An Independent Analysis
of

Floating-point DSP Design Flow and Performance on Altera 28-nm FPGAs

By the staff of

Berkeley Design Technology, Inc.

October 2012

OVERVIEW

FPGAs are increasingly used as parallel processing engines for demanding digital signal processing applications. Benchmark results show that on highly parallelizable workloads, FPGAs can achieve higher performance and superior cost/performance compared to digital signal processors (DSPs) and general-purpose CPUs. However, to date, FPGAs have been used almost exclusively for fixed-point DSP designs. FPGAs have not been viewed as an effective platform for applications requiring high-performance floating-point computations. FPGA floating-point efficiency and performance has been limited due to long processing latencies and routing congestion. In addition, the traditional FPGA design flow, based on writing register-transfer-level hardware descriptions in Verilog or VHDL, is not well suited to implementing complex floating-point algorithms.

Altera has developed a new floating-point design flow intended to streamline the process of implementing floating-point digital signal processing algorithms on Altera FPGAs, and to enable those designs to achieve higher performance and efficiency than previously possible. Rather than building a datapath consisting of elementary floating-point operators (for example, multiplication followed by addition followed by squaring), the floating-point compiler generates a fused datapath that combines elementary operators into a single function or datapath. In doing so, it eliminates the redundancies present in traditional floating-point FPGA designs. In addition, the Altera design flow is a high-level, model-based design flow using Altera's DSP Builder Advanced Blockset with MATLAB and Simulink from MathWorks. Altera expects that by working at a high level, FPGA designers will be able to implement and verify complex floating-point algorithms more quickly than would be possible with traditional HDL-based design.

BDTI performed an independent analysis of Altera's floating-point DSP design flow. BDTI's objective was to assess the performance that can be obtained on Altera FPGAs for demanding floating-point DSP applications, and to evaluate the ease-of-use of Altera's floating-point DSP design flow. This paper presents BDTI's findings, along with background and methodology details.

WP-01187-1.0

Contents

NOTATION AND DEFINITIONS

M Bold capital letter denotes a matrix.
z Bold small letter denotes a vector.

L^* The conjugate transpose of matrix L.
l^* The conjugate transpose of element l.

Hermitian Matrix A square matrix with complex entries that is equal to its own conjugate transpose. This is the complex extension to a real symmetric matrix.

Positive Definite Matrix A Hermitian matrix **M** is positive definite if $\mathbf{z}^*\mathbf{Mz} > 0$ for all non-zero complex vectors **z**. The quantity $\mathbf{z}^*\mathbf{Mz}$ is always real because **M** is a Hermitian matrix for the purposes of this paper.

Orthonormal Matrix A matrix **Q** is orthonormal if $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ where **I** is the identity matrix.

Cholesky Decomposition A factorization of a Hermitian positive definite matrix **M** into a lower triangular matrix **L** and its conjugate transpose **L*** such that $\mathbf{M} = \mathbf{LL}^*$.

QR Decomposition A factorization of a matrix **M** of size *m*-by-*n* into an orthonormal matrix **Q** of size *m*-by-*n* and an upper triangular matrix **R** of size *n*-by-*n* such that $\mathbf{M} = \mathbf{QR}$.

F_{max} The maximum frequency of an FPGA design.

1. Introduction

Two Floating-point Design Examples

Advances in digital chips are enabling complex algorithms that were previously limited to research environments to move into the realm of everyday embedded computing applications. For example, for many years, linear algebra (and specifically solving for systems involving large sets of simultaneous linear equations) has been used mainly in research environments, where large-scale compute resources are available and real-time computation is usually not required. Solving for large systems involves either matrix inversion or some kind of matrix decomposition. In addition to being computationally demanding, these techniques can suffer from numeric instability if sufficient dynamic range is not used. Therefore, efficient and accurate implementation of such algorithms is only practical in floating-point devices.

Altera recently introduced floating-point capability in the DSP Builder Advanced Blockset tool chain to simplify implementation of floating-point DSP algorithms on Altera FPGAs, while improving performance and efficiency of floating-point designs compared to traditional FPGA design techniques. In a previous white paper [1], BDTI analyzed and evaluated the performance and efficiency of the Altera Quartus II software v11.0 tool chain on a single-channel floating-point Cholesky solver implementation example, synthesized for the 40-nm Stratix IV and Arria IV FPGAs.

In this paper, we evaluate the effectiveness of Altera's approach using the newer Quartus II software v12.0 tool chain and assess the performance of Altera's 28-nm Stratix V and Arria V FPGAs. For this evaluation, BDTI focused on solving a large set of simultaneous linear equations using two types of decompositions: a multi-channel Cholesky matrix decomposition and the QR decomposition using the Gram-Schmidt process. These decompositions combined with forward and backward substitutions constitute a solution for the vector **x** in a simultaneous set of linear equations of the form $\mathbf{Ax} = \mathbf{B}$.

Matrix decomposition is used in many advanced military radar applications such as Space-Time Adaptive Processing (or STAP) and various estimation problems in digital communications. The QR decomposition is commonly used for any general m-by-n matrix **A**, while the Cholesky decomposition is the preferred algorithm for a square, symmetric, and positive definite matrix for its high computational efficiency. Both decompositions use very computationally demanding algorithms and require high data precision, making floating-point math a necessity. In addition, the two examples studied in this paper use vector dot products and nested loops at the core of their algorithm, and these operations are found in a wide range of digital signal processing applications involving linear algebra and finite impulse response (FIR) filters.

In the examples described in this paper, simultaneous sets of complex-data linear equations are solved in both methods and the results are reported. Using the QR solver as shown in Section 4, an Altera Stratix V FPGA is capable of performing 315 matrix decompositions

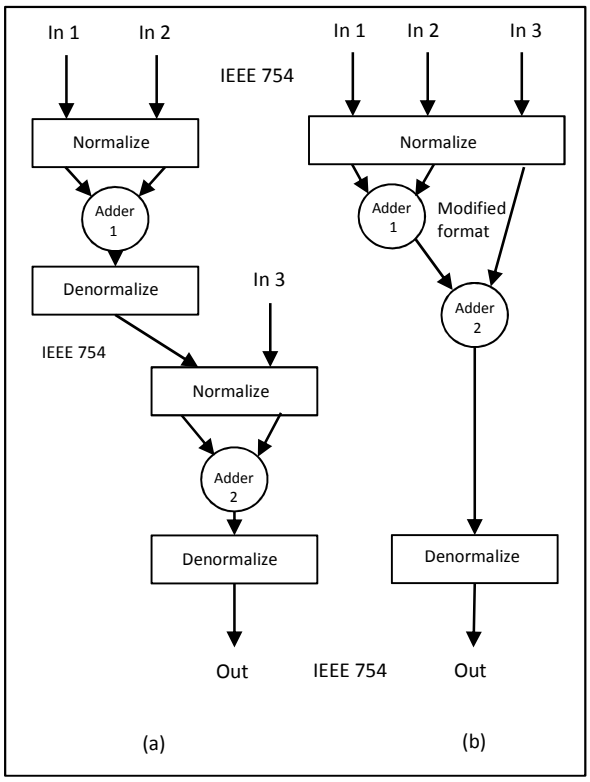

Figure 1 (a) Traditional floating-point implementation (b) Fused datapath

per second of size 400×400, running at 203 MHz and achieving 162×10^9 floating-point operations per second (GFLOPS).

