 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. 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 preserved.

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 effectively compute the division. Effectively, at most we only perform one division per column.

The floating point units are generated using Altera's MegaCore IP functions [16]. The multiplication and subtraction units are fully pipelined and have a throughput of one per cycle. Since we only have to perform the division once in a while, we allow the division unit to take multiple cycles to prevent it from being 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. Typically, k is on the order of 100, therefore these on-chip memories are very wide, on the order of 3200 data bits. 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 multiplier or the one division operator, and therefore the top block has a data width of 32 bits for single precision.

D. 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 how the matrix blocks need to be stored on the on-chip memory. 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 [16]. 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 suitable size read or write commands to 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 operation of the speed of the off-chip memory to be independent of the speed and bandwidth (number of bits of width) of the on-chip memory. Speed and external memory bus width are two key parameters to the compute engine generator. The decoupling is accomplished through the use of the FIFOs illustrated in the figure. The left hand side of the FIFOs operate at the external memory clock speed, while the right hand side operates on the compute engine's clock speed.

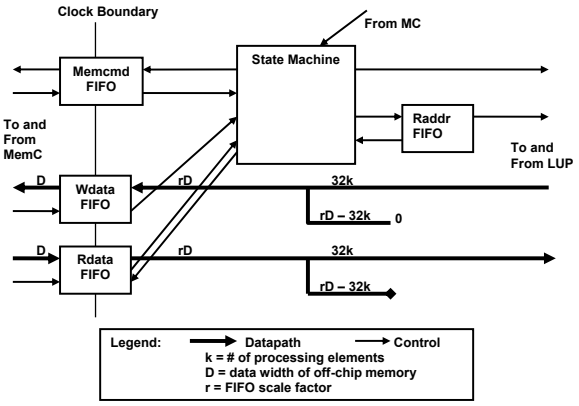

Fig. 4. Diagram of the Data Transfer Unit which performs the data marshalling.

The FIFOs in the DTU are generated using Altera's multiple clock FIFOs MegaCore IP functions, which have parameterized input and output data width. One side of the FIFO has to match the data width of the MemC and external memory itself, which will be 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. 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 FPGA. We decided to waste the extra bits by padding it with zeroes. 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. The user is required to prepare the input matrix by adding the necessary padding.

TABLE I
A SUBSET OF THE PARAMETERS USED TO GENERATE LU FACTORIZATION

Name	Description
k	Number of processing elements
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 (in bits)
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

E. Portability and Scalability in Design

Portability and scalability is achieved by having the compute engine adapt to the available resources of the FPGA and the specific external memory used. To be portable, the engine must move to a new FPGA and external memory interface with minimal human effort, and to be scalable, the engine must 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 generator which can create an HDL design implementation that matches the desired parameters. Table I shows a subset of the parameters used as input to the compute engine generator. Some of the parameters deal with the variability of resources on the FPGA while some other parameters deal with the external memory. In addition, some parameters are used to optimize the engine by trading area for clock speed.

We found that Hardware Description Languages, such as Verilog, were not sufficiently powerful to fully adapt the compute engine to all the parameters. Consequently, we implement our generator as software (written in C language) that generate HDL code in Verilog. The compute engine consists of automatically created HDL code from the generator and cores generated from the Altera MegaCore IP functions.

One of the key portability and scalability parameters is the number of processing elements. The generator can be set to produce an engine with any number of processing elements. As more FPGA real estate is available on a chip (or in moving to a larger chip) the number can be increased, improving performance. Other parameters include choosing the pipeline latency of the floating point units. Most FPGA vendors offer floating point unit implementations with a range of latencies to allow area/speed tradeoffs, and by exposing this latency parameter we preserve this tradeoff in our compute engine. In the future, if a better floating-point unit with a different latency is created, the generator will automatically take care of it based on this parameter.

We achieve portability and scalability of the external memory interface by (1) parameterizing key aspects of the memory interface and (2) ensuring there is a clean divide between our design and the off-chip memory controller including different clock domain. Table 1 lists some of the memory interface parameters – DDR2 data width, address width, and burst length. 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 memory controller and data is transferred to the on-chip memory at the appropriate times. Parameterizing the memory interface in this way allows the user to use DDR2 SDRAM of various data widths and speeds without any redesign.

