对Altera 28-nm FPGA浮点DSP设计流程和性能的独立分析

作者：Berkeley设计技术有限公司员工

2012年10月

简介

FPGA越来越多的用作实现要求较高的数字信号处理并行处理引擎。基准测试结果表明，对于并行度很高的工作负载，与数字信号处理器(DSP)和通用CPU相比，FPGA能够进一步提高性能，性价比非常高。但是，目前为止，FPGA一直还仅用于定点DSP设计。对于需要高性能浮点计算的应用，FPGA还没有被认为是高效的实现平台。由于较长的处理延时和布线拥塞，FPGA浮点效率和性能一直较低。此外，基于用Verilog或者VHDL编写寄存器传送级硬件描述的传统的FPGA设计流程并不适合实现复数浮点算法。

Altera开发了新的浮点设计流程，目的是在Altera FPGA上简单方便的实现浮点数字信号处理算法，与以前相比，进一步提高设计性能和效率。浮点编译器并没有构建由基本浮点算子(例如，先是乘法，然后是加法和平方运算)组成的数据通路，而是产生一个融合数据通路，此通路可将基本算子组合在一个函数或者数据通路中。这样，避免了传统浮点FPGA设计中的重复表示。而且，Altera设计流程是基于模型的高层设计流程，它使用了Altera的DSP Builder高级模块库，以及MathWorks的MATLAB和Simulink。Altera希望通过在高层次上工作，与基于HDL的传统设计相比，使设计人员能够更高效迅速地实现并验证复数浮点算法。

BDTI对Altera浮点DSP设计流程进行了独立分析。BDTI的目的是评估Altera FPGA在实现要求较高的浮点DSP应用时的性能，以及Altera浮点DSP设计流程的易用性。本文阐述了BDTI的评估结果以及背景和详细的方法。

目录

1. 引言

两个浮点设计实例

数字芯片技术的进步使得复数算法能够在常用的嵌入式计算应用中实现，而这以前只限于研究环境。例如，线性代数多年以来(特别是用于解具有大量联立线性方程的系统)主要用在研究环境中，这种环境能够提供大量的计算资源，但通常不需要进行实时计算。解大规模系统需要矩阵求逆或者某些矩阵分解算法。这些方法不但需要进行大量的计算，而且，如果没有足够的动态范围，还有可能出现数值不稳定问题。因此，只有在浮点器件中才有可能高效精确的实现这类算法。

Altera最近在DSP Builder高级模块库工具链中引入了浮点功能，以简化在Altera FPGA上的浮点DSP算法实现，此功能与传统的FPGA设计方法相比提高了浮点设计的性能和效率。BDTI在以前的一篇白皮书[1]中分析并评估了Altera Quartus II软件v11.0工具链单通道浮点Cholesky求解器实现实例的性能和效率，针对40-nm Stratix IV和Arria IV FPGA进行了综合。

在本文中，我们评估Altera使用新版Quartus II软件v12.0工具链方法的效率，以及Altera 28-nm Stratix V和Arria V FPGA的性能。针对这一评估，BDTI的重点是使用两类分解方法来解大规模联立线性方程：多通道Cholesky矩阵分解以及使用Gram-Schmidt过程的QR分解。这些分解结合了前向和后向代换，在一组联立线性方程Ax = B中，解出矢量x。

矩阵分解用在很多高级军用雷达应用中，例如，空时自适应处理(即，STAP)，以及数字通信中的各类估算问题等。QR分解常用于普通m乘n矩阵**A**中，而Cholesky分解以其计算效率高而成为应用于平方、对称以及正定矩阵的首选算法。这些分解方法的计算量非常大，需要很高的数据精度，因此，必须使用浮点计算。此外，本文中研究的这两个例子在其核心算法中使用矢量点乘和嵌套循环算法，很多涉及到线性代数和有限冲击响应(FIR)滤波器的数字信号处理应用都使用了这些算法。

在本文所介绍的例子中，采用这两种方法解一组联立复数线性方程，并得出其结果。第4部分介绍了使用QR求解器，Altera Stratix V FPGA能够以运行速率为203 MHz完成每秒315次400×400矩阵分解计算，浮点运算是每秒162×10^9次(GFLOPS)。

本文评估的这两个设计实例是其DSP Builder软件v12.0工具链的一部分，也可以通过floatingpoint@altera.com申请获得。

浮点设计流程

传统上，FPGA并不是要求较高的浮点应用的平台选择。FPGA供应商虽然提供了浮点基本库，但是，FPGA在浮点应用中的性能表现非常有限。由于浮点算子很深的流水线特性以及较宽的算法结构，产生了较大的数据通路延时和布线拥塞，因此，传统的浮点

符号和定义

M粗体大写字符表示一个矩阵。

z粗体小写字符表示一个矢量。

L*表示矩阵L的共轭转置矩阵。

l*表示元素l的共轭转置。

厄米矩阵 平方矩阵，复数部分等于自己的共轭转置。这是实数对称矩阵的复数扩展。

正定矩阵 如果对于所有非零复数矢量**z**，**z*Mz** > 0，那么厄米矩阵**M**是正定的。出于本文的目的，**M**是厄米矩阵，因此**z*Mz**总是实数。

标准正交矩阵 如果矩阵$\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$，其中，**I**是单位矩阵，那么**Q**是标准正交的。

Cholesky分解 将厄米正定矩阵**M**因数分解成下三角形矩阵**L**及其共轭转置**L***，**M** = **LL***。

QR分解 将大小为*m*乘*n*的矩阵**M**分解成一个大小为*m*乘*n*的标准正交矩阵**Q**和一个大小为*n*乘*n*的上三角矩阵**R**，这样，**M** = **QR**。