Both design examples evaluated in this paper are provided as part of the DSP Builder software v12.0 tool chain, or they may be requested from floatingpoint@altera.com.

Floating-point Design Flow

Traditionally, FPGAs have not been the platform of choice for demanding floating-point applications. Although FPGA vendors have offered floating-point primitive libraries, the performance of FPGAs in floating-point applications has been very limited. The inefficiency of traditional floating-point FPGA designs is partially due to the deeply pipelined nature and wide arithmetic structures of the floating-point operators, which create large datapath latencies and routing congestion. In turn, the latencies create hard-to-manage problems in designs with high data dependencies. The final result is often a design with a low operating frequency.

The Altera DSP Builder Advanced Blockset tool flow attacks these issues at both the architectural level and the system design level. The Altera floating-point compiler fuses large portions of the datapath into a single floating-point function instead of building them up by composition of elemental floating-point operators. It does this by analyzing the bit growth in the datapath and choosing the optimum input normalization to allocate enough precision through the datapath in order to eliminate as many normalization and denormalization steps as possible, as shown in Figure 1. The IEEE 754 format is only used at datapath boundaries; within datapaths, larger mantissa widths are used to increase the dynamic range and reduce the need for denormalization and normalization steps between successive operators. Normalization and denormalization functions use barrel shifters of up to 48 bits wide for a single precision floating-point number. This consumes a significant amount of logic and routing resources and is the main reason why floating-point implementations on FPGAs have not been efficient. The *fused datapath* methodology eliminates a significant number of these barrel shifters. Multiplications involving the larger precision mantissas use Altera's 27-bit×27-bit multiplier mode in Stratix V and Arria V devices. Figure 1(b) shows the fused datapath methodology for the simple case of a two-adder chain as compared to the traditional implementation shown in Figure 1(a). The fused datapath in Figure 1(b) eliminates the inter-operator redundancy by removing the denormalization of the output of the first adder and the normalization of the input of the second adder. The elimination of the extra logic and routing, and the use of hard multipliers make timing and latency across complex datapaths predictable. Both single- and double- precision IEEE 754 floating-point algorithms are implemented with reduced logic and higher performance. Altera claims that a fused datapath contains 50% less logic and 50% less latency than the equivalent datapath constructed out of elementary operators [2]. And, as a result of the wider internal data representation, typically the overall data accuracy is higher than that achieved using a library with elementary IEEE 754 floating-point operators.

The Altera floating-point DSP design flow incorporates the Altera DSP Builder Advanced Blockset, Altera's Quartus II software tool chain, the ModelSim simulator, as well as MATLAB and Simulink from MathWorks. The Simulink environment allows the designer to operate at the

algorithmic behavioral level to describe, debug, and verify complex systems. Simulink features such as data type propagation and vector data processing are incorporated into the DSP Builder Advanced Blockset, enabling a designer to perform quick algorithmic design space exploration.

In order to evaluate the efficiency and performance of Altera's floating-point design flow, BDTI used the DSP Builder Advanced Blockset tool flow to validate two examples, both solving a set of simultaneous linear equations using complex data-type in single-precision floating-point representation. One solution is achieved using the Cholesky decomposition, while the other uses QR decomposition via the Gram-Schmidt process. Section 2 of the paper describes the implementation of the two floating-point examples. Section 3 presents BDTI's experience with the design flow and tool chain. Section 4 presents the performance of the implementation on two different Altera FPGAs: the high-end medium-size Stratix V 5SGSMD5K2F40C2N device and the mid-range Arria V 5AGTFD7K3F40I3N device. Finally, Section 5 presents BDTI's conclusions.

2. Implementation

Background

Sets of linear equations of the form $\mathbf{Ax} = \mathbf{b}$ arise in many applications. Whether it is an optimization problem involving linear least squares, a Kalman filter for a prediction problem, or MIMO communications channel estimation, the problem remains one of finding a numerical solution for a set of linear equations of the form $\mathbf{Ax} = \mathbf{b}$. For a general matrix of size *m*-by-*n*, where *m* is the height of the matrix and *n* its width, QR decomposition may be used to solve for vector **x**. The algorithm decomposes **A** into an orthonormal matrix **Q** of size *m*-by-*n* and an upper triangular matrix **R** of size *n*-by-*n*. Since **Q** is orthonormal, $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ and $\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$. Given that **R** is an upper triangular matrix, **x** can easily be solved by backward substitution without even inverting the original matrix **A**. In the QR example in this paper we work with over-determined matrices with $m \geq n$.

When matrix **A** is symmetric and positive definite, such as covariance matrices that arise in many problems, the Cholesky decomposition and solver are commonly used. The algorithm finds the inverse of matrix **A** thus solving for vector **x** in $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. The Cholesky decomposition is at least twice as efficient as **QR** decomposition depending on the algorithms used for the decomposition. Since these decomposition algorithms are recursive in nature and involve division, a large numeric dynamic range becomes a necessity as the matrix size increases. Most implementations, even for matrix sizes as small as 4×4 in MIMO channel estimation, for example, are performed using floating-point operations. For larger systems requiring high throughput, such as the ones found in military applications, the required floating-point operation rate has typically been prohibitive for embedded systems. Frequently, designers either abandon the whole algorithm for a sub-optimum solution or resort to using multiple high-performance floating-point processors, raising cost and design effort.

Architectural Overview

Cholesky Solver

In our design example, the Cholesky solver is implemented in the FPGA as two subsystems operating in parallel in a pipelined fashion. The first subsystem executes the Cholesky decomposition and forward substitution—steps 1 and 2 in the sidebar titled *The Cholesky Solver*. The second subsystem executes the backward substitution—step 3 in the same sidebar. Since the input matrix is Hermitian and the decomposition generates complex conjugate transposed triangular matrices, memory utilization is optimized by loading only the lower triangular half of the input matrix **A** and overwriting it as the lower triangular matrix **L** is generated. Both subsystems are pipelined, utilizing an input stage and a processing stage to allow processing to occur in one area of a memory while the other half is used for loading new data. The output of the decomposition and forward substitution stage goes into the input stage of the backward substitution, as shown in Figure 2.

The core of the decomposition is the complex vector dot product engine (also referred to as the vector multiplier) which computes equations (3) and (4). For the Stratix V device, a vector size (VS) of up to 90 complex-data elements is used, whereas for the Arria V device, a vector size of up to 45 complex-data elements is implemented. The vector size also corresponds to the number of parallel memory reads needed to supply the dot product engine with a new set of data every clock cycle and thus determines the width and

partitioning of the on-chip dual-port memory. For implementation reasons, the storage of a matrix of a given size is partitioned into *ceil (N/VS)* memory banks, where *ceil()* is the ceiling function and *N* is the size of the matrix.

