An Independent Analysis
of
Altera's FPGA Floating-point DSP Design Flow

By the staff of

Berkeley Design Technology, Inc.

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 flow using Altera's DSP Builder Advanced Blockset and the MathWorks' MATLAB and Simulink tools. Altera hopes 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.

Contents

1. Introduction

A Floating-point Design Example

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 a long time, linear algebra (and specifically solving for systems with a large set 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 very computationally demanding, these techniques can suffer from numeric instability if sufficiently high 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 order to evaluate the effectiveness of Altera's approach, BDTI focused on a large matrix inversion problem implemented using the Cholesky matrix decomposition algorithm combined with forward and backward substitution. These three processes combined constitute a Cholesky solver which finds the inverse of a Hermitian positive definite matrix to solve for the vector **x** in a simultaneous set of linear equations of the form **Ax** = **B**.

The Cholesky solver is an important algorithm due to its increasing use in military radar applications such as Space-Time Adaptive Processing (or STAP). In addition, the Cholesky decomposition itself, which is the core of the solver, is used in many estimation and optimization problems where covariance matrices are found. The Cholesky decomposition is a very computationally demanding algorithm and requires high data precision, so floating-point math is necessary. In addition to being an important problem in many applications, matrix inversion can serve as an example of a wide range of floating-point DSP algorithms. For example, the Cholesky decomposition uses vector dot products and nested loops that are found in a range of digital signal processing applications involving linear algebra and finite impulse response (FIR) filters.

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 **z*****Mz** > 0 for all non-zero complex vectors **z**. The quantity **z*Mz** is always real because **M** is a Hermitian matrix for the purposes of this paper.

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

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

Using the Cholesky solver, as described in Section 4, an Altera Stratix IV FPGA is capable of performing 3,204 matrix inversions per second on matrices of size 240×240, running at a clock speed of 200 MHz and at an accuracy that exceeds that of the single-precision IEEE Standard for Floating-Point Arithmetic (IEEE 754) number representation.

The Cholesky solver example evaluated in this paper is available at www.altera.com/floatingpoint and will be made available by Altera as a design example packaged with the DSP Builder Advanced Blockset tool chain starting with tool version 11.1.

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 that 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

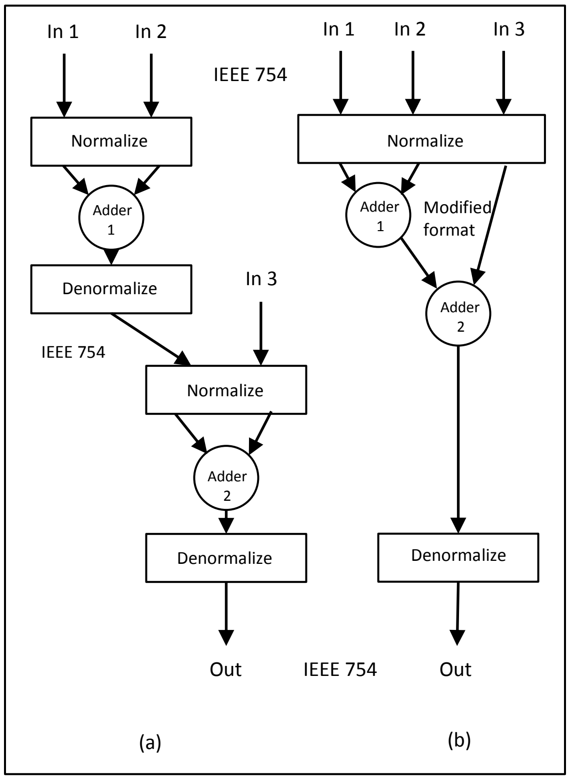

Figure 1 (a) Traditional floating-point implementation (b) Fused datapath implementation