Fmax FPGA设计的最大频率。

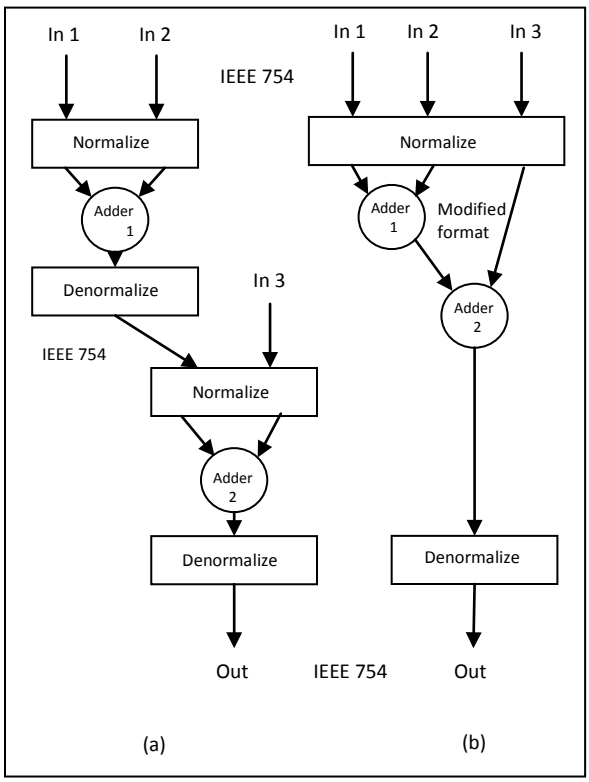

图1 (a) 传统浮点实现(b) 融合数据通路实现

FPGA设计效率较低。延时也在需要进行大量数据处理的设计中产生了难以处理的问题。最终的结果是设计的工作频率非常低。

Altera DSP Builder高级模块库工具流程在体系结构层面以及系统设计层面解决了这些问题。Altera浮点编译器将大部分数据通路融合至一个浮点功能中，而不是通过组合基本浮点算子来构建它们。它分析数据通路的位增长，选择最优归一化输入，为数据通路分配足够的精度，尽可能消除归一化和去归一化步骤，从而实现了这一点，如图1所示。IEEE 754格式仅用于数据通路边界；在数据通路内部，使用了较大的尾数宽度，以提高动态范围，减少对连续算子之间去归一化和归一化步骤的需求。归一化和去归一化函数使用了48位宽桶形移位寄存器来实现单精度浮点数。这需要大量的逻辑和布线资源，是在FPGA上实现浮点算法效率不高的主要原因。*融合数据通路*方法避免了大量使用这类桶形移位寄存器。在Stratix V和Arria V器件中，涉及到高精度尾数的乘法运算使用了Altera的27位×27位乘法器模式。图1(b)显示了融合数据通路方法，与图1(a)传统的实现方法相比，这一简单实例使用了两个加法链。图1(b)中的融合数据通路不需要对第一个加法器的输出进行去归一化，以及对第二个加法器的输入进行归一化，因此避免了算子交叉冗余。不需要使用额外的逻辑和布线资源，而是使用硬核乘法器，因此，能够预测复数数据通路上的时序和延时。实现单精度和双精度IEEE 754浮点算法不但减少了对逻辑资源的占用而且还提高了性能。Altera宣称，与由基本算子构成的等价数据通路相比，融合数据通路方法减少了50%的逻辑，延时减小了50% [2]。结果，内部数据表示范围更宽，总的数据精度一般高于使用基本IEEE 754浮点算子库的方法。

Altera浮点DSP设计流程采用了Altera DSP Builder高级模块库、Altera的Quartus II软件工具链、ModelSim仿真器，以及MathWorks的MATLAB和Simulink。利用Simulink环境，设计人员能够在算法行为级来描述、调试并验证复杂系统。DSP Builder高级模块库采用了数据类型传播以及矢量数据处理等Simulink特性，使设计人员能够快速进行算法设计空间管理。

为评估Altera浮点设计流程的效率和性能，BDTI使用了DSP Builder高级模块库工具流程，验证两个实例，使用单精度浮点表示的复数数据类型解一组联立线性方程。使用Cholesky分解得到一个解，而另一个则通过Gram-Schmidt过程，使用QR分解得出解。本文的第2部分介绍怎样实现两个浮点实例。第3部分介绍BDTI在设计流程和工具链上的使用体验。第4部分介绍在两种不同Altera FPGA上实现的性能：高端中等容量Stratix V 5SGSMD5K2F40C2N器件和中端Arria V 5AGTFD7K3F40I3N器件。最后，第5部分是BDTI的结论。

2. 实现

背景

在很多应用中都会遇到一组线性方程**Ax** = **b**。无论这是一个涉及到线性最小平方的优化问题，Kalman滤波器预测问题，还是MIMO通信通道估算问题，都需要找到一组线性方程Ax = b的数值解。对于大小为*m*乘*n*的普通矩阵，*m*是矩阵高度，*n*是宽度，QR分解可以

用于解出矢量**x**。这一算法将**A**分解成一个大小为*m*乘*n*的标准正交矩阵**Q**和一个大小为*n*乘*n*的上三角矩阵**R**。由于**Q**是标准正交的，因此，$\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$，$\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$。而**R**是上三角矩阵，用后向代换方法很容易解出**x**，甚至不需要置反原始矩阵**A**。在本文的QR实例中，我们应用了$m \geq n$的超定矩阵。

当矩阵A是对称而且正定时，例如很多问题中出现的协方差矩阵，通常使用Cholesky分解和求解器。此算法可找到矩阵A的逆矩阵，从而解出$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$中的矢量x。取决于分解时所使用的算法，Cholesky分解的效率至少是QR分解方法的两倍。由于这些分解算法本质上都是递归的，需要进行除法，因此，随着矩阵规模的增大，数字动态范围也在不断增大。例如，对于MIMO通道估算中的矩阵规模只有4×4，也需要使用浮点运算来实现。对于需要较大吞吐量的大规模系统，例如，军事应用中的例子，嵌入式系统一般很难实现所需的浮点运算速率。设计人员通常舍弃整个算法，而寻求次优解决方案，或者，使用多个高性能浮点处理器，从而增加了成本，增大了设计投入。