The Cholesky solver design is a multi-channel implementation. The maximum number of channels is a compile-time parameter and is limited only by the available memory on the device. The memory partitioning is identical to a single-channel implementation except that multiple copies of the same structure are used.

The decomposition is performed one element at a time, column-wise, starting from the top left corner, proceeding in a vertical zigzag fashion, and ending at the bottom right corner of the matrix as shown in Figure 3(a). The diagonal element of each column is calculated first, followed by all the non-diagonal elements below it in the same column before moving to the diagonal element at the top of the next column to the right. The schedule of events and iterations is controlled with a four-level nested *for* loop. The outermost loop implements column-wise processing; the second loop implements bank-wise processing; the third inner loop processes the rows; and the innermost loop processes the multiple channels. Placing the channel processing as the innermost loop effectively turns the floating-point accumulator into a time-shared accumulator and hides its latency better. The *NestedLoops* block in the DSP Builder Advanced Blockset integrates up to three nested loops in a single processing block, making the implementation faster and more resource efficient when compared to a similar function that is implemented as three separate *for-loop* blocks. The block abstracts away the intricate loop control signals, reduces design and debug times, and makes the overall loop structure more readable.

The dot product engine operates on the rows of the matrix and calculates up to vector size multiplications in the summation term of equations (3), (4), and (6) simultaneously in one cycle. A circular memory structure is used at the input of the dot product engine to cycle over the rows of the multiple input matrices. For vector dot products shorter than vector size, unused terms are masked out and are not included in the summation. For dot products longer than vector size, partial sums of products are calculated and saved at bank boundaries until the output of all banks for a given element in that row are available for a final summation, as shown in Figure 3(b). The summation of the bank outputs is performed in a single accumulator loop using the floating-

THE CHOLESKY SOLVER

The recursive Cholesky algorithm used to solve for vector **x** in **Ax** = **b** has three steps:

Step 1. Decomposition, i.e. finding the lower triangular matrix **L**, where **A** = **LL***

$$l_{11} = \sqrt{a_{11}} \quad (1)$$

for i = 2 to n,

$$l_{i1} = {}^{a_{i1}}/_{l_{11}} \quad (2)$$

for j = 2 to (i-1),

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} \times l_{jk}{}^*)/l_{jj} \quad (3)$$

end

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1}(l_{ik} \times l_{ik}{}^*)} \quad (4)$$

end

Note the dependencies in the equations above. The diagonal elements in eq. (4) depend only on elements to their left in the same row. Non-diagonal elements depend on elements to their left in the same row, and on the elements to the left of the corresponding diagonal element above them.

Step 2. Forward substitution, i.e. solving for **y** in the equation **Ly** = **b**,

$$y_1 = b_1/l_{11} \quad (5)$$

for i = 2 to n,

$$y_i = (b_i - \sum_{k=1}^{i-1} y_k \times l_{ik})/l_{ii} \quad (6)$$

end

Step 3. Backward substitution, i.e. solving for **x** in the equation **L*** **x** = **y**,

$$x_n = y_n/l_{nn}^* \quad (7)$$

for i = n-1 to 1,

$$x_i = (y_i - \sum_{k=i+1}^{n} x_k \times l_{ik}^*)/l_{ii}^* \quad (8)$$

end

where,

n = the dimension of matrix **A**
l_{ij} = element at row i column j of matrix **L**
a_{ij} = element at row i column j of matrix **A**
y_i = element at row i of vector **y**
b_i = element at row i of vector **b**
x_i = element at row i of vector **x**

The output of step 1 is the Cholesky decomposition, and the output of step 3 is the solution **x** of the linear equation **Ax** = **b**. Note that the algorithm indirectly finds the inverse of matrix **A** to solve for **x** = **A**$^{-1}$ **b**.

point adder block from the DSP Builder Advanced Blockset. This feedback loop has a latency of 13 cycles. By swapping the *for Banks* loop and the *for Rows* loop relative to the order that one would traditionally have in a software implementation, and adding multi-channel processing, the floating-point accumulator latency is hidden and hardware utilization is improved. The DSP Builder Advanced Blockset automatically calculates loop delays to address this type of latency. The tool can be instructed to calculate and fill in the minimum delay by checking the *Minimum delay* box in the *Loop Delay* block. Paths that incur identical delays may be assigned an equilvalence group number and the tool will then assign the same delay value to all members in the group. Although not used in the examples evaluated in this paper, the DSP Builder Advanced Blockset provides an Application Specific Floating-Point Accumulator where the user can customize the accumulator to optimize its speed and resource requirements by configuring parameters such as maximum input size and required accumulation accuracy. This block allows for single-cycle-per-sample accumulation of a single stream of floating-point numbers at a high clock rate.

The second subsystem performs backward substitution. This subsystem has its own input and output memory blocks. Like the Cholesky forward substitution subsystem, it is pipelined into an input stage and a processing stage. Since the complexity of the backward substitution is on the order of N^2 compared to N^3 for the decomposition, vector processing is not employed for the dot product. Instead, a single complex multiplier is used for the dot product which is enough to keep pace with the Cholesky decomposition and the forward substitution subsystem.

QR Solver

The QR decomposition and solver is implemented as two subsystems operating in parallel in a pipelined fashion as shown in Figure 4. The first subsystem executes steps 1 and 2 in the sidebar titled *The QR Solver*, whereas the second subsystem executes the backward substitution in step 3. The backward substitution subsystem is identical to the one used in the Cholesky solver. In contrast to Cholesky where equations (1) to (6) are implemented as a single four-deep nested loop, the QR solver employs a simple finite state machine (FSM) to cycle through four main operations; finding the magnitude square of a vector in equation (2), the dot product

THE QR SOLVER

Solving for **x** in **Ax** = **b** via QR decomposition using the Gram-Schmidt process has three steps:

Step 1. Decomposition of matrix **A** of size *m*-by-*n*, i.e. finding the matrices **Q** of size *m*-by-*n*, and **R** of size *n*-by-*n*, where **A** = **QR**.

$$\mathbf{u}_1 = \mathbf{a}_1 \quad (1)$$

for k = 1 to n,

$$r\_sq_{kk} = \sum_{j=1}^{m} u_{jk}^* \times u_{jk} \quad (2)$$

$$r_{kk} = \sqrt{r\_sq_{kk}} \quad (3)$$

for i = (k+1) to n, and k < n,

$$r_{ki} = \frac{1}{r_{kk}} \sum_{j=1}^{m} u_{jk}^* \times a_{ji} \quad (4)$$

end

t = k + 1, and k < n, (5)

for i = 1 to m, and k < n,

$$u_{it} = a_{it} - \sum_{j=1}^{k} r_{jt} \times u_{ij} / r\_sq_{jj} \quad (6)$$

end

end

Note that the orthonormal matrix **Q** = [$\mathbf{q}_1$ $\mathbf{q}_2$ …$\mathbf{q}_n$], is not explicitly calculated but rather its orthogonal form, **U** = [$\mathbf{u}_1$ $\mathbf{u}_2$ … $\mathbf{u}_n$], is found and used in this restructured set of equations, where $\mathbf{q}_i = \mathbf{u}_i / \|\mathbf{u}_i\|$.

