Floating Point Vector Processing using 28nm FPGAs

Michael Parker
DSP Product Planning
Altera Corporation
San Jose, California

Dan Pritsker
Hardware Engineering Group
Altera Corporation
San Diego, California

I. Introduction

Vector processing is useful for implementation of many linear algebra algorithms used in many commercial, government and military applications. Typically, this is implemented using software on specialized multi-core CPU or GPU architectures. A compelling alternative is FPGA-based implementation, using floating point single precision implementation. This paper examines implementation of one such algorithm, the QR decomposition and back substitution, a common for solution of non-square over-determined systems of equations. This has been implemented using a mid-sized 28nm FPGA. Performance (GFLOPs), throughput, Fmax and power consumption are measured. The algorithm is implemented as a parameterizable core, which can be easily configured for all the matrix sizes benchmarked herein.

II. FPGA Design Entry Methodology

FPGA vendors have long offered floating point operator libraries such as multiply and add/subtract which have similar areas, performance levels, and latencies. The combination of multiple arithmetic operators into higher level functions such a vector dot product operator are inefficient, and suffer from significantly reduced F_{max}. Typical latencies for both multipliers and adders are in the range of 10 clock cycles; a dot product operator with a few tens of inputs may therefore exceed a latency of 100. Routing congestion and data path latencies are have been critical restrictions on floating point implementation on FPGA architectures. Parallelism is a key advantage of a hardware solution like FPGAs, but it is often not applied to floating point signal processing because the long latencies make the data dependencies in algorithms such as matrix decomposition difficult to manage. Therefore, the resultant systems offered poor performance levels, uncompetitive to other platforms such as GPU or multi-core CPU architectures.

An alternative FPGA design flow can overcome these issues. Rather than building up a data path from individual operators, the entire data path can be considered as a single function, with inter-operator redundancy factored out. Mantissa representation can be converted to hardware-friendly twos complement, and mantissa widths extended to reduce the frequency of normalizations. This approach, when combined with the Altera's new 28nm Variable Precision DSP block architecture, offers extremely high data processing capabilities, in excess of one Teraflop on a single FPGA die. High order math functions and basic operations are supported. Of interest in this application are the square root, and inverse square root functions. These can be implemented for little cost – usually 3-4 times the logic resources of a floating point adder or multiplier, and produce one result per clock cycle. This is in contrast to CPUs or GPUs , where various math.h functions can require up to 100 times more resources, in term of cycles.

This design flow, DSP Builder Advanced Blockset (DSPBA) facilitates these types of designs with several important features:

- Leverages Mathworks design environment, allowing the MATLAB/Simulink simulation to act as the test bench and design environment
- Fixed and floating point data type support, with automatic data-type propagation. The upstream source determines the data-type downstream unless it is purposely constrained or converted.
- The DSPBA takes advantage of Simulink's vector processing capability and most components support vector input/output and processing. It also has a "complex" data type, allowing easy use in DSP applications.
- The DSPBA performs registering and pipelining during its high level synthesis based upon the requested Fmax. It automatically balances and schedules all parallel paths. As a result, the design entry is behavioral in nature. The design is better described as an zero latency algorithmic block diagram, where the hardware specific elements such as pipeline registers, which you find in typical hardware oriented design schematics, are abstracted away during the design entry stage, and will be automatically inserted later during the compilation

stage. Only registers or delays which are a functional part of the algorithm are entered by the designer.

- Required processing latency in iterative loops is automatically computed by the tool. The designer merely places a FIFO or memory in the feedback path to model the cumulative pipeline latencies in the forward path. The depth of the FIFO is updated by the tool after simulation analysis.

- DSPBA toolflow supports variable mantissa floating point implementation. In addition to the common single (23 bit mantissa) and double precision (52 bit mantissa) formats, the design can also choose single reduced (16 bit mantissa) for reduced logic, or single extended (32 bit mantissa) for extra precision to stabilize certain linear algebra implementations.