体系结构简介

Cholesky求解器

在我们的设计实例中，通过在FPGA中作为以流水线方式并行工作的两个子系统来实现Cholesky求解器。第一个子系统执行Cholesky分解和前向代换——名为*Cholesky求解器*的工具条中的第1步和第2步。第二个子系统执行后向代换——同一工具条中的第3步。由于输入矩阵是厄米矩阵，分解产生复数共轭转置三角矩阵，因此，只装入输入矩阵A的下三角部分，覆写它，生成下三角矩阵L，从而提高了对存储器的利用率。这两个子系统都采用了输入级和处理级流水线运作，，这样允许在同一存储器的区域里进行处理，而另一半用于装入新数据。分解和前向置换级的输出被传送到后向置换的输入级，如图2所示。

分解的核心是复数矢量点乘引擎(也称为矢量乘法器)，用于计算方程(3)和(4)。对于Stratix V器件，矢量长度(VS)的上限为90个复数元素，而对于Arria V器件，实现中可用复数元素的矢量长度上限为45。矢量长度也对应于每个时钟周期需要为点乘引擎提供一组新的数据所需的并行存储器读操作数，，从而确定片内双端口存储器的宽度，以及怎样划分这些存储器。出于实施方面的考虑，将大小一定的矩阵存储器分成ceil(N/VS)存储块，其中，ceil()是取整函数，N是矩阵的大小。

CHOLESKY求解

解出**Ax** = **b**中矢量**x**的Cholesky递归算法有三个步骤：第1步。分解，找到下三角矩阵**L**，其中，**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

注意，上面方程的相关性。方程(4)中的斜对角元素仅与同一行中的左侧元素相关。非对角元素与同一行中的左侧元素，以及其上相应对角元素的左侧元素相关。

第2步。前向代换，例如，解出方程**Ly** = **b**中的**y**，

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

for i = 2 to n,

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

end

第3步。后向代换，例如，解出方程**L*** **x** = **y**中的**x**，

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

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

其中，

n = 矩阵**A**的维数

l_{ij} = 矩阵**L**第*i*行第*j*列的元素

a_{ij} = 矩阵**A**第*i*行第*j*列的元素

y_i = 矢量**y**的第*i*行元素

b_i = 矢量**b**的第*i*行元素

x_i = 矢量**x**的第*i*行元素

第1步的输出是Cholesky分解，第3步的输出是线性方程**Ax** = **b**的解**x**。注意，算法直接找到矩阵**A**的转置，解出**x** = $\mathbf{A}^{-1}$ **b**。

Cholesky求解器设计是一种多通道实现方式。通道最大数是一个编译时间参数，仅受器件可用存储器的限制。存储器划分与单通道实现方式一致，只是使用了相同结构的多个副本。

分解运算每次处理一个元素，按照列的顺序，从左上角开始，按照垂直之字形方式进行，在矩阵右下角结束，如图3(a)所示。首先计算每一列的对角元素，然后是其下同一列中的所有非对角元素，再转到右侧下一列顶部的对角元素。采用四级嵌套*for*循环对事件和迭代进度进行控制。最外层循环实现按列处理；第二层循环实现按块处理；第三个内循环处理行；最内层循环处理多个通道。将通道处理放在最内层循环能够有效的将浮点累加器转换为时间共享累加器，更好的隐藏其延时。DSP Builder高级模块库中的*NestedLoops*模块在一个处理模块中集成了三个嵌套循环，与作为三个分立的*for-loop*模块实现的相似功能相比，这提高了处理速度，更有效的使用了资源。模块提取出了最复杂的循环控制信号，缩短了设计和调试时间，循环结构更具可读性。

点乘引擎对矩阵的每一行进行操作，在一个周期中，同时对方程(3)、(4)和(6)中的求和项进行矢量乘法计算。环形存储器结构用在点乘引擎的输入上，对多个输入矩阵的行进行循环处理。对于矢量点乘长度比矢量长度短的情况，屏蔽未使用的项，不含在求和项中。对于点乘长度长于矢量大小的情况，计算部分乘积和，在块边界进行保存，直到该行中某一元素所有块的输出能够用于最终的求和计算，如图3(b)所示。使用DSP Builder高级模块库的浮点加法器块，在一个累加器循环中进行块输出求和计算。反馈循环的延时是13个周期。通过将软件实现中传统的*for块循环和for行*循环顺序进行交换，并增加多通道处理，隐藏了浮点累加器的延时，提高了硬件的利用率。DSP Builder高级模块库自动计算循环延时，以解决这类延时问题。通过检查*Loop Delay*块中的*Minimum delay*框，可以让工具计算并填充最小延时。可以将导致同一延时的通路分配相等的组号码，工具会给组中所有的成员分配相同的延时值。DSP Builder高级模块库提供了专用浮点累加器，虽然并没有用在本文评估的实例中，但是，用户可以定制累加器，配置最大输入容量和

所需的累加精度，来优化其速率和资源需求。该模块支持在较高时钟速率时，一个浮点数据流的每个采样单周期的累加计算。

第二个子系统进行后向代换。这一子系统有自己的输入和输出存储器模块。与Cholesky/前向代换子系统相似，以流水线方式输入到输入级和处理级。与分解操作的N^3相比，后向代换的复杂度在N^2量级，因此，点乘不采用矢量处理。相反，点乘采用一个复数乘法器，这能够满足Cholesky分解以及前向代换子系统的要求。