Step 2. Calculating **d**, where **d** = $\mathbf{Q}^T\mathbf{b}$.

for k = 1 to n,

$$d_k = \sum_{j=1}^{m} u_{jk}^* \times b_j / r_{kk} \quad (7)$$

end

Step 3. Backward substitution, i.e. solving for **x** in the equation **Rx = d**.

for i = n-1 to 1,

$$x_i = (d_i - \sum_{k=i+1}^{n} r_{ik}^* \times x_k) / r_{ii}^* \quad (8)$$

end

where,

m = the row dimension of matrix A
n = the column dimension of matrix **A**
$\mathbf{u}_i$ = i^{th} column of matrix **U**
u_{ij} = element at row *i* column *j* of matrix **U**
r_{ij} = element at row *i* column *j* of matrix **R**
$\mathbf{a}_i$ = i^{th} column of matrix **A**
a_{ij} = element at row *i* column *j* of matrix **A**
d_i = element at row *i* of vector **d**
b_i = element at row *i* of vector **b**
x_i = element at row *i* of vector **x**

The QR solver finds the solution **x** of the linear equations **Ax** = **b** without finding the inverse of matrix **A** which may be undefined.

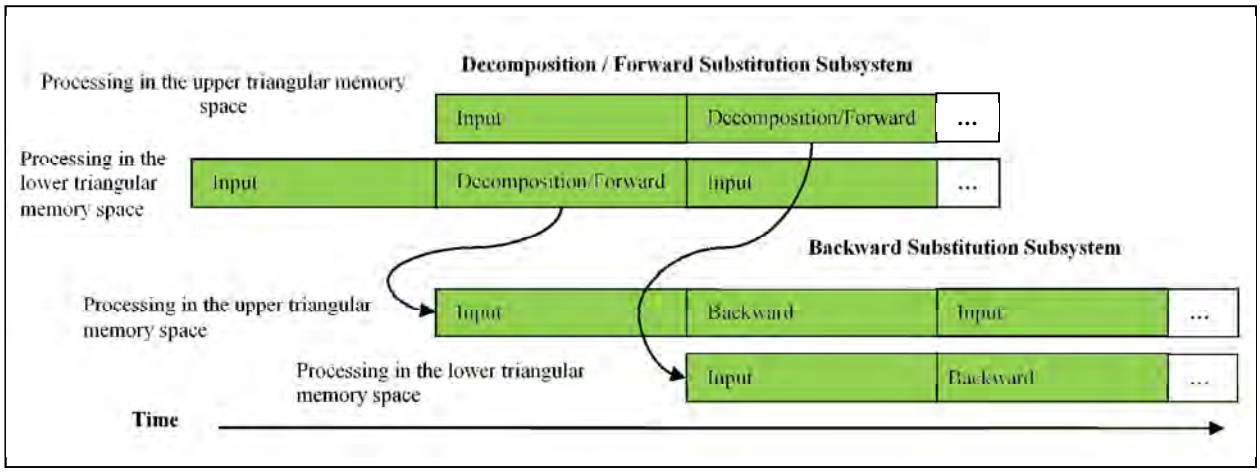

Figure 2 Cholesky process pipelining and memory reuse

of two vectors in equation (4), subtracting a dot product from a vector in equation (6), and the dot product in equation (7) to find vector **d**. The *NestedLoop* block in the DSP Builder Advanced Blockset is used in all phases of the FSM to generate all control and event signals for the datapath.

The common processing block in the four operations listed above is the dot product engine. In order to increase performance, vector processing is used to calculate the dot product. Similar to the Cholesky design, the vector size is a compile-time parameter. To reuse this engine in all four states of the FSM, a data multiplexer is used at its input and controlled by the FSM event controller. The multiplexer selects the correct inputs for the dot product engine for each state of the FSM. The details of the dot product engine and the floating-point accumulator are similar to those in the Cholesky design and hence are not presented here.

The memory needed for the QR decomposition is optimized by reusing the main core memory block that initially holds matrix **A** and input vector **b**. Processing happens column-wise, left to right, in the core memory. Once a column is consumed and no longer needed, it is overwritten by the corresponding column of a new matrix. By the time the old matrix **A** is decomposed, the original contents of this memory block are replaced by the new matrix, thus maintaining the capability of back-to-back matrix processing without any stalls.

In the first two states of the FSM, elements of the **R** matrix are generated element-by-element row-wise starting with the diagonal element for each row. In the subtraction state of the FSM, the columns of matrix **A** in the core memory are recursively updated and replaced by the partially calculated vectors. For example, at column *k*, all columns *k+1* to *n* are updated in memory by subtracting the scaled version of from each of

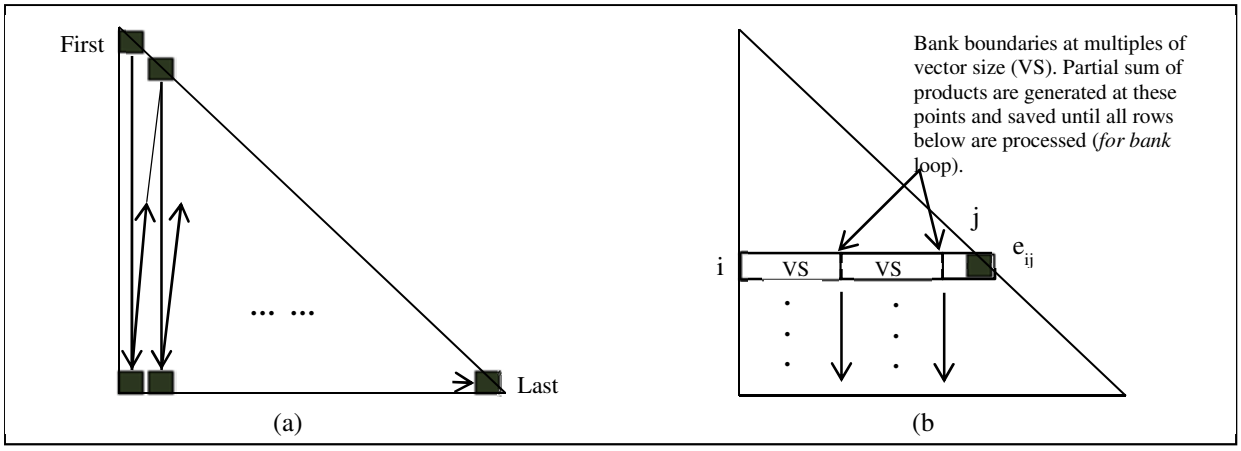

Figure 3 (a) Processing sequence (b) Computation of diagonal element e_{ij} includes two partial sum-of-products at j=VS and j=2*VS, plus the last remaining partial dot product section