III. QR Decomposition

Linear equation system is defined in matrix form using $Ax = b$ where A is a [m x n] matrix , x and b are n and m size vectors respectively. There many ways to solve this algebra problem and to find the unknown vector x. The direct method is using inverse A matrix to calculate x vector. However direct method suffers from mathematical complexity to find A inverse matrix and solution stability issues. On the other hand there are techniques to decompose A matrix to other simpler to solve matrices that simplify the linear equation system solution. QR Decomposition is one of these techniques.

Defining $A = Q \cdot R$ where Q is [m x n] orthogonal matrix and R is a [m x n] upper triangular matrix.

The new form of linear problem is given by: $Q \cdot R \cdot x = b$

Since Q matrix is orthogonal, meaning $Q^T \cdot Q = I$ and $Q^{-1} = Q^T$ Thus $R \cdot x = Q^T \cdot b \equiv d$

Once we find d by multiplying [matrix x vector] $Q^T \cdot b$, we can easily solve the system by doing back substitution operation since the R matrix is upper triangular matrix.

As defined above A is decomposed to Q orthogonal matrix and R upper triangular. For QR Decomposition, a number of algorithms exist, including Gram Schmidt and modified Gram Schmidt, Householder transformations and Givens rotations. Gram-Schmidt has implementation advantages: It largely consists of multiplications and additions, which are efficient to implement in FPGAs, in both fixed and floating point format. A variation referred to as "modified Gram-Schmidt" is similar in implementation cost and offers greater numerical stability. Givens rotations require more complex operations, such as arctan, cos, sin and square roots. Due to higher latency required, this is undesirable in the implementation of highly iterative algorithms. Householder transformations need to process in columns and rows. This is difficult in high speed applications, where parallel access across memory locations is required, since memory cannot easily be organized to provide easy access to both rows and columns at the same time.

IV. Gram Schmidt

Define the projection:

$$proj_e a \equiv \frac{\langle e, a \rangle}{\langle e, e \rangle} e$$

Where $\langle v, w \rangle$ is inner product that satisfies $\langle v, w \rangle = v^* \cdot w$ for complex numbers. According to Gram-Schmidt algorithm U_k is orthogonal set and E_k is corresponding orthonormal set after being normalized:

$$u_1 = a_1, e_1 = \frac{u_1}{\|u_1\|}$$

$$u_2 = a_2 - proj_{e1} a_2, e_2 = \frac{u_2}{\|u_2\|}$$

$$u_3 = a_3 - proj_{e1} a_3 - proj_{e2} a_3, e_3 = \frac{u_3}{\|u_3\|}$$

$$\vdots$$

$$u_n = a_n - \sum_{j=1}^{n-1} proj_{ej} a_n, e_n = \frac{u_n}{\|u_n\|}$$

After rearranging the equations so Ai on the left side:

$$a_1 = e_1 \|u_1\|$$

$$a_2 = proj_{e1} a_2 + e_2 \|u_2\|$$

$$a_3 = proj_{e1} a_3 + proj_{e2} a_3 + e_3 \|u_3\|$$

$$\vdots$$

$$a_n = \sum_{j=1}^{n-1} proj_{ej} a_n + e_n \|u_n\|$$

Rewriting the equations using the definition for "proj" and the fact that denominator $\langle e_i, e_i \rangle = 1$, as the set of E is orthonormal.

$$
\begin{aligned}
a_1 &= e_1\|u_1\| \\
a_2 &= \langle e_1, a_2\rangle e_1 + e_2\|u_2\| \\
a_3 &= \langle e_1, a_3\rangle e_1 + \langle e_2, a_3\rangle e_2 + e_3\|u_3\| \\
&\vdots \\
a_n &= \sum_{j=1}^{n-1}\langle e_j, a_n\rangle e_j + e_n\|u_n\|
\end{aligned}
$$