QR求解器

如图4所示，作为以流水线方式并行工作的两个子系统来实现QR分解和求解器。在名为*QR*求解器的工具条中，第一个子系统执行第1步和第2步，而第2个子系统执行第3步中的后向代换。后向代换子系统与Cholesky求解器中使用的相同。以单个四级深度嵌套循环的方式来实现Cholesky的方程(1)至(6)，与之不同，QR求解器采用了一个简单有限状态机(FSM)，通过四次主要操作进行循环处理；找到方程(2)中的矢量的平方值，方程(4)中两个矢量的点乘，从方程(6)中的矢量中提取出点乘，以及方程(7)中的点乘，以找到矢量**d**。DSP Builder高级模块库中的*NestedLoop*模块用于FSM的所有阶段中，以产生数据通路的所有控制和事件信号。

上面列出的四次操作中的公共处理模块是点乘引擎。为增强性能，采用了矢量处理来计算点乘。与Cholesky设计相似，矢量大小是编译时间参数。为能够在FSM的所有四个状态中重新使用这一引擎，在其输入上使用了一个数据复用器，由FSM事件控制器对其进行控制。对于FSM的每一状态，复用器为点乘引擎选择正确的输入。点乘引擎以及浮点累加器的详细组成与Cholesky设计的相似，因此，这里没有再列出。

通过重新使用主内核存储器模块，该模块最初保持了矩阵**A**和输入矢量**b**，优化了QR分解所需的存储器。在内核存储器中，处理过程按列进行，从左到右。使用了一列之后，并且不再需要该列了，由新矩阵中相应的列将其覆写。对老矩阵**A**进行了分解之后，由新矩阵替换这一存储器模块中最初的内容，从而维持了背靠背矩阵处理功能，不会出现任何阻塞。

QR求解

通过QR分解，使用Gram-Schmidt过程解出**Ax** = **b**中**x**的三个步骤：

第1步。分解大小为*m*-乘*n*的矩阵**A**，即，找到大小为*m*乘*n*的矩阵**Q**，大小为*n*乘*n*的**R**，其中，**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

注意，在这一组重构方程组中，并没有明确的计算出标准正交矩阵**Q** = [$\mathbf{q}_1\ \mathbf{q}_2 \ldots \mathbf{q}_n$]，而是找到并使用了其正交形式**U** = [$\mathbf{u}_1\ \mathbf{u}_2 \ldots \mathbf{u}_n$]，其中，$\mathbf{q}_i = \mathbf{u}_i / \|\mathbf{u}_i\|$。

第2步。计算**d**，其中，**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

第3步。后向代换，即，解出方程**Rx** = **d**中的**x**。

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

其中，

m = 矩阵**A**的行数
n = 矩阵**A**的列数
$\mathbf{u}_i$ = 矩阵**U**的第i^{th}列
u_{ij} = 矩阵**U**第*i*行第*j*列的元素
r_{ij} = 矩阵**R**第*i*行第*j*列的元素
$\mathbf{a}_i$ = 矩阵**A**的第i^{th}列
a_{ij} = 矩阵**A**第*i*行第*j*列的元素
d_i = 矢量**d**的第*i*行元素
bi = 矢量**b**的第*i*行元素
xi = 矢量**x**的第*i*行元素

QR求解器找到线性方程**Ax** = **b**的解**x**，不用找到未定义的转置矩阵**A**。

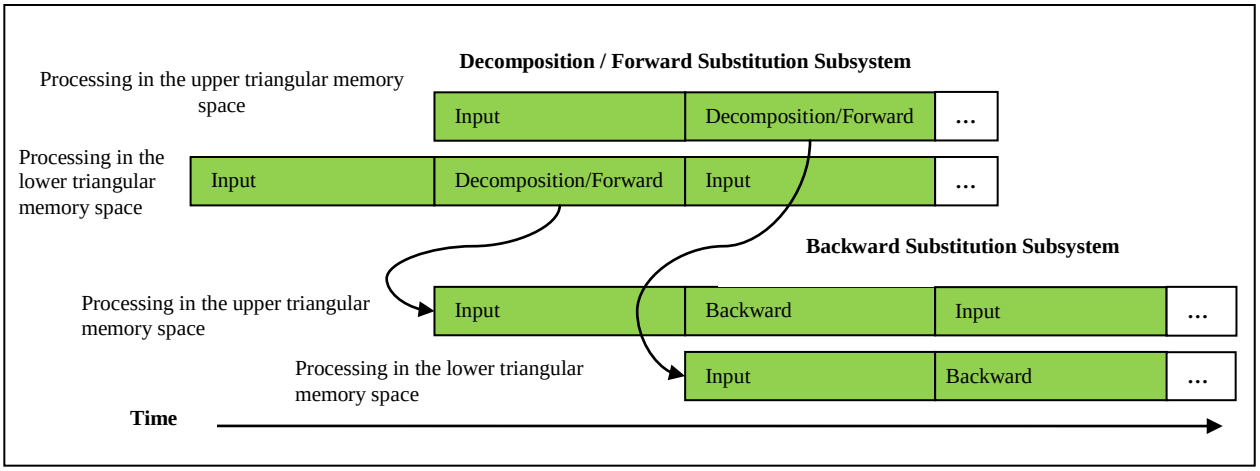

图2. Cholesky过程流水线和存储器重新使用

在FSM的前两个状态中，从每一行的对角元素开始，逐元素逐行产生**R**矩阵的元素。在FSM的减法状态中，对内核存储器的矩阵**A**进行递归更新，由部分计算的$\boldsymbol{u}_i$矢量进行替换。例如，在*k*列，通过从*k+1*至*n*的每一列中减去换算后的$\boldsymbol{u}_k$，更新了存储器中*k+1*至*n*的所有列。这一过程回归计算矢量$\boldsymbol{u}_{k+1}$至$\boldsymbol{u}_n$，对于硬件实现，是方程(6)更高效的版本。在FSM的第4个状态中，按列进行处理，每一新计算的$\boldsymbol{u}_k$与输入矢量**b**相乘，产生列矢量**d**的一个元素d_k。并没有明确的生成**Q**矩阵，而是生成其正交列$\boldsymbol{u}_k$，由FSM下一周期新矩阵相应的列$\boldsymbol{a}_k$将其覆写，用在FSM的后续阶段。图5显示了QR分解子系统的存储器组织和处理序列。