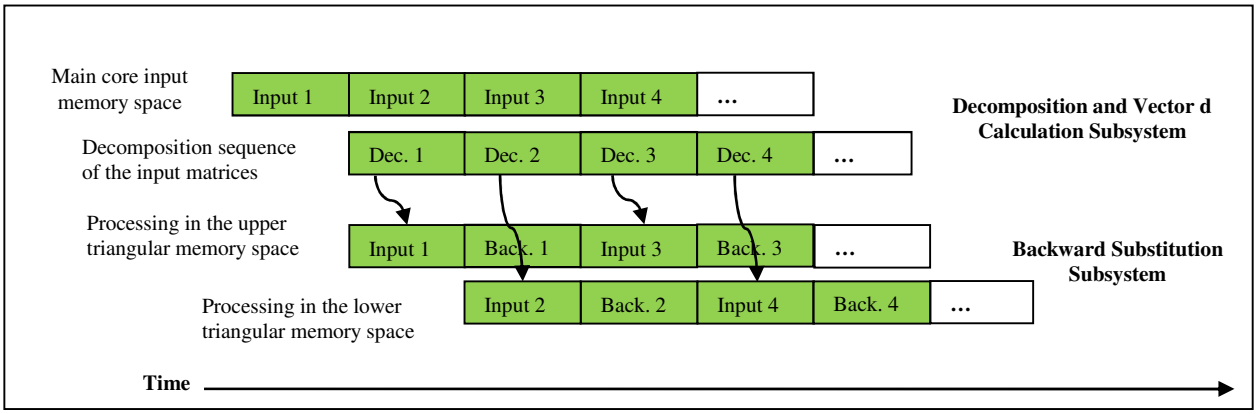

Figure 4 QR process pipelining and memory reuse

the columns *k+1* to *n*. This process recursively calculates the vectors $\boldsymbol{u}_{k+1}$ to $\boldsymbol{u}_n$ and is a more efficient version of equation (6) for hardware implementation. Processing column-wise, in the fourth state of the FSM, each newly calculated $\boldsymbol{u}_k$ is multiplied with input vector **b** to generate a single element d_k of column vector **d**. The **Q** matrix is not explicitly generated, but its orthogonal columns, $\boldsymbol{u}_k$, are generated, used in subsequent phases of the FSM, and overwritten by the corresponding column, $\boldsymbol{a}_k$, of the new matrix in the next cycle of the FSM. Figure 5 shows the memory organization and processing sequence of the QR decomposition subsystem.

The outputs of the first subsystem are the **R** matrix and the column vector **d**. The **R** matrix is an upper triangular matrix and is generated row-wise left to right as shown in equations (3) and (4). A rectangular memory structure is used in a ping-pong manner. While the upper triangular section is being processed by the backward substitution subsystem, the lower triangular section is getting filled by the decomposition subsystem, and vice versa. The **d** column vector is appended to the **R** matrix and is generated row-wise top to bottom. The output of the backward substitution, is the solution vector **x** of the linear equations **Ax** = **b**.

The QR solver may be implemented in a multi-channel format similar to the Cholesky solver to improve utilization, reduce latency, and increase the throughput of the design. The throughput improvement would mainly come from the effective reduction in the latency of the floating-point accumulator.

3. Design Flow and Tool Chain

Evaluation Methodology

For this evaluation, Altera provided BDTI with implementations of the Cholesky and the QR solvers created using the DSP Builder Advanced Blockset. Altera provided BDTI engineers a PC with the necessary Altera and MathWorks tools installed. BDTI engineers then examined the Altera designs, simulated and synthesized them, all under the Simulink environment. Additionally, the

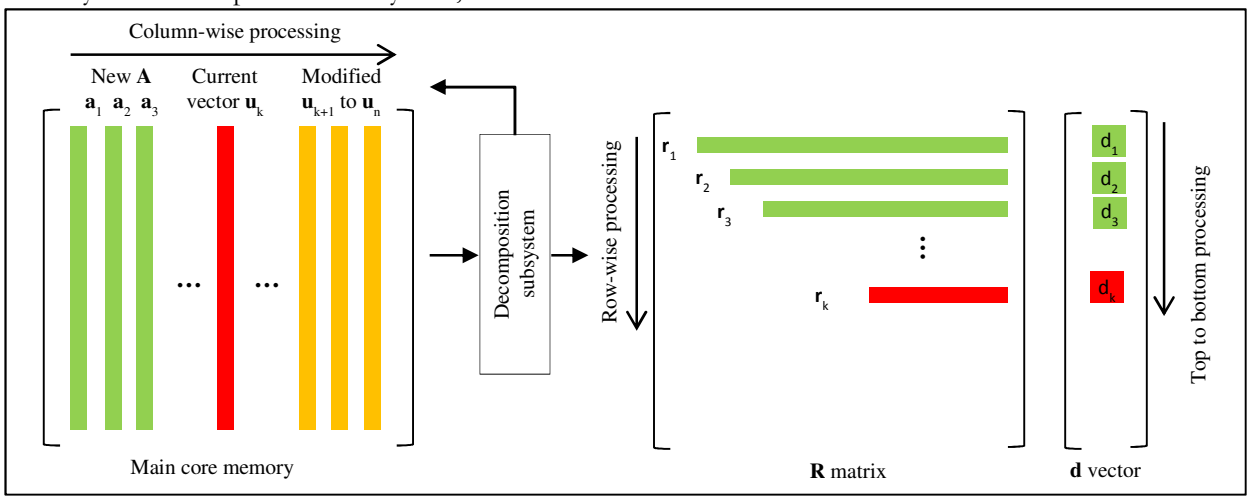

Figure 5 Memory organization of the QR decomposition subsystem

synthesized designs were run on two separate devices: the Stratix V and Arria V FPGAs. In the process, BDTI evaluated the Altera design flow and the performance of the two design examples.

Since the boards used for this evaluation lacked the ability to generate the stimuli for the example designs, each of the two designs contains a stimulus generation block that is implemented using DSP Builder Advanced Blockset blocks and compiled with the application design under test (DUT). In order to minimize the impact of the stimulus block on DUT performance and FPGA resource usage, matrix **A** is generated on-the-fly by the stimulus block from a much smaller set of random data. This small set of data is generated by MATLAB m-file scripts, loaded into the stimulus block memory and compiled with the design. These MATLAB scripts use the same algorithm as the stimulus block in the DUT to generate the double-precision floating-point format reference data for the solution vector **x** to measure the error performance of the Simulink models and the designs running on the FPGAs. In the Cholesky design this algorithm guarantees Hermitian positive definiteness for matrix **A**, whereas in the QR solver design it guarantees the creation of a well-conditioned **A** matrix.