operators. It does this by analyzing the bit growth in the datapath and appropriately choosing the optimum input normalization to allocate enough precision through the datapath in order to eliminate as many normalization and denormalization steps as possible. The IEEE 754 format is only used at datapaths boundaries; within datapaths, larger mantissa widths are used to increase the dynamic range and reduce the need for denormalization and normalization functions between successive operators. Normalization and denormalization functions use barrel shifters of up to 48-bit range for a single precision floating-point number. This consumes significant amount of logic and routing resources and is the main reason why floating-point implementations on FPGAs are not efficient. The *fused datapath* methodology eliminates a significant number of these barrel shifters. Multiplications involving the larger precision mantissas use Altera's 36×36 multiplier mode in Stratix IV and Arria II 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 need to denormalize the output of the first adder and to normalize 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 [1]. As a result of the wider internal data representation, on the average, 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 RTL tool chain, ModelSim simulator, as well as MathWorks' MATLAB and Simulink tools. 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 in the DSP Builder Advanced Blockset, enabling a designer to perform quick algorithmic design space exploration.

In the evaluation described in this white paper, BDTI used the Altera DSP Builder Advanced Blockset tool flow to validate a complex data-type floating-point Cholesky solver design example and evaluate the efficiency and performance of Altera's floating-point design flow. Section 2 of the paper describes the implementation of the Cholesky solver. 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 Stratix IV EP4SE360H29C2 device and the mid-range Arria II EP2AGX125DF25I3 device. Finally, Section 5 presents BDTI's conclusions.

2. Implementation

Background

Sets of linear equations of the form **Ax** = **b** arise in many applications. Whether it is an optimization problem involving linear least squares, a Kalman filter for a prediction problem, or MIMO channel estimation, the problem remains one of finding a numerical solution for a set of linear equations of the form **Ax** = **b**. When matrix **A** is symmetric and positive definite, which is true for the covariance matrices used in these 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 other methods such as the **LU** or **QR** decompositions. Since all 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.

THE ALGORITHM

The recursive Cholesky algorithm 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***

```
$l_{11} = \sqrt{a_{11}}$    (1)
For i = 2 to n,
    $l_{i1} = {a_{i1}}/{l_{11}}$    (2)
    For j = 2 to (i-1),
        $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} \times l_{jk}^*)/l_{jj}$    (3)
    end
    $l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1}(l_{ik} \times l_{ik}^*)}$    (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**,

```
$y_1 = b_1/l_{11}$    (5)
For i = 2 to n,
    $y_i = (b_i - \sum_{k=1}^{i-1} y_k \times l_{ik})/l_{ii}$    (6)
end
```

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

```
$x_n = y_n/l_{nn}^*$    (7)
For i = n-1 to 1,
    $x_i = (y_i - \sum_{k=i+1}^{n} x_k \times l_{ik}^*)/l_{ii}^*$    (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**.

Architectural Overview

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 Algorithm*. The second subsystem executes the backward substitution—step 3 in the sidebar. Since the input matrix is Hermitian and the decomposition generates complex conjugate transposed triangular matrices, memory utilization is optimized by loading only half of the input matrix **A** and generating only the lower triangular matrix, which overwrites **A** as the latter is being consumed. 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 pipeline stages go into the input stages of the backward substitution, as shown in Figure 2.

In mathematical terms, the forward substitution of equation (6) can be considered a subset of the decomposition equation (3) except for the conjugation of l_{jk}^*, and thus in the implementation, they are combined in a single process by appending the transpose of vector **b** to the last row of matrix **A** without incurring any significant latency in processing.

The core of the decomposition is the complex vector dot product engine (also referred to as the vector multiplier) in equations (3) and (4). For the Stratix IV SE360 device, a vector size (VS) of up to 60 complex elements is possible, limited by the number of available DSP elements in the device, whereas for the Arria II GX125 device, a vector size of up to 30 elements may be 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 dual-port memory used internally. For implementation reasons, the memory of a matrix of a given size is partitioned into *ceil(N/VS)* banks, where *ceil()* is the ceiling function and *N* is the size of the matrix. The largest matrix used in this evaluation is a Hermitian matrix of size 240×240. The size of the matrix is limited by the available memory in the device.