Moving to a different (non-DDR2) memory technology is very low effort if the vendor-supplied off-chip memory controller has the same interface to the FPGA logic as that of the DDR2 off-chip memory controller we use. If the memory controller has a different interface, some bridging hardware must be designed; this is akin to creating a new device driver in the software world.

By using different clock domains for the external memory interface and the main computation units, we can run each part of our design 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 a scalable design.

IV. RESULTS

The computation described in Section II-B was implemented in the architecture described in Section III-A, employing a mixture of coding in Verilog and C to implement the wide range of parameters given in Table I and quite a number of others. This hardware design was targeted to a Stratix III 3SL340F1760C3 FPGA. We assume that the FPGA is attached to off-chip DDR2 SDRAM of size 256MB and 64bit wide. We are able to create many versions of the design by changing various parameters; this allows us explore the trade-offs in the resulting designs. For example, performance increases as the number of processing elements (k) increases, up to the point where the occupancy of the FPGA is so large that the tools have difficulty optimizing operating frequency, thereby making overall performance suffer. For the key comparisons presented below, we selected the best performing FPGA implementation by experimenting with a wide range of parameters (but clearly k is the most important one) and choosing the best.

The top-performing design employs 120 processing elements and achieves a maximum operating frequency of 200MHz. This design was compared to a software version from the Intel highly vendor-optimized MKL library [17] and more basic code running on an Intel Xeon 5160 dual core 3.0 GHz 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. The more basic code is single-threaded, and it is modeled after the pseudo-code in Section II-A. (We include a performance comparison to this more basic code to show how impressive the vendor-optimized software is).

For each platform/code type, we measure the performance in useful GFLOPS (the number of floating-point operations per second used in the LU factorization calculation), which is

TABLE II
PERFORMANCE ON 65 NM FPGA AND PROCESSOR PLATFORMS

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 III
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

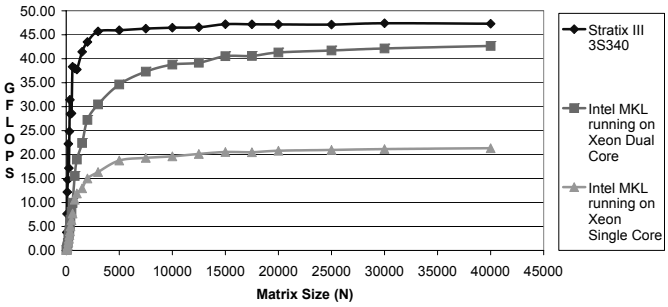

Fig. 5. Performance as a Function of Matrix Dimension.

calculated by taking the total number of operations needed to solve the LU factorization and dividing by the total runtime. The total runtime is calculated by multiplying the number of cycles required to perform the computation (as measured by running the design in the Modelsim logic simulator) times the cycle time as determined by post-placement and routing timing analysis performed by Altera's Quartus II version 7.2.

Table II gives the performance of several platforms, with all chips fabricated in the same 65nm IC process. The first column lists the platforms, including the the top-performing Stratix III 3S340 FPGA implementation described above, a single core Intel Xeon processor, the dual core (both running the optimized MKL version) and the single core running the more basic code. The table also states 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, which is perhaps a more fair comparison as the second core resides on the same chip. 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 II-B) 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 II. 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 learning 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 given by Intel to produce this optimized library, perhaps not unlike our effort to create the FPGA-based design.

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 III shows the power consumption of each of the platforms listed in Table II 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 not simulated, this method of measuring is believed to be within 20% of the actual power consumption. The power consumption of the Xeon dual core processor was determined from the specification on the Intel website [18], which should be close to the actual power since the MKL 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 the table, 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.

A. Matrix Size

We observed that many previous works in this area 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 RAM and therefore is much more widely applicable. Figure 5 measures the performance (in GFLOPS) for the various platforms as a function of Matrix dimensions, N. Here you can see that the performance for all platforms eventually levels out and reaches a maximum as matrix size increases. We use these leveled-off performance values in our comparison. The FPGA implementation is able to reach its maximum achievable performance faster and thus, there is a larger speed-up when comparing for small matrices.