Four configurations of vector, matrix, and channel sizes were implemented for the Cholesky solver design, three configurations on the Stratix V FPGA and two configurations on the Arria V device, with one common configuration between the two. For the QR solver design, four configurations were implemented for the Stratix V FPGA, and two of these were also implemented on the Arria V device. All configurations were evaluated for FPGA resource utilization, achievable clock rate, throughput, and functional correctness. FPGA design constraints, such as clock rate, device selection, and speed grade are specified in the Simulink environment.

Throughput and performance were evaluated at the Simulink model and the hardware platform levels. The Simulink model of both designs is instrumented to display both the forward processing and the total of forward and backward processing cycles, and the Quartus II software runs give the F_{max} achieved for each configuration. Each configuration is then downloaded to the hardware platform, its operating frequency set to F_{max}, and processing is started. The solution vectors **x** of each solver are captured for each configuration and compared against the corresponding MATLAB generated double-precision floating-point references.

A post-simulation script calculates the difference between the Simulink IEEE 754 single-precision floating-point output and the MATLAB generated double-precision floating-point reference vectors. Similarly, a script calculates the difference between the captured output of the hardware simulations and that of the MATLAB generated double-precision floating-point references.

Evaluation of the Tool Chain

Simulink is built upon and requires the MATLAB framework. In the Simulink environment, the evaluated designs use blocks from the Altera DSP Builder Advanced Blockset, which is a separate blockset from the standard DSP Builder library. The DSP Builder Advanced Blockset is geared towards block-based implementation of DSP algorithms and datapaths and uses a higher level of abstraction than the standard DSP Builder library, which comprises more general and elementary blocks. The DSP Builder Advanced Blockset library contains over 50 common trigonometric, arithmetic, and Boolean functions in addition to the more complex fast Fourier transform (FFT) and FIR filter building blocks. Notable additions in Quartus II software v12.0 are the low-latency square root, the nested *for* loop, and a customizable floating-point accumulator blocks. Elements from the standard Blockset and the DSP Builder Advanced Blockset cannot be mixed in a datapath structure at the same hierarchy level; only blocks from the DSP Builder Advanced Blockset support the floating-point compiler. Blocks from the standard Blockset are not optimized for floating-point processing. In addition, although importing of hand-coded HDL is available for the standard Blockset, it is not available in the DSP Builder Advanced Blockset since the tool cannot perform optimizations at the HDL level. In general, the block-based design-entry approach is well suited for DSP algorithms; however, due to the lack of constructs such as *case* or *switch* in the block library, a text-based approach is more intuitive for designs that are control-intensive and involve state machines.

Starting a simulation with DSP Builder Advanced Blockset compiles the Simulink model, generates HDL code and constraints for the Altera Quartus II software environment, builds a

test bench and script files for the ModelSim environment, and simulates the Simulink model. The time required to run the simulations for various configurations ranged from 3 minutes to 28 minutes depending on matrix size on a 3 GHz Intel Xeon W3550 PC. The Simulink simulation generates detailed resource utilization estimates without the need to run a Quartus II software compilation, thus helping the designer to quickly determine the device size needed. When using a hardware development kit, the user must supply pin-out assignments based on the board layout; the user can re-use the pin-out constraints in the provided *.qsf* file in the Quartus II software project folder for the design, or use the pin planner in Quartus II software to assign and manage pin-outs from scratch.

Experiments were performed on the model to evaluate the ease of algorithm exploration and the corresponding HDL generation. Input parameters, such as vector dot product size, matrix size, and data type were changed in the stimulus block and simulations run. In all cases, the correct RTL code was generated within minutes and simulation outputs matched the MATLAB reference.

All configurations were synthesized using the Quartus II design software, which may be launched directly from the Simulink environment. A designer may use the Quartus II software in *push-button* mode with either default or user-selected optimization parameters, or use the Design Space Explorer (DSE) tool. Available as part of the Quartus II software, DSE automatically runs multiple router passes using a different seed for each pass. The route with the best clock rate is saved. This is an automatic process requiring no user intervention but takes much longer than the push-button method. The Quartus II software push-button runs ranged from 1 hour to 6.5 hours depending on the design size.

The higher abstraction level employed by the DSP Builder Advanced Blockset design flow allows for a faster algorithmic space exploration and simulation cycles, thus reducing the overall time to reach a final optimized design. However, this advantage is not inherent to Simulink in the same way in which, for example, data-type propagation is inherent to Simulink. In order to exploit the design space exploration advantages offered by the block-based design approach over hand-written RTL, additional steps are required by the designer when creating the Simulink model. In particular, the model must be structured to enable a parameter-driven algorithmic space exploration. For the examples analyzed in this paper, the models are implemented to enable experimentation with different matrix sizes, vector sizes, and (in the case of the Cholesky solver) number of parallel channels. Once a model has been created incorporating this type of flexibility, performance and the resource usage estimates for various design configurations can be explored by varying these parameters. An understanding of hardware design is also required to achieve good throughput rates and resource utilization, as exemplified in the floating-point accumulator block discussed in Section 2 of this paper.

Training for the DSP Builder Advanced Blockset design flow entails a 4-hour class by Altera and approximately 10 hours of on-line tutorials and demos. In addition, BDTI spent approximately 90 hours exploring the tool and both models for hands-on experience. The time and effort required to get up to speed with the tool chain will depend on the skills and background of the designer. A seasoned engineer with both Simulink block-based design and FPGA hardware design experience will likely find the DSP Builder Advanced Blockset approach efficient and easy to use. For an FPGA designer with little or no knowledge of MATLAB and Simulink, designing at a higher level of abstraction may represent a new way of thinking and thus an initial challenge, entailing a significant learning curve. Once the methodology is mastered, the designer can achieve significantly faster design cycles than a HDL approach. One can focus on implementing the algorithm and not worry about hardware design details such as pipelining. Design and verification time is significantly reduced since the majority of functional simulation and verification is done in the Simulink environment. The RTL output from the Simulink compilation may be run in the ModelSim software for a full functional simulation.

The learning curve may be less steep for an engineer with system-level design background who has little or no skills in hardware design. Although the tool chain integrates hardware compilation, synthesis, routing, and automatic script generation within the Simulink environment and abstracts away many complex design concepts such as data pipelining and signal vectorizing, some knowledge of hardware design is still needed to complete an implementation.

4. Performance Results

This section presents the results of BDTI's independent evaluation of the Altera Cholesky and QR solvers floating-point implementation examples.