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. 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 three-level nested *for*

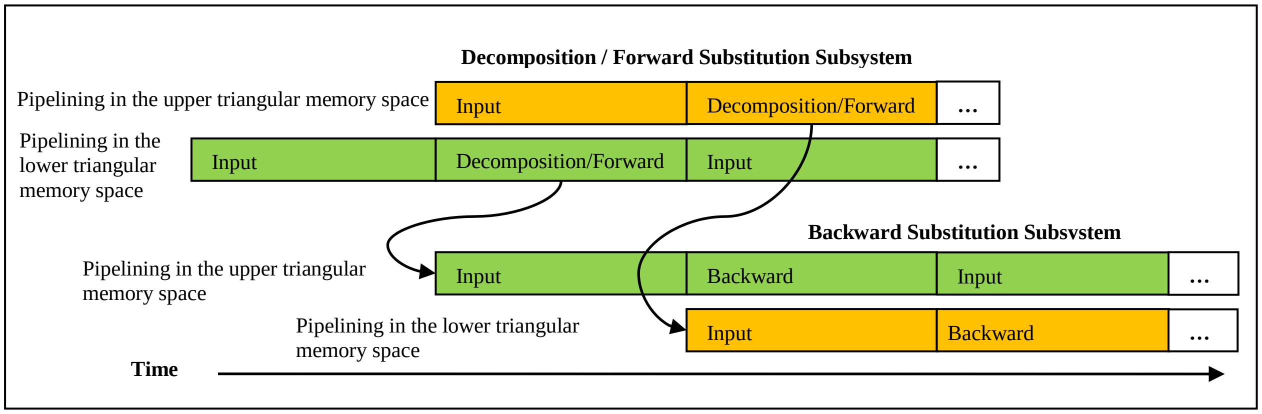
Decomposition / Forward Substitution Subsystem
Pipelining in the upper triangular memory space
Input
Decomposition/Forward
…
Pipelining in the lower triangular memory space
Input
Decomposition/Forward
Input
…
Backward Substitution Subsystem
Pipelining in the upper triangular memory space
Input
Backward
Input
…
Pipelining in the lower triangular memory space
Input
Backward
…
Time

Figure 2 Process pipelining and memory reuse

loop. The "For Loop" block in the DSP Builder Advanced Blockset is perfect for implementing iterative loops. This block abstracts away the intricate control signals that would have been necessary in a hand-written RTL implementation, allowing the designer to focus on the algorithm itself. The outermost loop implements the column-wise processing for *j = 1 to N;* the middle loop implements the bank-wise processing for *bank = 1 to ceil(N/VS)*; and the innermost loop processes the row elements for *i = 1 to N*. The processing sequence is shown in Figure 3(a).

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. 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. The summation of the bank outputs is performed in a single accumulator loop using the floating-point adder block from the DSP Builder Advanced Blockset. This feedback loop has a latency of 13 cycles. In order to avoid this significant stall while waiting for the accumulator to finish, the middle loop in the three-level nested loop is the "*for Banks*" loop rather than the "*for Rows*" loop as one would traditionally have in a software implementation. By performing this swap, the floating-point accumulator latency is hidden and hardware utilization improved. Currently, the DSP Builder Advanced Blockset does not automatically add loop delay elements to address this type of latency. This is because adding a delay element may unintentionally change the functionality of a digital signal processing feedback loop. Hence, it is the designer's responsibility to specify this value. However, DSP Builder will generate an error message indicating to the designer a deficiency in the delay value. The designer may experiment to find the exact value needed. Figure 3(b) shows the computation of the diagonal element e_{ij}. The dot product that generates this element spans over two full-length banks of vector size plus a third partial bank. Since the *for Banks* loop is the middle loop, the full value of the e_{ij} element is only available after all the partial sum-of-products at j=VS and at j=2*VS boundaries for all the rows below it are processed. These partial sum-of-products are stored temporarily in an internal FIFO.