第一个子系统的输出是**R**矩阵和列矢量**d**。

R矩阵是上三角矩阵，如方程(3)和(4)所示，从左到右按行生成该矩阵。在乒乓方式中，采用了矩形存储器结构。由后向代换子系统处理上三角部分，分解子系统填充下三角部分，反之亦然。**d**列矢量附在**R**矩阵上，方式是按行从上到下产生。后向代换的输出是，线性方程**Ax** = **b**的解矢量**x**。

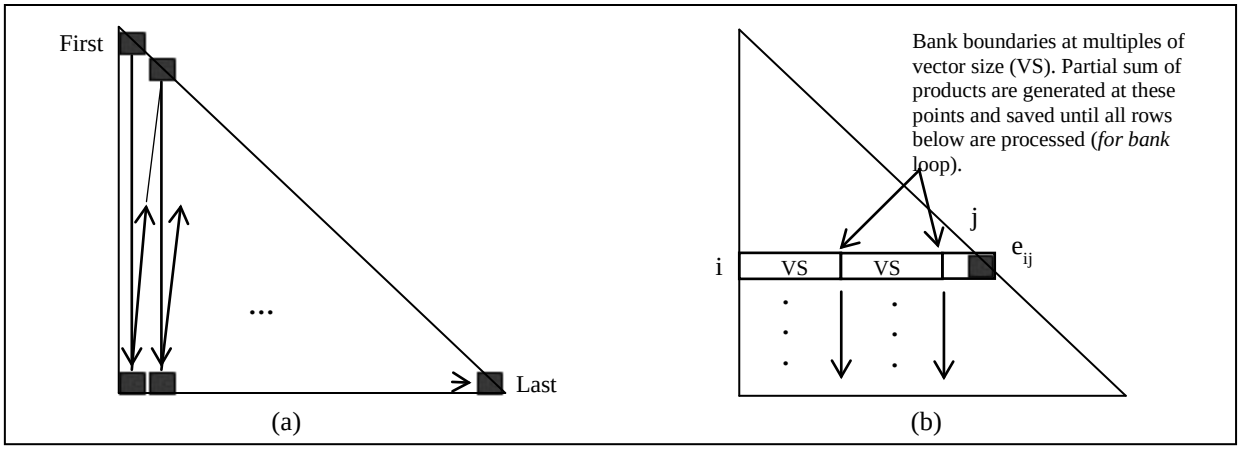

图2 (a) 处理序列 (b)对角元素e_{ij}的计算包括j=VS和j=2*VS的两个部分乘积求和，加上最后剩余的部分点乘。

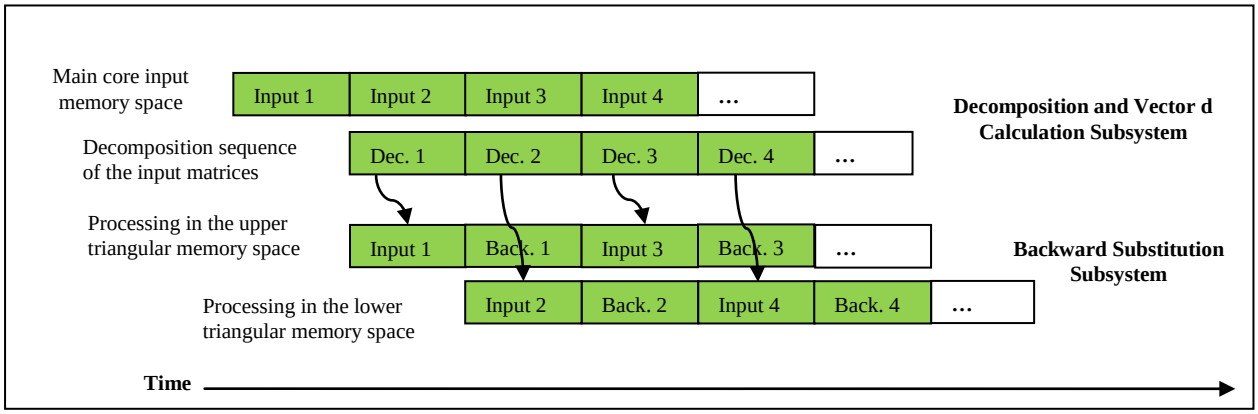

图4. QR过程流水线和存储器重新使用

与Cholesky求解器相似，以多通道格式实现QR求解器，以提高设计的利用率，减小延时，提高吞吐量。之所以能够提高吞吐量，主要是有效的减小了浮点累加器的延时。

3. 设计流程和工具链

评估方法

出于评估目的，Altera为BDTI提供了Cholesky和QR求解器，它是由DSP Builder高级模块库产生的。Altera为BDTI工程师提供了PC，安装了所需的Altera和MathWorks工具。BDTI工程师然后检查了Altera设计，在Simulink环境中对其进行仿真和综合。此外，在两个独立的FPGA器件上运行综合后的设计：Stratix V和Arria V FPGA。在这一过程中，BDTI评估了Altera设计流程，以及两个设计实例的性能。由于这一评估所使用的电路板不能为设计实例产生激励，因此，两个设计的每一个都含有一个激励生成模块，使用DSP Builder高级模块库中的模块来产生这些激励生成模块，并通过待测应用设计(DUT)进行编译。为减小激励模块对DUT性能和FPGA资源使用的影响，从一组更小的随机数据中，由激励模块随时产生矩阵**A**。MATLAB m文件脚本产生这一小组数据，装入到激励模块存储器，由设计进行编译。这一MATLAB脚本使用与DUT中激励模块相同的算法，产生双精度浮点格式参考数据，解矢量**x**作为参考来衡量Simulink模型以及在FPGA中运行的设计的误码性能。在Cholesky设计中，这一算法保证了矩阵**A**的厄米正定性，而在QR求解器设计中，它保证了产生良态**A**矩阵。