All designs used Altera's DSP Builder Advanced Blockset v12.0, with MathWorks release R2011b, and built with Quartus II design software v12.0 SP1. RTL simulations were done using ModelSim 10.1. The designs were built for two Altera 28-nm FPGAs: the high-end medium-size Stratix V 5SGSMD5K2F40C2 device, and the mid-range Arria V 5AGTFD7K3F40I3N device. The Stratix V FPGA used in this analysis, features 345.2K ALUTs, 1,590 27×27-bit variable-precision multipliers, and 2,014 M20K memory blocks. The Arria V FPGA features 380.4K ALUTs, 1,156 27×27-bit variable-precision multipliers, and 2,414 M10K memory blocks. The hardware platforms used for RTL evaluation were the DSP Development Kit, Stratix V Edition, and the Arria V FPGA Development Kit. The ModelSim software was used for one configuration to assess ease of use of the tool from the Simulink environment.

Combined, a total of eleven cases were simulated and built for both designs on the two devices. Resource utilization, performance, and accuracy results were recorded for each case. Table 1 lists the resource utilization and clock speed achieved for the Cholesky and QR solvers for each configuration. The Cholesky solver design provides a maximum matrix size parameter. At runtime, matrix sizes smaller than the maximum design size may be used. For the resource utilization results presented in Table 1, each configuration was synthesized with the maximum matrix size parameter equal to the matrix size under evaluation in order to obtain the actual resources consumed by the reported matrix size. The resources used by the stimuli blocks were not included in the totals. It is worthwhile to note that none of the configurations evaluated in this paper used the FPGAs to capacity. To achieve the best F_{max} in a reasonable amount of synthesis and place-and-route time in the Quartus II software, identical preset optimization parameters were used for each design to improve speed. We chose the 6/90×90/45 configuration for the Cholesky design example to run the Quartus II software's Design Space Explorer (DSE) to assess the speed improvement and time taken for synthesis as compared to the push-button mode. In this case, a speed improvement of 12.5% was achieved, however the Quartus II software run time increased from 2 hours to 7.5 hours to synthesize the design.

The FPGA resource utilization is consistent with expectations for the designs under evaluation. Memory use is dominated by matrix

Example	Device	Configuration (Channel Size/ Matrix Size/ Vector Size)	ALUT (K) (Used / % of Total)	Registers (K) (Used / % of Total)	DSP Blocks (Variable-Precision 27×27 Multipliers used/ % of Total)	M20K (Stratix) /M10K (Arria) (Blocks used / % of Total)	F_{max}, (MHz) P: Push-button used D: DSE used
Cholesky	Stratix V	1 / 360×360 / 90	198 / 57%	339 / 49%	391 / 25%	1411 / 70%	189 (P)
		20 / 60×60 / 60	135 / 39%	235 / 34%	268 / 17%	955 / 48%	234 (P)
		64 / 30×30 / 30	74 / 22%	124 / 18%	146 / 9%	793 / 39%	288 (P)
	Arria V	6 / 90×90 / 45	104 / 27%	179 / 24%	214 / 19%	1094 / 45%	176 (P) 198 (D)
		64 / 30×30 / 30	73 / 19%	121 / 16%	154 / 13%	1694 / 70%	185 (P)
QR	Stratix V	1 / 400×400 / 100	184 / 53%	377 / 55%	428 / 27%	1566 / 78%	203 (P)
		1 / 200×100 / 100	180 / 52%	375 / 54%	428 / 27%	504 / 25%	207 (P)
		1 / 200×100 / 50	96 / 28%	201 / 29%	228 / 14%	281 / 14%	260 (P)
		1 / 100×50 / 50	95 / 28%	198 / 29%	227 / 14%	230 / 12%	259 (P)
	Arria V	1 / 200×100 / 50	97 / 25%	202 / 27%	238 / 21%	372 / 15%	171 (P)
		1 / 100×50 / 50	95 / 25%	200 / 26%	237 / 21%	245 / 10%	170 (P)

Table 1 Resource utilization and clock speed

Example	Device	Configuration (Channel Size/ Matrix Size/ Vector Size)	Throughput Reported by Simulink (kMatrices/sec)	Overall Latency (µsec) @F_{max} (MHz)	GFLOPS (Real Data Type)
Cholesky	Stratix V	1 / 360×360 / 90	1.43	1112 @ 189	91
		20 / 60×60 / 60	118.35	330 @ 234	39
		64 / 30×30 / 30	544.28	222 @ 288	26
	Arria V	6 / 90×90 / 45	31.31 35.22	347 @ 176 308 @ 198	34 38
		64 / 30×30 / 30	349.62	344 @ 185	16
QR	Stratix V	1 / 400×400 / 100	0.315	3970 @ 203	162
		1 / 200×100 / 100	8.76	167.0 @ 207	141
		1 / 200×100 / 50	6.17	204.5 @ 260	99
		1 / 100×50 / 50	32.82	43.3 @ 259	66
	Arria V	1 / 200×100 / 50	4.05	311 @ 171	65
		1 / 100×50 / 50	21.54	66 @ 170	44

Table 2 Performance Results

storage and is proportional to the matrix size and the number of channels for a multi-channel design. The DSP blocks usage increases linearly with the vector size. The vector multiplier requires 4 variable-precision DSP blocks per 27-bit × 27-bit complex valued floating-point multiplication. Given a vector size of 60 complex floating-point values, 240 DSP blocks are required for the vector dot product engine.

Table 2 shows the performance of the Cholesky and the QR solvers for all configurations. The performance for each case is given at the F_{max} reported in Table 1. The throughput is calculated by dividing F_{max} by the cycles consumed by the solver forward subsystem execution. Since the backward substitution subsystem executes in parallel and with lower latency than the forward subsystem, the overall throughput is not affected by the former. For the multi-channel Cholesky solver throughput, this result is multiplied by the number of channels that are processed in parallel (Channel Size parameter in Table 2). The overall latency for each case is calculated by dividing the total cycles taken by the execution of the forward and backward subsystems by F_{max}. The choice of vector size relative to matrix size is a compromise and is application dependent. If the vector size is much smaller than the matrix size, the design will be resource efficient at the expense of latency, as shown for the QR solver 200×100 matrix size configurations with different vector sizes.

The multi-channel Cholesky design improves upon the single-channel design that was analyzed in the previous paper by BDTI. In the single-channel implementation, latencies, such as those found in the floating-point accumulator were partially hidden by rearranging the processing order in the algorithm. As reported in reference [1], the efficiency of the single-channel implementation depended mostly on the matrix and vector sizes. Looking at the throughput column of Table 2, the multi-channel implementation has significant benefits in processing efficiency, particularly for smaller size matrices and vector sizes. Multi-channel processing improves throughput by completely hiding the implementation latencies described in Section 2 of this paper. For a given matrix and vector sizes, a multi-channel implementation will deliver a higher peak throughput than its single-channel counterpart.