It can be rewritten in matrix form:

$$
\begin{bmatrix} a_1 & a_2 & a_3 & \dots \end{bmatrix} = \begin{bmatrix} e_1 | e_2 | \dots . | e_n \end{bmatrix} \begin{pmatrix} \|u_1\| & \langle e_1, a_2\rangle & \langle e_1, a_3\rangle & \dots \\ 0 & \|u_2\| & \langle e_2, a_3\rangle & \dots \\ 0 & 0 & \|u_3\| & \dots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} =
$$

$$
\begin{bmatrix} a_1 & a_2 & a_3 & \dots \end{bmatrix} = \begin{bmatrix} e_1 | e_2 | \dots . | e_n \end{bmatrix} \begin{pmatrix} \|u_1\| & \langle u_1, a_2\rangle \dfrac{1}{\|u_1\|} & \langle u_1, a_3\rangle \dfrac{1}{\|u_1\|} & \dots \\ 0 & \|u_2\| & \langle u_2, a_3\rangle \dfrac{1}{\|u_2\|} & \dots \\ 0 & 0 & \|u_3\| & \dots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}
$$

A = Q · R

V. Algorithm Implementation using DSPBA

The QR Decomposition can be described in code using the following looping construction, using Gram-Schmidt method.

```
for k=1:n
   r(k,k) = norm(A(1:m, k));
   for j = k+1:n
     r(k,j)=dot(A(1:m,k),A(1:m,j))/r(k,k);
   end
   q(1:m, k) = A(1:m, k) / r(k,k);
   for j = k+1:n
     A(1:m,j)=A(1:m,j)-r(k, j)*q(1:m,k);
   end
end
```

The calculations of the diagonal elements r(k,k) requires n square root operations and n*m complex multiplications and additions. The calculation of the other elements of *R* require approximately $n^2/2$ divides and $m*n^2/2$ complex multiplications and additions. The calculation of *Q* requires a further m*n divisions, and the updates to the original matrix *A* for further processing needs a further $m*n^2/2$ complex multiplications and additions. In total, n square root operations, $n^2/2$ + m*n divisions and $m*(n^2+n)$ complex multiplications and additions need to be performed.

High throughput requires minimizing computational latencies, as this is an iterative algorithms. The calculation of the diagonal elements r(k,k) must be completed before subsequent elements of *R* or *Q* can be calculated. The calculation of q(1:m,k) must finish prior to updating the *A* matrix for processing the next column. This can cause large number of inactive or idle cycles, where no useful computations can be performed. The number of inactive cycles that is introduced is $k*l_r$, where l_r is the latency, in clock cycles, for calculating r(k,k), and $k* l_q$, where l_q is the latency of the calculation of q. Therefore, the calculation of the diagonal elements is split into a dot product, which returns the squared magnitude r2, and the square root calculation. The calculation of the other elements of R can be split into a dot product that returns the un-normalised version rn of element r, and a division by r(k,k) to compete the normalization. The calculation of A can be optimized by substituting q(1:m,k) with A(1:m,k/r(k,k), and r(k,j) can be substituted with rn(k,j)/r(k/k). This allows reordering the algorithm into two separate loops:

For loop 1:

```
for k=1:n
   r2(k,k) = dot(A(1:m, k), A(1:m,k);
   for j = k+1:n
     rn(k,j) = dot(A(1:m, k), A(1:m, j));
   end
   for j = k+1:n
     A(1:m,j)=A(1:m,j)-(rn(k,j)/r2(k,k))*A(1:m,k);
   end
end
```

For loop 2:

```
for k=1:n
   r(k,k) = sqrt(r2(k,k));
   for j = k+1:n
    r(k,j) = rn(k,j)/ r(k,k);
   end
   q(1:m,k) = A(1:m,k) / r(k,k);
end
```

These loops are simulated in the MATLAB/Simulink environment, and then implemented in an optimized fashion in the FPGA using DSPBA. This different than the timing optimizations performed by Quartus II or similar FPGA design tools. For design entry using Verilog or VHDL, the Quartus II tool is performs place and route the design to minimize delays on critical paths in the design. Design optimization is not possible, only placement/routing. DSPBA will change the design itself to the optimize critical paths, performing the designer from the timing closure process. Additional benefits are design reuse, as the tool free the designer from needing to take advantage of features particular to a particular FPGA architecture. In this way, DSPBA also "future-proofs" designs, allowing a given design to be easily

ported to future FPGA families with new features or higher levels of performance in an automated fashion.