The choice of the vector size affects the hardware efficiency and system latency. If vector size is large relative to the matrix size, many terms in the dot product are not used until column indices greater than vector size are reached thus reducing hardware efficiency. Refer to Section 4 for a more detailed analysis of the implications of matrix-to-vector size choices on latency and hardware utilization.

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 for the dot product is not employed. 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.

3. Design Flow and Tool Chain

For this evaluation, Altera provided BDTI with an implementation of the Cholesky solver created using the DSP Builder Advanced Blockset. BDTI engineers installed the Altera and MATLAB tools necessary for the evaluation, and took a short training class to familiarize themselves with the design flow. BDTI engineers then examined the Altera design, simulated it, synthesized it, and ran ModelSim RTL simulations, all under the Simulink environment. In the process, BDTI evaluated the Altera design flow and the performance of the Cholesky solver example. Installation of the tool chain was straightforward and painless.

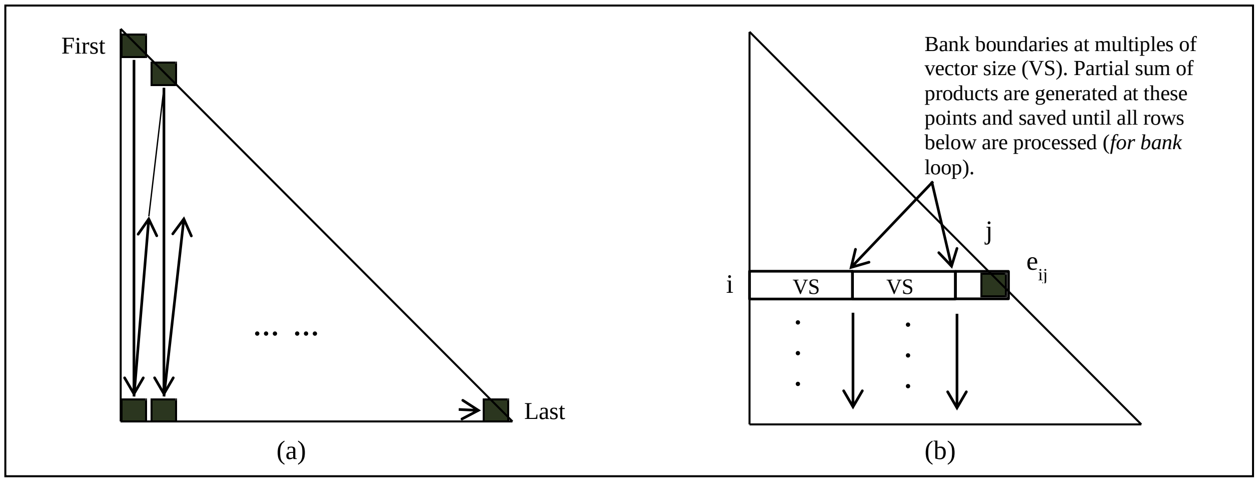
First
Last
... ...
(a)
Bank boundaries at multiples of vector size (VS). Partial sum of products are generated at these points and saved until all rows below are processed (*for bank* loop).
j
i
VS
VS
e_{ij}
(b)

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

Simulink is built upon and requires the MATLAB framework. The input stimuli for the Cholesky model are generated via a MATLAB m-file script. The input matrix is synthetically generated to guarantee Hermitian positive definiteness. The two main input parameters are the matrix size and the vector size. The vector size is a design parameter that determines the size of the vector dot product engine and the maximum allowable matrix size in the synthesized design. In this implementation, the maximum matrix size is set to be four times the vector size. The input parameter matrix size is a user variable and may be of any size equal to or smaller than the maximum-allowed matrix size.

The MATLAB m-file script saves the synthetically generated **L** and **x**, and the calculated vector **y** in double-precision floating-point format as reference to measure the error performance of the Simulink model and the synthesized RTL design.