B. Scalability

A key aspect of our design goals is scalability - having a compute engine generator that can grow the engine to take advantage of more FPGA logic resources as they become

TABLE IV
PERFORMANCE RATIO FOR VARIOUS COMPUTING ENGINE

Number of Processing Elements	Clock Frequency	GFLOPS	Performance Ratio
30	240 MHz	14	1.00
60	220 MHz	26	1.9
90	215 MHz	38	2.7
120	200 MHz	47	3.4

available. To illustrate the scalability of our system, we implemented four versions of the LU factorization design with 30, 60, 90 and 120 processing elements. Table IV gives the performance in GFLOPS for each number of processing elements (k). The fourth column in the table gives the performance relative to the 30 processing element compute engine. With more processing elements, the engine can compute more operations per cycle. However, this also results in a decrease in the clock frequency as more resources on the FPGA are used, resulting in a loss of operating frequency due to congenstion. In general, the performance increases with the number of processing elements, thus achieving scalability. We also expect that the performance of a multi-core processor chip will scale with the number of cores on the chip, and thus we expect the performance gap to remain consistent.

C. Caveats and Future Work

While one of key goals was to provide a core that is portable, there are a few restrictions that limit its portability. To reduce development time, we used pre-designed cores from Altera (that are vendor-specific and therefore not portable to other vendor's 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 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.

While our current design only solves 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 blocks can be reused for any algorithm. 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.

V. CONCLUSION

In this paper, we have shown how to create a portable and scalable computational engine generator for the LU factorization method of solution of linear systems that should make it easy to employ FPGAs in supercomputing applications. We have shown that the generated compute engine has significant performance and performance per watt advantages over a single-core processor, but not nearly as large as expected when we compared to the vendor-optimized software library for the same computation. Our scalable FPGA core is 2.2 times faster than a single core processor (built in the same IC fabrication process) and 5 times more power efficient.

ACKNOWLEDGMENT

The authors are grateful to Altera and an NSERC CRD grant for funding this research, as well as an NSERC post-graduate scholarship.

REFERENCES

[1] XtremeData, Inc. http://www.xtremedatainc.com.
[2] SRC Computers, Inc. http://www.srccomp.com.
[3] Cray Inc. http://www.cray.com.
[4] Agility Design Solutions Inc. Handel-C. http://www.agilityds.com/products/c_based_products/dk_design_suite/handel-c.aspx.
[5] Mentor Graphics Corporation. Catapult Synthesis. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm.
[6] Open SystemC Initiative. http://www.systemc.org.
[7] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Soft.*, vol. 28, no. 2, pp. 135–151, 2002.
[8] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, 2005, pp. 63–74.
[9] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*. Philadelphia, PA: SIAM, 1998.
[10] K. D. Underwood and K. S. Hemmert, "Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 219–228.
[11] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, 2005, pp. 75–85.
[12] G. R. Morris and V. K. Prasanna, "Sparse matrix computations on reconfigurable hardware," *Computer*, vol. 40, no. 3, pp. 58–64, March 2007.
[13] A. R. Lopes and G. A. Constantinides, "A high throughput fpga-based floating point conjugate gradient implementation," *Lecture Notes in Computer Science*, no. 4943, pp. 75–86, 2008.
[14] L. Zhuo and V. K. Prasanna, "High-performance and parameterized matrix factorization on fpgas," in *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*, 2007, pp. 1–6.
[15] V. Daga, G. Govindu, S. Gangadharpalli, V. Sridhar, and V. K. Prasanna, "Efficient floatin-point based block lu decomposition on fpgas," in *Proceedings of the International Conference on Engineering of Reconfigureable Systems and Algorithms*, 2004.
[16] Altera Corporation. http://www.altera.com/products/ip/ipm-index.html.
[17] Intel. Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.
[18] Intel. Intel Xeon Processor 5160. http://processorfinder.intel.com/Details.aspx?sSpec=SLABS.