对于Cholesky求解器设计，实现了矢量、矩阵和通道大小的四种配置，在Stratix V FPGA上实现了三种配置，在Arria V器件上则是两种配置，两种器件之间还有一个公共配置。对于QR求解器设计，Stratix V FPGA实现了四种配置，还在Arria V器件上实现了这些

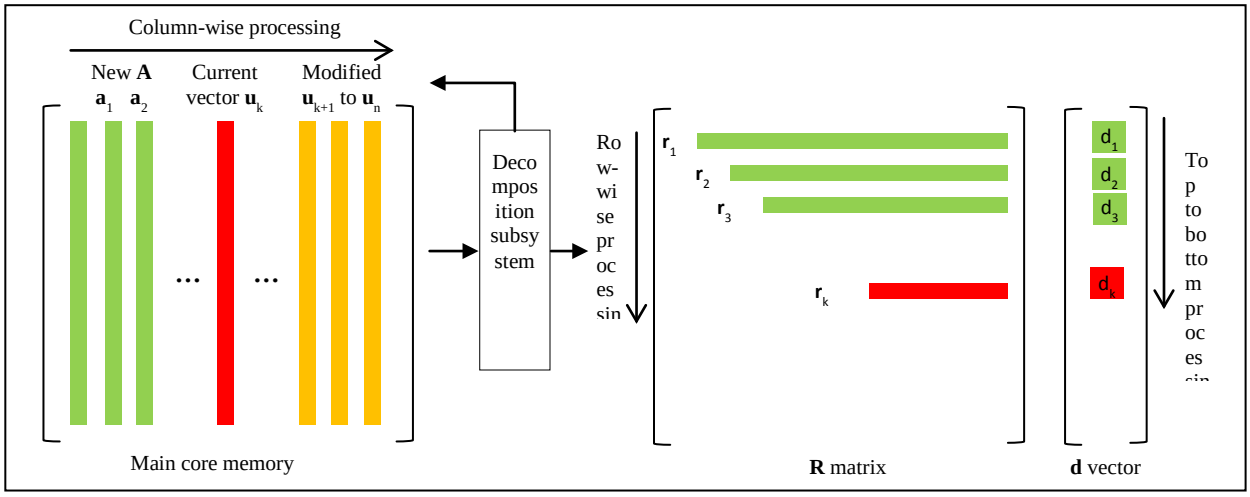

图5. QR分解子系统的存储器组织

配置中的两个。在FPGA资源利用率、可用时钟速率、吞吐量以及功能正确性上对FPGA的所有配置进行了评估。在Simulink环境中对时钟速率、器件选择以及速率等级等FPGA设计约束进行了设定。

在Simulink模型和硬件平台级对吞吐量和性能进行了评估。对两个设计的Simulink模型进行了仪表显示处理，以显示前向处理以及所有的前向和后向处理周期，Quartus II软件运行给出了每一配置所实现的F_{max}。然后将每一配置下载到硬件平台上，其工作频率被设置为F_{max}，开始进行处理。针对每一配置来采集每一求解器的解矢量**x**，与MATLAB生成的双精度浮点参考进行对比。

后仿真脚本计算Simulink IEEE 754单精度浮点输出和MATLAB产生的双精度浮点参考矢量之间的不同。相似的，脚本计算硬件仿真采集的输出和MATLAB产生的双精度浮点参考之间的不同。

工具链的评估

构建Simulink，需要MATLAB工作台。在Simulink环境中，评估的设计使用了Altera DSP Builder高级模块库的模块，它是从标准DSP Builder库中分离出的模块库。DSP Builder高级模块库适用于基于模块的DSP算法和数据通路的实现，与标准DSP Builder库相比，使用了高层抽象，包含了更通用、更基本的功能模块。DSP Builder高级模块库含有50多个常用三角函数、算术函数以及布尔函数，还有更复杂的快速傅里叶变换(FFT)和FIR滤波器构建模块。Quartus II软件v12.0中最显著的特点是增加了低延时平方根，嵌套*for*循环，以及可定制浮点累加器模块。在同一层次的数据通路结构上，不能混合使用标准模块库和DSP Builder高级模块库的单元；只有DSP Builder高级模块库的模块支持浮点编译器。标准模块库的模块并没有针对浮点处理进行优化。而且，虽然导入的手动编码HDL适用于标准模块库，由于工具不在HDL级进行优化，因此，它并不适用于DSP Builder高级模块库。一般而言，基于模块的设计输入方法非常适合DSP算法；但是，由于在模块库中缺少*case*或者*switch*等结构，因此，基于文本的方法更适用于需要大量控制功能以及涉及到状态机的设计。

采用DSP Builder高级模块库开始仿真，编译Simulink模型，为Altera Quartus II软件环境产生HDL代码和约束，为ModelSim环境构建测试台和脚本文件，仿真Simulink模型。在各种配置下，运行仿真所需要的时间从3分钟到28分钟不等，具体取决于3 GHz Intel Xeon W3550 PC的矩阵容量。Simulink仿真产生详细的资源利用率估算，不需要运行Quartus II软件编译，从而帮助设计人员迅速确定所需要的器件规模。当使用硬件开发套件时，用户必须提供基于电路板布板的引脚输出分配；用户可以在设计中重新使用Quartus II软件工程文件夹中*.qsf*文件所提供的引脚输出约束，或者使用Quartus II软件中的引脚规划器从头开始分配并管理引脚输出。

在模型上进行试验，以评估算法以及相应的HDL产生是否简单易行。在激励模块中修改矢量点乘大小、矩阵大小以及数据类型等输入参数，运行仿真。在所有情况下，几分钟就可以产生正确的RTL代码，仿真输出与MATLAB参考相匹配。