In the Simulink environment, the Cholesky solver design uses 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 design of DSP algorithms and datapaths and uses a higher level of abstraction than the standard DSP Builder library, which uses more general and elemental functional blocks. The 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. Elements from the DSP Builder 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 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 a text-based approach is still more intuitive for designs that are control based and involve state machines.

Starting a simulation in Simulink compiles the model, generates HDL code and constraints for the Altera Quartus II environment, builds a test bench and script files for the ModelSim environment, and runs the Simulink model simulation. The time required to run the simulation for a single Cholesky solver ranged from 6 minutes to less than 1 minute depending on the matrix size. The Simulink simulation generates detailed resource utilization estimates without the need to run a Quartus II compile, thus helping the designer to quickly determine the device size needed for various algorithmic modifications.

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 saved MATLAB reference.

The higher abstraction level of development allows faster algorithm space exploration, simulation, and time optimized RTL generation. However, the flexibility afforded by this new approach requires a lot of forethought into the structure of the model before starting the high-level block-based design. There is a design methodology that needs to be understood and followed to support the flexibility of exploring the design space with input parameters such as the vector size and the matrix size for this example. An understanding of some hardware design is still required to achieve good throughput rates and resource utilization as exemplified in the floating-point accumulator section of this model.

Seven configurations of vector and matrix sizes were evaluated for FPGA resource utilization, maximum 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.

All configurations were synthesized using the Quartus II development software, which may be launched directly from the Simulink environment. The results reported by the Quartus II software for a single routing run achieve clock rates of 154 MHz to 203 MHz depending on the configuration. The Design Space Explorer tool was used to achieve higher clock rates. Available as part of the Quartus II software, Design Space Explorer 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 and took 4 to 6 hours to converge on an optimized implementation for the Cholesky solver design. The improvement achieved ranged from 8% to 23% depending on the matrix size and the vector size used. For example, a matrix size of 60×60 with a vector size of 60 achieved 202 MHz using Design Space Explorer compared to a single routing fit of 154 MHz.

The FPGA resource utilization is consistent with expectations for such a design: memory use is proportional to the square of the matrix size, and multiplier usage increases linearly with the vector size. Due to the memory granularity in an FPGA, it is generally difficult to accurately determine memory requirements. For example, a synthesis tool may choose to use memory blocks to accommodate for signal delays. However, in this example, the majority of the memory is consumed by matrix data storage. The multiplier utilization is more predicable: the vector multiplier requires sixteen 18×18 DSP elements per complex valued floating-point multiplication. Given a vector size of 60 complex floating-point values, 960 18×18 DSP elements are required for the vector dot product engine. Refer to Section 4 for a breakdown of the resource requirements for each of the seven configurations.

Throughput and performance were evaluated at the RTL level using a modified version of the design. A counter that is enabled by the vector multiplier active signal was added to the design in the Simulink schematic capture environment to determine the processing efficiency. The actual processing time of the Cholesky solver was determined by measurement in the ModelSim simulation environment. For all configurations, the Simulink processing time calculations matched the ModelSim measurements.

The performance results listed in Section 4 were achieved with the Design Space Explorer tool; no hand optimization or floor planning was performed. Upon a closer analysis, BDTI found that the worst case delays in the design are due to routing rather than chains of logic, which indicates that the tool has efficiently pipelined the datapath. The tool chain achieved usable speeds and resource utilization without any low level design modifications or floor planning.

A post-simulation script calculates the difference between the Simulink IEEE 754 single-precision floating-point output and the synthetically generated MATLAB double-precision floating-point reference. Similarly, the script calculates the difference between the output of the ModelSim simulation of the single-precision floating-point synthesized RTL created using DSP Builder Advanced Blockset and that of the synthetically generated MATLAB double-precision floating-point reference. Refer to Section 4 for the error performance results.