The last column in the table shows the number of real-data floating-point operations per second in units of 10^9 (GFLOPS) for each of the configurations. The number of operations required by each solver depends on the decomposition algorithm used. The reported numbers were derived from the actual implementation of the Cholesky solver and QR solver algorithms in floating-point complex-data format on the two FPGAs used for this evaluation. For the Cholesky solver, the number of real-data floating-point operations is approximated to the second order term $4n^3/3$ +

Example	Device	Configuration (Reported Channel Number / Matrix Size/ Vector Size)	MathWorks Simulink IEEE 754 Floating-Point Single-Precision Error (Frobenius Norm /Maximum Normalized Error)	Altera's DSP Builder Synthesized RTL Floating-Point Single-Precision Error (With Fused Datapath Methodology) (Frobenius Norm / Maximum Normalized Error)
Cholesky	Stratix V	1 / 360×360 / 90	2.11e-6 / 1.02e-4	1.16e-6 / 8.58e-5
		7 / 60×60 / 60	4.24e-7 / 8.59e-6	1.82e-7 / 2.62e-6
		53 / 30×30 / 30	7.48e-8 / 2.08e-6	3.84e-8 / 1.15e-6
	Arria V	3 / 90×90 / 45	4.08e-7 / 9.72e-6	1.99e-7 / 5.52e-6
		63 / 30×30 / 30	8.93e-8 / 2.38e-6	5.91e-8 / 1.24e-6
QR	Stratix V	1 / 400×400 / 100	4.53e-6 / 1.45e-4	5.15e-6 / 1.03e-4
		1 / 200×100 / 100	1.24e-6 / 1.13e-5	9.97e-7 / 8.15e-6
		1 / 200×100 / 50	8.38e-7 / 6.70e-6	8.97e-7 / 4.15e-6
		1 / 100×50 / 50	9.13e-7 / 4.68e-6	6.96e-7 / 4.94e-6
	Arria V	1 / 200×100 / 50	9.27e-7 / 2.33e-5	9.31e-7 / 9.95e-6
		1 / 100×50 / 50	9.13e-7 / 4.68e-6	6.96e-7 / 4.94e-6

Table 3 Error performance of the Simulink model and the synthesized RTL compared to the MATLAB double-precision floating-point reference

$12n^2$, whereas for the QR solver $8mn^2 + 6.5n^2 + mn$ is used.

Table 3 shows the error performance of the Cholesky and QR solvers for both the Simulink simulation and the design implementation running on the hardware development boards using single-precision floating-point operations. The error is calculated by comparing the output of each of the Simulink and the hardware platform simulations with the double-precision floating-point reference for the solution vectors **x** generated by MATLAB. For the multi-channel Cholesky solver cases, only the error performance of a single randomly chosen channel is reported for brevity. Although the error performance is input data dependent, on average, the RTL implementation benefits from the fused datapath methodology and achieves a statistically equal or higher precision than the standard IEEE 754 single-precision implementation as demonstrated by comparing the Frobenius Norm in columns (4) and (5) of Table 3. We use the Frobenius Norm to get a measure of the overall error magnitude in the resultant vector and is given by:

$$\|E\|_F = \sqrt{\sum_{i=0}^{N} |e_i|^2}$$

Where N is the size of the vector, **e** is the difference vector between observed **x** and its MATLAB generated golden reference, and i is the index of the elements in vector **e**. The maximum normalized error is given by:

$$\max_i \left(\left| (x_{i\_obs} - x_{i\_ref}) / x_{i\_ref} \right| \right)$$

5. Conclusions

In this paper, we evaluated a new approach to implementation of floating-point DSP algorithms on FPGAs using Altera's DSP Builder Advanced Blockset design flow. This design flow incorporates the Altera DSP Builder Advanced Blockset, Altera's Quartus II software tool chain, and ModelSim simulator, as well as MATLAB and Simulink from MathWorks. This approach allows the designer to work at the algorithmic behavioral level in the Simulink environment. The tool chain combines and integrates the algorithm modeling and simulation, RTL generation, synthesis, place and route, and design verification stages within the Simulink environment. This integration enables quick development and rapid design space exploration both at the algorithmic level and at the FPGA level, and ultimately reduces overall design effort. Once the algorithm is modeled and debugged at a high level, the design can be synthesized, and targeted to an Altera FPGA.

For the purpose of this evaluation, the design examples were single-precision complex-data IEEE 754 floating-point Cholesky and QR solvers modeled in Simulink using the Altera DSP Builder Advanced Blockset. The largest design example we evaluated was a QR solver for a complex-valued floating-point matrix of size 400×400 and a vector size of 100. Running at 203 MHz this example processes 162 GFLOPS. The reported GFLOPS values in Table 2 are for the actual

implementation of the Cholesky solver and the QR solver algorithms in floating-point complex-data format on the two FPGAs. For a valid comparison with other competing platforms, the same algorithms should be implemented on these platforms and their performance measured. All reported performance results were achieved using the Altera DSP Builder Advanced Blockset tool flow with no hand optimization or floor planning. Starting from a high-level block-based design in Simulink, the tool chain automatically pipelined, generated the RTL code and synthesized the design to achieve usable speeds and resource utilization. The Altera floating-point design flow simplifies the process of implementing complex floating-point DSP algorithms on an FPGA by streamlining the tools under a single platform. With its fused datapath methodology, complex floating-point datapaths are implemented with higher performance and efficiency than previously possible.

The new approach analyzed in this paper also entails a significant learning curve for using the DSP Builder Advanced Blockset. This is especially true for a designer not familiar with MATLAB and Simulink. The block-based design-entry approach may present an initial challenge for a traditional hardware designer. In addition, in order to exploit the advantages offered by the block-based design approach over hand-written RTL, additional steps are required by the designer when creating the Simulink model. For example, to enable experimentation with different matrix and vector sizes, as is done in the two design examples in this paper, the Simulink model was structured to incorporate a parameter-driven design to explore the various design configurations.

Currently, designers using the DSP Builder Advanced Blockset must limit themselves to the elements provided by the blockset in order to achieve optimized performance. Elements from the standard DSP Builder Blockset are not optimized with the floating-point compiler nor can they be mixed with the Advanced Blockset at the same hierarchy level. Hand-coded HDL blocks may only be imported into the Standard Blockset. Additionally, the DSP Builder Advanced Blockset is geared towards DSP implementations and may have limited use for designs involving heavy control and state machines.

The next version of the DSP Builder Advanced Blockset, expected to be released by the end of 2012, will include floating-point extensions. The designer will no longer be limited to the two standard IEEE 754 single-precision and double-precision formats, but will have the choice of a total of seven different precisions ranging from 16 to 64 bits (exponent plus mantissa). Using the new *Enhanced Precision Support* block in the DSP Builder Advanced Blockset, the designer may choose the data-width that best fits their application.

6. References

[1] Berkeley Design Technology, Inc., 2011. "An Independent Analysis of Altera's FPGA Floating-point DSP Design Flow". Available for download at http://www.altera.com/literature/wp/wp-01166-bdti-altera-floating-point-dsp.pdf.

[2] S.S. Demirsoy, M. Langhammer, 2009. "Fused datapath floating point implementation of Cholesky decomposition." FPGA '09, February, 2009.