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 Fmax. 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 $[m \times n]$ 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 a $[m \times n]$ orthogonal matrix and $R$ is a $[m \times n]$ 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 $[\text{matrix x vector}] Q^T \cdot b$, 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:

$$\text{proj}_i a = \frac{\langle e, a \rangle}{\langle e, e \rangle} e$$

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:

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

$$u_2 = a_2 - \text{proj}_{e_1} a_2, e_2 = \frac{u_2}{\|u_2\|}$$

$$u_3 = a_3 - \text{proj}_{e_1} a_3 - \text{proj}_{e_2} a_3, e_3 = \frac{u_3}{\|u_3\|}$$

$$\vdots$$

$$u_n = a_n - \sum_{j=1}^{n-1} \text{proj}_{e_j} a_n, e_n = \frac{u_n}{\|u_n\|}$$

After rearranging the equations so $a_i$ on the left side:

$$a_1 = e\|u_1\|$$

$$a_2 = \text{proj}_{e_1} a_2 + e_2\|u_2\|$$

$$a_3 = \text{proj}_{e_1} a_3 + \text{proj}_{e_2} a_3 + e_3\|u_3\|$$

$$\vdots$$

$$a_n = \sum_{j=1}^{n-1} \text{proj}_{e_j} a_n + e_n\|u_n\|$$

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.
\[
a_1 = e_1 \|u_1\|
\]
\[
a_2 = \langle e_1, a_2 \rangle e_1 + e_2 \|u_2\|
\]
\[
a_3 = \langle e_1, a_3 \rangle e_1 + \langle e_2, a_3 \rangle e_2 + e_3 \|u_3\|
\]
\[
\vdots
\]
\[
a_n = \sum_{j=1}^{n-1} \langle e_j, a_n \rangle e_j + e_n \|u_n\|
\]

It can be rewritten in matrix form:
\[
\begin{bmatrix}
a_1 & a_2 & a_3 & \ldots \\
a_1 & a_2 & a_3 & \ldots \\
a_1 & a_2 & a_3 & \ldots \\
\end{bmatrix}
= 
\begin{bmatrix}
\|u_1\| & \langle u_1, a_2 \rangle & \langle u_1, a_3 \rangle & \ldots \\
0 & \|u_2\| & \langle u_2, a_3 \rangle & \ldots \\
\vdots & \vdots & \vdots & \ddots \\
\end{bmatrix}
\]

\[
A = QR
\]

V. Algorithm Implementation using DSPBA

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

```matlab
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
```

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 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.

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.

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.

<table>
<thead>
<tr>
<th>Algorithm Step</th>
<th>Number of Real FLOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>QR Decomposition</td>
<td>5.33mn²</td>
</tr>
<tr>
<td>$Q^T \cdot b$</td>
<td>8mn − 2n</td>
</tr>
<tr>
<td>Backward Substitution</td>
<td>4n²</td>
</tr>
<tr>
<td>Total:</td>
<td>5.33mn² + 8mn − 2n + 4n²</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>Input Matrix Size</th>
<th>Vect or Size</th>
<th>ALUTs / Memory blocks / 27x27s</th>
<th>Latency @ Operating frequency</th>
<th>Throug h-put (Matrix per second)</th>
<th>GFLOPS per core (complex single precision)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50x100</td>
<td>50</td>
<td>105K</td>
<td>45 us @ 250 MHz</td>
<td>31.681</td>
<td>43.8</td>
</tr>
<tr>
<td>100x200</td>
<td>50</td>
<td>106K</td>
<td>213 us @ 250 MHz</td>
<td>5,920</td>
<td>64.3</td>
</tr>
<tr>
<td>100x200</td>
<td>100</td>
<td>202K</td>
<td>173 us @ 200 MHz</td>
<td>8,467</td>
<td>91.9</td>
</tr>
<tr>
<td>250x400</td>
<td>100</td>
<td>200K</td>
<td>1586 us @</td>
<td>789</td>
<td>106</td>
</tr>
</tbody>
</table>

*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 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.

<table>
<thead>
<tr>
<th>Input Matrix Size</th>
<th>Vect or Size</th>
<th>Throug h-put (Matrix per second)</th>
<th>GFLOPS per core (complex single precision)</th>
<th>Core power consumptio n as measured using Altera 5SGSD5 eval board</th>
<th>GFLOP s/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>50x100</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td>10.77 W</td>
</tr>
<tr>
<td>100x200</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td>13.9 W</td>
</tr>
<tr>
<td>100x200</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td>20.97 W</td>
</tr>
<tr>
<td>400x400</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td>25.20 W</td>
</tr>
<tr>
<td>450x450</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td>20.25 W</td>
</tr>
</tbody>
</table>

*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:

\[ \|E\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} e_{ij}^2} \]

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.

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

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