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 close to 90 hours exploring the Cholesky solver model and making modifications for a 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 an HDL approach. One can focus on implementing the algorithm and not worry about hardware design details such as pipelining. Simulation time is significantly reduced as a full functional simulation in ModelSim can be done once the majority of functional verification has been completed first in the Simulink environment.

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 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 solver floating-point implementation example.

Device	Configuration (Matrix Size/ Vector Size	Logic Elements Used (LEs / % of Total)	DSP Blocks Used (18x18 Multipliers / % of Total)	Memory (Size / % Total)			Clock Rate (F_{max}, MHz)
				M144K (Blocks)	M9K (Blocks)	MLAB (64-bit Blocks)	
Stratix IV EP4SE360H29C2	240×240 / 60	162K / 57%	1,014 / 98%	0 / 0%	899 / 72%	13.4K / 9%	218
	180×180 / 60	133K / 47%	1,014 / 98%	0 / 0%	771 / 60%	4.7K / 3%	224
	120×120 / 60	131K / 46%	1,014 / 98%	0 / 0%	276 / 60%	4.6K / 3%	225
	60×60 / 60	143K / 50%	1,014 / 98%	0 / 0%	66 / 5%	8.1K / 6%	202
Arria II EP2AGX125 DF25I3	120×120 / 30	63K / 64%	534 / 93%	N/A	440 / 60%	1.3K / 3%	214
	60×60 / 30	64K / 65%	534 / 93%	N/A	284 / 39%	1.2K / 3%	228
	30×30 / 30	67K / 68%	534 / 93%	N/A	226 / 31%	3.0K / 6%	207

Table 1 Resource utilization and clock speed

All designs used Altera's DSP Builder Advanced Blockset 11.0, implemented using MathWorks' MATLAB 7.10, Simulink 7.5, and built with Quartus II design software version 11.0. The RTL simulations were done using Altera ModelSim 6.6d. The designs were built for two Altera 40-nm FPGAs: the high-end Stratix IV EP4SE360H29C2 device with -2 speed grade, and the mid-range Arria II EP2AGX125DF25I3 with -3 speed grade. (These are the fastest grades for each device.) In all cases, Design Space Explorer was employed to optimize the clock rate (F_{max}).

A total of seven cases were simulated and built: four matrix sizes with a vector size of 60, and three matrix sizes with a vector size of 30. Resource utilization, performance, and accuracy results were recorded for each case. Table 1 lists the resource utilization and clock speed achieved for each configuration.

The Cholesky solver design provides a 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 get the actual resources consumed by the tested matrix size.

It should be noted that a very simple design consisting of a single or a few floating-point operators may run much faster than the clock rates achieved in the Cholesky solver example shown in Table 1. This, however, is not a particularly meaningful comparison because routing congestion tends to severely limit performance as floating-point designs get more complex.

Table 2 shows the performance of the Cholesky solver implementation for the seven configurations. The vector multiplier utilization percentages were calculated by dividing the vector multiplier active cycles by the total cycles consumed by the Cholesky solver, both of which were reported by Simulink. Note that this utilization does not take into account the non-valid terms in the dot product engine due to the triangular

Configuration (Matrix Size/ Vector Size	Throughput Reported by Simulink (Matrices/sec)	Throughput Reported by ModelSim (Matrices/sec)	Vector Multiplier Utilization	
			Reported by Simulink	Reported by ModelSim
240×240 / 60	3,204	3,204	88%	88%
180×180 / 60	6,113	6,113	78%	78%
120×120 / 60	12,680	12,680	58%	58%
60×60 / 60	28,998	28,998	27%	27%
120×120 / 30	9,921	9,921	69%	69%
60×60 / 30	27,886	27,886	33%	33%
30×30 / 30	59,665	59,665	14%	14%