使用Quartus II设计软件对所有配置进行了综合，可以从Simulink环境中直接启动该软件。设计人员可以在*push-button*模式中使用Quartus II软件，采用默认或者由用户选择的优化参数，还可以使用设计空间管理器(DSE)工具。作为Quartus II软件的一部分，DSE在每一通路上使用不同的种子，自动运行多个布线通路。保存具有最佳时钟速率的布线。这是一个自动完成的过程，不需要用户干预，但是时间要比按键方法长很多。与设计规模有关，Quartus II软件按键方法运行时间从1小时到6.5小时不等。

DSP Builder高级模块库设计流程所采用的高层抽象支持高速算法空间研究，加速了仿真周期，从而缩短了实现最优设计的总时间。然而，这一优势并不是Simulink固有的，例如，与数据类型传播是Simulink所固有的特性并不一样。为实现基于模块的设计方法相对于手写RTL的设计空间管理优势，设计人员在建立Simulink模型时还需要其他的步骤。特别是，模型必须是结构化，才能实现参数驱动的算法空间管理。对于本文分析的实例，实现的模型支持对不同矩阵大小、矢量长度和(对于Cholesky求解器的情况)并行通道数进行试验。通过这一灵活的方法建立了模

型后，可以改变这些参数来估算各种设计配置的性能和资源使用情况。还需要理解一些硬件设计，以实现良好的吞吐率和资源利用率，正如本文第2部分讨论的浮点累加器模块实例所示。

DSP Builder高级模块库设计流程培训课程包括Altera的4小时课程，以及大约10小时的在线教程和演示。此外，BDTI花费了大约90小时，在手动编码基础上来研究工具和所有模型。加速实现工具链的时间和投入取决于设计人员的技能和背景知识。对Simulink基于模块的设计以及FPGA硬件设计非常熟悉的工程师会发现DSP Builder高级模块库方法效率很高而且使用方便。对于不熟悉MATLAB和Simulink的FPGA设计人员，在高层抽象上进行设计可能需要从新的角度进行思考，一开始就会遇到困难，花费较长的学习时间。掌握了这一方法后，与HDL方法相比，设计人员能够很快的加速实现设计。设计人员可以把精力集中在算法实现上，而不用担心流水线等详细的硬件设计。由于大部分功能仿真和验证都在Simulink环境中完成了，因此，显著缩短了设计和验证时间。可以在ModelSim软件中运行Simulink编译的RTL输出，进行全面功能仿真。

具有系统级设计背景的工程师，即使不太熟悉硬件设计，也不会花费太长的学习时间。工具链虽然在Simulink环境中集成了硬件编译、综合、布线和自动脚本产生，抽象出很多复杂设计概念，例如数据流水线和信号矢量化，但是，还是需要了解一些硬件设计知识才能完成设计。

4. 性能结果

这一部分介绍BDTI对Altera Cholesky和QR求解器浮点实现实例的独立评估。

所有的设计都使用了Altera的DSP Builder高级模块库v12.0，此模块库采用MathWorks R2011b，并用Quartus II设计软件v12.0 SP1进行构建。使用ModelSim 10.1完成RTL仿真。这些设计是针对两种Altera 28-nm FPGA进行构建：高端中等容量Stratix V 5SGSMD5K2F40C2器件和中端Arria V 5AGTFD7K3F40I3N器件。在这一分析中使用的Stratix V FPGA具有345.2K ALUT、1,590个27×27位精度可调乘法器，以及2,014个M20K存储器模块。Arria V FPGA具有380.4K ALUT、1,156个27×27位精度可调乘法器，以及2,414个M10K存储器模块。用于RTL评估的硬件平台是Stratix V版的DSP开发套件，以及

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)

表1 资源使用率和时钟速率

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

表2 性能结果

Arria V FPGA开发套件。ModelSim软件只用在一种配置中，以评估在Simulink环境中使用这个工具的方便程度。

针对这两个设计并结合这两个器件，总共仿真和建造了11个案例。对每一案例，记录了资源使用情况、性能以及精确度结果。

表1列出了每一配置的Cholesky和QR求解器的资源使用情况以及时钟速率。Cholesky求解器设计提供最大矩阵容量参数。在实际运行时，可用其容量小于最大设计容量的矩阵。对于表1列出的资源使用结果，每个配置综合是在最大矩阵容量参数等于待评估的矩阵容量的条件下进行的。以获得所报告的矩阵占用的实际资源。此结果不包含激励模块所使用的资源。值得一提的是，本文中评估的所有配置都没有使得FPGA满负荷运转。为能够在Quartus II软件中以合理的综合和布局布线时间来获得最佳F_{max}，每一设计都采用了同样的预设优化参数，以提高速率。我们在Cholesky设计实例中选择6/90×90/45配置，运行Quartus II软件设计空间管理器(DSE)，评估与按键模式相比速度提高了多少，缩短了多少时间。在这一案例中，速度提高了12.5%，而Quartus II软件运行时间则从2小时增加到7.5小时，才能完成设计综合。

FPGA资源使用情况与待评估设计的期望相一致。对存储器占用最多的是矩阵存储，它与矩阵容量和多通道设计的通道数量成正比。DSP模块的使用率随着矢量大小而线性增加。矢量乘法器的每一个27位×27位复数浮点乘法操作需要4个精度可调DSP模块。对于长度为60个复数浮点值的矢量，矢量点乘引擎需要240个DSP模块。