The output of the DSPBA tool is a VHDL file optimized for the FPGA device specified by the designer. This is then input to the Quartus project, and can be integrated into any other circuit blocks in the FPGA, with ports for I/O, memory mapped register access, or any necessary external DDR memory access.

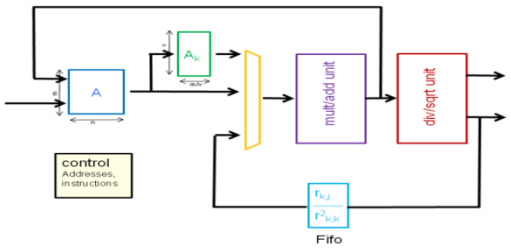

Figure 1: QRD Block Diagram

The implementation can the thought of as building a vector processing engine, with separate blocks for the vector processor, control circuits and the memory. Control circuitry for the processing engine is automatically generated by the tool. One useful control block is a nested "for-while" looping block, which allows for easy implementation of complex looping and address structures. The width of the vector engine is referred to as the vector size, and is a design parameter. This controls the degree of parallelism of the design. The matrix width is frequently a multiple of the vector size. In this way a trade-off between FPGA resources and matrix throughput can be achieved.

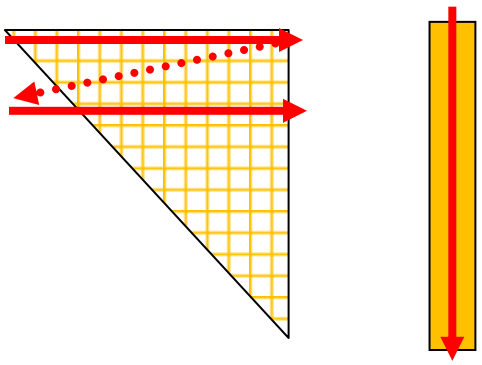

Figure 2: R *Matrix and Q'*b sample calculation order*

An advantage of this transformation is that the calculations of Q can be need not be stored, as Q^T each row is used as generated to calculate each value for the vector d. Since internal memory is limited, efficient use is required. With QR Decomposition, we need to continuously read and write from a large memory, which stores the entire matrix. One write port is sufficient, but it would be desirable to have two read ports for the calculations of the new A values and the rn values, as they both require the k^{th}vector A(1:m,k) which is combined with the remaining vectors A(1:m,j) . Instead of using a separate read port on the large A memory, an additional memory is introduced that just stores the l^{th} vector. This memory is loaded during the r(k,k) calculation, when A(1:m,k) is read from the A memory, and stored for re-use for the other operations.

Latency modeling is also required in feedback loops. This must be accounted for in the design, as it will be determined by the algorithmic and circuit pipeline delays present in algorithm datapath. In this case, all the latency is placed into a single memory or FIFO in the feedback path, and the DSPBuilder tool will distribute this delay throughout the design as needed to assure optimal performance. The tool will indicate and update the design with the minimum amount of delay required in the feedback memory.

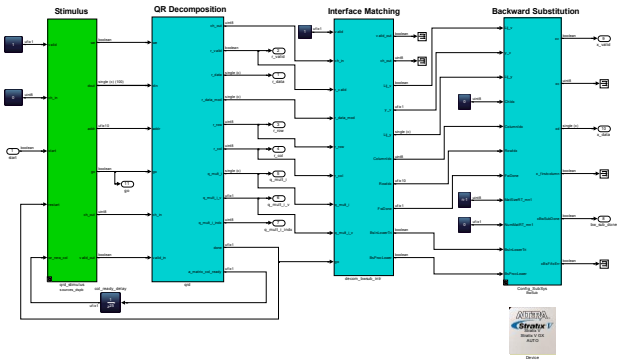

Figure 3: Top level design file in Simulink

The QR Decomposition plus back substitution are packaged hierarchically into a parameterizable core, where the user can specify the vector engine width and maximum matrix size at compile time, and the desired matrix size at run time.