Table 2 Performance of the Cholesky solver

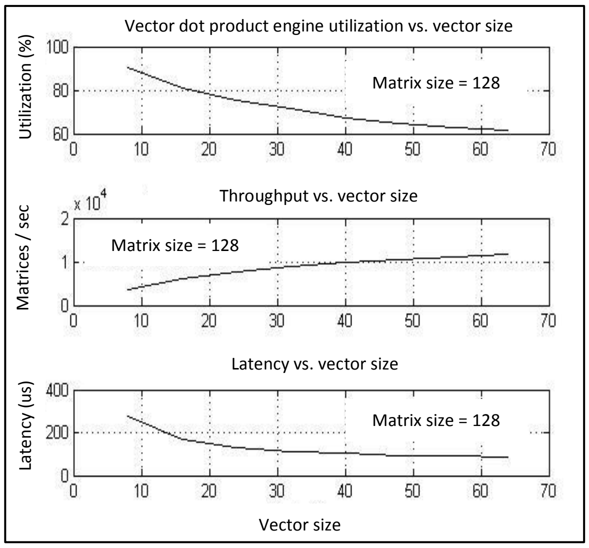

Figure 4 Effect of vector size

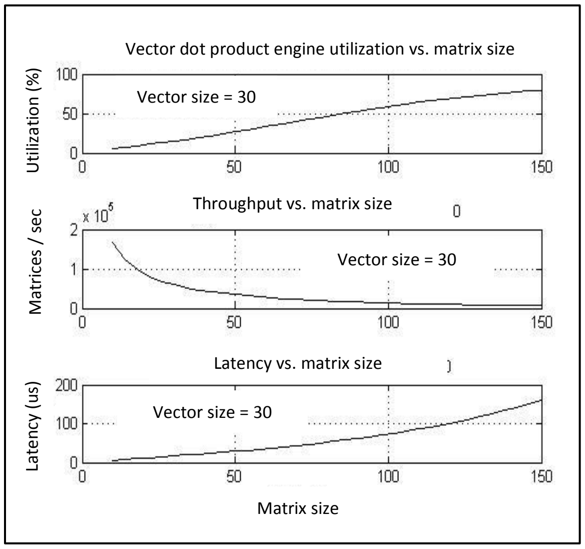

Figure 5 Effect of matrix size

nature of the input matrix. It indicates the utilization of the dot product engine as a single working unit. The performance is given for a 200-MHz clock rate. As the table shows, the Simulink and ModelSim calculations match. The throughput is calculated by dividing 200 MHz by the cycles consumed per Cholesky solver execution. Since the backward substitution subsystem executes in parallel and with lower latency than the Cholesky decomposition, the overall throughput is not affected by the backward substitution subsystem. The latency for each case is the inverse of the throughput. The dot product engine in the Cholesky decomposition and forward substitution is the core of the algorithm and its utilization percentage in Table 2 is a good measure of the efficiency of the implementation. Note how the efficiency goes down as the ratio of matrix size to vector size approaches 1:1. Hypothetically speaking, for smaller covariance matrices, the decomposition of multiple matrices can be interleaved and computed by the same datapath. This time division multiplexing increases the efficiency as long as the numerical value of the number of matrices times the matrix size is larger than the latency of a single decomposition.

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. On the other hand, if the vector size is close or equal to the matrix size then the latency will be shorter at the expense of resource utilization. In general, a vector size equal to a quarter of the matrix size is a good compromise. Figure 4 and Figure 5 show the effect of various vector sizes and matrix sizes on resource utilization, throughput, and latency.