表2显示了所有配置时的Cholesky和QR求解器的性能。表1报告的F_{max}给出了每一情况下的性能。F_{max}除以求解器前向子系统执行所占用的周期，可得到吞吐量。由于并行执行后向置换子系统，其延时又小于前向子系统，总吞吐量不会受前者的影响。对于多通道Cholesky求解器吞吐量，是这一结果与并行处理的通道数相乘(表2中通道大小参数)。将前向和后向子系统执行所需要的总周期除以F_{max}，可得到每一情况下的总延时。相对于矩阵大小来选择矢量长度是一种折中的方法，与应用有关。如果矢量长度远远小于矩阵大小，那么，可以提高设计的资源占用效率，但代价是延时，如所示的具有不同矢量长度的QR求解器在200×100矩阵大小下的配置。

相对于BDTI在以前文章中分析的单通道设计，多通道Cholesky设计有很大的改进。在单通道实现中，在算法中重新安排处理顺序，从而隐藏了部分延时，例如，浮点累加器中的延时。正如参考[1]所报告的，单通道实现的效率主要取决于矩阵和矢量的大小。看一下表2的吞吐量一栏，多通道实现在处理效率上有很大的优势，特别是在矩阵和矢量

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

表3 与MATLAB双精度浮点参考相比，Simulink模型和综合RTL的误码性能

都较小的的情况下。多通道处理完全隐藏了本文第2部分描述的实现延时，从而提高了吞吐量。对于大小一定的矩阵和矢量，一个多通道实现要比相应的单通道实现有更高的峰值吞吐量。

表中的最后一列显示了每一配置的每秒以10^9 (GFLOPS)为单位的实数浮点操作量。每一求解器所需要的操作次数取决于所使用的分解算法。对于这一评估，所报告的数据来自Cholesky求解器和QR求解器算法的实际实现，采用了两片FPGA浮点复数格式。对于Cholesky求解器，实数浮点操作数与二阶项$4n^3/3 + 12n^2$接近，而对于QR求解器，使用了$8mn^2 + 6.5n^2 + mn$次浮点操作。

表3显示了采用单精度浮点运算时用Simulink仿真和运行在硬件开发板上的设计实现的Cholesky和QR求解器的误码性能。通过对每一Simulink和硬件平台仿真的输出与MATLAB用双精度浮点产生的解矢量**x**的比较来计算误码。对于多通道Cholesky求解器的情况，为简洁起见，只报告了一个随机选择的通道的误码性能。虽然误码性能与输入数据相关，平均而言，RTL实现比较适合采用融合数据通路方法，与表3中第(4)列和第(5)列的Frobenius范数相比，实现的精度在统计上等于或者高于标准IEEE 754单精度实现。我们使用Frobenius范数来衡量得到的矢量的总误码大小；计算如下：

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

其中，N是矢量长度，**e**是所观察的**x**及其MATLAB生产的最佳参考之间的差矢量，i是矢量**e**中的元素指数。最大归一化误码由下式给出：

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

5. 结论

在本文中，我们评估了在FPGA中实现浮点DSP算法的新方法，它使用了Altera的DSP Builder高级模块库设计流程。这一设计流程采用了Altera DSP Builder高级模块库、Altera的Quartus II软件工具链、ModelSim仿真器，以及MathWorks的MATLAB和Simulink。通过这一方法，设计人员在Simulink环境中，能够在算法行为级进行工作。在Simulink环境中，工具链结合并集成了算法模型和仿真、RTL产生、综合、布局布线以及设计验证级等。通过功能集成，在算法级和FPGA级实现了快速开发和设计空间管理，最终减少了在总体设计上的投入。在高层对算法进行建模并调试后，很容易面向Altera FPGA对设计进行综合。

出于评估的目的，这些设计实例是在Simulink中用Altera DSP Builder高级模块库建立的单精度复数数据IEEE 754浮点Cholesky和QR求解器。我们评估的最大的设计实例是

QR求解器，应用于大小为400×400的复数浮点矩阵和长度为100的矢量。运行在203 MHz，这一实例处理162 GFLOPS。表2中所报告的GFLOPS数据来自Cholesky求解器和QR求解器算法以浮点复数格式的实际实现，此实现采用了两片FPGA。为能够与其他相抗衡的计算平台进行有效对比，应在这些平台上实现相同的算法，并测量其性能。文中所报告的性能结果是在使用了Altera DSP Builder高级模块库工具流而得到的，没有进行手动优化或者平面规划。从Simulink中基于模块的高层设计开始，所用的工具链自动形成处理流水线，生成RTL代码，并对设计进行综合，实现了可行的速率和资源利用。Altera浮点设计流程通过在一个单一的平台上精简所用的工具来简化在FPGA中实现复数浮点DSP算法的过程。采用融合数据通路方法，与以前相比，实现的复数浮点数据通路性能更好，效率更高。

对于本文中分析的新方法，在使用DSP Builder高级模块库时需要花一些时间来学习和掌握。对于不熟悉MATLAB和Simulink的设计人员而言尤其如此。对于传统硬件设计人员，在开始使用基于模块的设计输入方法时会遇到一些困难。此外，为充分利用基于模块的设计方法相对于手写RTL的优势，设计人员在建立Simulink模型时还需要其他的步骤。例如，为使实验能用于不同大小的矩阵和矢量，正如本文中两个设计实例所作的，Simulink模型应是结构化的，使其可用参数驱动的设计来探索各种设计配置。

目前，使用DSP Builder高级模块库的设计人员只能用模块库提供的元素来求得最佳性能。标准DSP Builder模块库的元素没有经过浮点编译器的优化，所以不能在同一层次级上与高级模块库混合使用。手工编码的HDL模块可导入到标准模块库中。此外，DSP Builder高级模块库适用于DSP实现，对于需要大量控制和状态机的设计，其使用可能会受到限制。

下一版本的DSP Builder高级模块库预计于2012年底发布，会包括浮点扩展。设计人员将不再受限于仅有的两种标准IEEE 754单精度和双精度格式，而是可以选择从16位到64位(指数加尾数)总计7种不同的精度范围。使用DSP Builder高级模块库中新的*增强精度支持*模块，设计人员可以选择最适合其应用的数据宽度。

6. 参考

[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.