This level of automated optimization is far more than the timing optimizations performed by Quartus II or similar FPGA design tools. For design entry using Verilog or VHDL, the Quartus II tool is limited to trying to place and route the design to minimize delays on critical paths in the design. It cannot optimize the actual design itself. In contrast, DSPBuilder is able to change the design itself to optimize critical paths, relieving the designer from the timing closure process. Additional benefits are allowing easy design updates, such as parameterizing the number of channels, FIR filter lengths, vector sizes and many other aspects of the design. DSPBuilder also "future-proofs" designs, allowing a given design to be easily ported to different FPGA families, including future FPGA families in an automated fashion.

VI. GFLOPs, Performance and Resource Usage

The FLOPs of the QRD algorithm is defined as follows, where n and m are matrix rows and columns respectively. This is for complex matrices, although the number of floating point operations is specified in real or scalar operations.

Algorithm Step	Number of Real FLOP
QR Decomposition	$5.33mn^2$
$Q^T \cdot b$	$8mn-2n$
Backward Substitution	$4n^2$
Total:	$5.33mn^2+8mn-2n+4n^2$

Table 1: QRD Solver Real Flops

The performance and resource usage of the QR Decomposition core is shown for several matrix and vector size combinations. Since these are user defined parameters, any reasonable size can be easily generated by the QRD core. All results are using single precision floating point, with complex input and output data.

The FPGA used to compile the QRD core is a mid-sized Stratix V FPGA, specifically the **5SGSD5** in –C2 speed grade. The Fmax figure shown is actually exceeded in most compile results. In some cases, faster performance can also be achieved using the Quartus II DSE feature.

The level of resources used indicates that multiple QRD cores may be built in the same device depending upon the matrix/vector size. This is particularly true if a larger Stratix V FPGA device is used.

By examining Table 1, it is evident that the GFLOPS, logic and multiplier resources are approximately proportional to the vector size chosen. The memory resources are approximately proportional to matrix size chosen.

Input Matrix Size	Vector Size	ALUTs / Memory blocks / 27x27s	Latency @ Operating frequency	Throughput (Matrix per second)	GFLOPS per core (complex single precision)
50x100	50	105K 230 M20K 227 mults	45 us @ 250 MHz	31,681	43.8
100x200	50	106K 304 M20K 228 mults	213 us @ 250 MHz	5,920	64.3
100x200	100	202K 504 M20K 428 mults	173 us @ 200 MHz	8,467	91.9
250x400	100	200K 858 M20K 428 mults	1586 us @ 200 MHz	789	106
400x400	100	203K 1566 M20K 428 mults	4029 us @ 200 MHz	310	106
450x450	75	157K 1985 M20K 328 mults	7121 us @ 200 MHz	165	80

Table 2: QRD performance using Stratix V 5SGSD5 FPGA

The throughput is a number of matrices processed per second, includes both the QR decomposition and back substitution. The latency is the time from the load in of the last input data sample to the reading out of the last output sample.

Power consumption and GFLOPS per watt measurements are also presented. The reader should keep in mind that these are actual measured GFLOPS/W on a complex algorithm (QRD), and should not to be compared to other published figures showing theoretical GFLOPS/W, often using a trivial implementation which is just exercising multipliers. This approach can result in "marketing" figures an order of magnitude higher, but are not realistic in what an actual application will experience.