Table 3 shows the error performance of the Cholesky solver for both the Simulink simulation and the RTL implementation using single-precision floating-point operations. The error is calculated by comparing the output of each of the Simulink and

Matrix Size /Vector Size	Function	MathWorks Simulink IEEE 754 Floating Point Single Precision (Norm / Max Error)	Altera's DSP Builder Synthesized RTL Floating Point Single Precision (With Fused Datapath Methodology) (Norm / Max Error)
240×240 / 60	Cholesky decomposition	3.53e-5 / 2.84e-6	2.01e-5 / 2.71e-6
	Forward substitution	5.06e-4 / 9.44e-5	1.87e-4 / 2.88e-5
	Backward substitution	2.29e-5 / 3.74e-6	1.04e-5 / 1.27e-6
60×60 / 60	Cholesky decomposition	8.89e-6 / 1.28e-6	3.97e-6 / 6.20e-7
	Forward substitution	9.35e-5 / 3.41e-5	2.26e-5 / 5.70e-6
	Backward substitution	1.98e-5 / 6.18e-6	4.13e-6 / 1.15e-6
120×120 / 30	Cholesky decomposition	1.38e-5 / 1.20e-6	8.70e-6 / 1.41e-6
	Forward substitution	1.26e-4 / 2.65e-5	5.95e-5 / 1.21e-5
	Backward substitution	1.17e-5 / 2.65e-6	5.80e-6 / 1.07e-6
30×30 / 30	Cholesky decomposition	3.33e-6 / 7.68e-7	1.80e-6 / 3.37e-7
	Forward substitution	2.20e-5 / 9.44e-6	6.90e-6 / 2.24e-6
	Backward substitution	8.59e-6 / 4.09e-6	2.62e-6 / 1.09e-6

Table 3 Error performance of the Simulink model and the synthesized RTL

ModelSim RTL simulations with the synthetically generated double-precision floating-point references **L**, **x**, and **y**. On average, the RTL implementation benefits from the fused datapath methodology and achieves higher precision than the standard IEEE 754 single-precision implementation as demonstrated by comparing the *Norm* in columns (3) and (4) in Table 3. We use the Frobenius *Norm* to get a measure of the overall error magnitude in the resultant matrix (or vector); it is calculated by:

$$\|E\|_F = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}|e_{ij}|^2}$$

Where,
N is the size of the matrix,
i,j are the row and column indices of the matrix respectively, and
e_{ij} is the error in the matrix element (i,j).

The Frobenius *Norm* for a vector is similar to that of a matrix but summation is only performed in one dimension. The maximum error is the maximum absolute error over all the elements e_{ij} in a matrix, or over all the elements e_i in a column vector.

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 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 example was a single-precision complex-data IEEE 754 floating-point Cholesky solver modeled in Simulink using the Altera DSP Builder Advanced Blockset. The design achieved a performance of 3,204 complex floating-point Cholesky solver executions per second with a matrix of size 240×240 at 200 MHz on a Stratix IV S360 FPGA device. This performance was achieved using the Altera DSP Builder Advanced Blockset tool flow (including Quartus II software and the Design Space Explorer tool) with no hand optimization or floor planning. Starting from a high-level block-based design in Simulink, the tool chain automatically pipelined, time optimized, and synthesized the design to achieve usable speeds and resource utilization.

The Altera floating-point DSP design flow incorporates the Altera DSP Builder Advanced Blockset, Altera's Quartus II RTL tool chain, and ModelSim simulator, as well as MathWorks' MATLAB and Simulink tools. 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 in the DSP Builder Advanced Blockset to enable a designer to perform quick algorithmic design space exploration.

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.

However, this new approach 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, there is a design methodology that should be understood and followed to support the flexibility afforded by Simulink. Careful forethought is necessary to create a design structure that allows for algorithm space exploration, such as that performed with matrix and vector sizes in the example presented in this paper.

Currently, designs using the DSP Builder Advanced Blockset are limited to the elements provided by the blockset to achieve optimized performance. Elements from the standard DSP Builder Blockset are not optimized with the floating-point compiler nor can 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.

Currently sampling, the next-generation 28-nm Stratix V and Arria V FPGAs contain a significantly larger number of multipliers and memory capacity. With the new variable-precision DSP block, and the higher precision 27×27 multiplier mode, a floating-point multiplication will require fewer resources than it currently does in the Stratix IV and Arria II devices. These enhancements and the potentially greater floating-point performance will further enable the use of floating-point designs on next-generation FPGAs.

6. References

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