Input Matrix Size	Vector Size	Throughput (Matrix per second)	GFLOPS per core (complex single precision)	Core power consumption as measured using Altera 5SGSD5 eval board	GFLOPS/Watt
50x100	50	31,681	43.8	10.77 W	4.07
100x200	50	5,920	64.3	13.9 W	4.63
100x200	100	8,467	91.9	20.97 W	4.38
400x400	100	310	106	25.20 W	4.21
450x450	75	165	80	20.25	3.95

Table 3: QRD GFLOPs/Watt using Stratix V 5SGSD5 FPGA

VII. Numerical Accuracy Concerns

These results are achieved using an FPGA based parallel processing architecture. Therefore, the results will not precisely match the same algorithm implemented serially with a microprocessor architecture (the same situation exists for most GPU implementations). To address this concern, care has been taken to assure that the hardware based results are equally or better accurate than those achieved by IEEE754 compliant CPU architectures. This is accomplished by using a larger then required mantissa width. For example, Stratix V FPGAs employ thousands of native 27x27 size hardened multipliers, which is larger than the 23x23 size

multiplier required to implement single precision floating point IEEE754.

The numerical precision is evaluated by first computing results in MATLAB using double precision. These results are compared to both single precision results using MATLAB on an IEEE754 compliant PC, and the single precision results computed in the FPGA. Both the maximum error and normalized error are shown. The normalized error is found using the Frobenius norm determined by:

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

The resultant errors are tabulated for a sample collection of matrix and vector sizes. The parallelized FPGA based signal processing produces normalized errors that are smaller than that computed the CPU architecture, employing an IEEE754 based serial architecture. The maximum error found is also slightly smaller on FPGA, compared to IEEE754.

Matrix / Vector Size	MATLAB using computer Norm/Max	DSPBA generated RTL Norm/Max
50x100 / 50	5.01e-005 / 6.42e-006	4.87e-005 / 6.02e-006
100x200 / 100	2.3e-5 / 1.24e-6	1.68e-5 / 9.97e-7
400x400 / 100	8.8e-5 / 4.81e-6	7.07e-5 / 4.03e-6

Table 4: QRD floating point error comparison

Several matrix and vector are presented to show the performance attainable in reasonable large matrix processing core. All possible permutations and options cannot be explored within this example design. However, due to the ability to design and quickly develop within the Mathworks environment, many design options can be investigated quickly, without resorting to Quartus II FPGA compiles, which is in contrast to HDL design methodology. Floating point designs of this complexity and performance are not feasible using traditional HDL design techniques.

VIII. Summary

Production released FPGAs and tools are available to build high throughput, low latency floating point linear algebra and other functions, which exceed the capabilities of CPUs and DSPs, and rival that of latest GPUs. In addition, the native connectivity, streaming capabilities, and power consumption advantages of FPGAs can provide a significant advantage over GPU based solutions. The resultant FPGA designs may also be used as hardware accelerators to off-load CPUs, allowing much of the existing code base to be preserved on current processors while still allowing a dramatic increase in system throughput or inclusion of higher computation rate algorithms to meet requirements.

Using the DSPBA design flow, similar results have been achieved using FPGAs on other designs, such as floating point FFTs, matrix multiplies, Cholesky decomposition, LU decomposition, and other functions.

A vendor independent benchmarking effort has been completed by Berkeley Design Technology Inc (BDTi) using QR and Cholesky decomposition cores on 28nm Stratix V and Arria V FPGAs. This report is available at www.bdti.com

References

[1] Michael Parker and Colman Cheung, "Hardware Based Floating Point Processing" *Proceedings of the 2011 HPEC conference*, September 2011, 2009

[2] Michael Parker and Volker Mauer, "Floating Point STAP Implementation on FPGAs," *Proceedings of the 2011 RadarCon Conference*, May 2011, 2009

[3] Mark Jervis, "Advances in DSP Design Tool Flows for FPGAs, MILCOM 2010

[4] Suleyman S. Demirsoy and Martin Langhammer, "Fused Datapath Floating Point Implementation of Cholesky Decomposition," *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February, 2009

[5] M. Langhammer, "High performance matrix multiply using fused datapath operators," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*. IEEE, October 2008, pp. 153-159

[6] M.A. Richard – Fundamentals of Radar Signal Processing