On Matrix Inversion for LTE MIMO Applications Using Texas Instruments Floating Point DSP

Mingjian Yan
WBI Engineering, Texas Instruments
Germantown, Maryland, USA
m-yan2@ti.com

Brighton Feng and Tommy Song
China Communication Infrastructure Application
Semiconductor Group, Texas Instruments
Shanghai, P. R. China
bf@ti.com; tommy-song@ti.com

***Abstract*— Multiple-Input Multiple-Output (MIMO) is one of the key technologies for the current and future broadband wireless services. Matrix inversion is the most costly computational module within the Minimum Mean-Square Error (MMSE) based MIMO receiver. For LTE Release 8, the order of the matrix to be inverted for an MMSE receiver is 2×2 for 2-stream MIMO. Going forward to LTE release 10, 4-stream MIMO can potentially double the throughput, but the matrix to be inverted will grow to 4×4, which is computationally intensive. In this paper, we will exam several software implementations of 4×4 matrix inversion. We will demonstrate how we can significantly reduce the cost (both in terms of cycle counts and development time) while maintaining enough output precision to meet performance requirements by using the floating-point feature of Texas Instruments' (TI) new multicore System-on-a-Chip (SoC) architecture.**

Keywords-component; 4×4 matrix inversion; Floating-point DSP; MIMO; LTE; Texas Instruments

I. INTRODUCTION

MIMO is one of the key technologies in broadband wireless services to improve bandwidth utilization efficiency. Figure 1 shows the example of LTE uplink 2-stream multi-user MIMO.

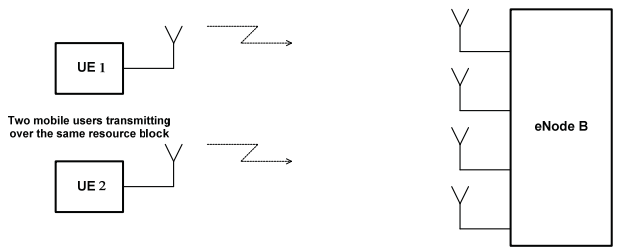


Figure 1. Uplink multi user MIMO

In the LTE application, an MMSE-based algorithm is most common and the solution can be written as follows:

$$B = \left(H^H R_{ZZ}^{-1} H + \frac{1}{\sigma_S^2} I_{N_T} \right) \quad (1)$$

$$\hat{D}_{MMSE} = B^{-1} H^H R_{ZZ}^{-1} Y \quad (2)$$

978-1-4244-5899-8/10/$26.00 ©2010 IEEE

Where:

$\hat{D}_{MMSE}$ is the output of the equalizer;

H is the channel matrix with size *RxAnts* by *TxStreams*, *RxAnts* is the number of receive antennas, and *TxStreams* is the number of transmit streams;

R_{ZZ}^{-1} is the inverse of the noise covariance matrix with size *RxAnts* by *RxAnts*;

σ_S^2 is the variance of the input signal;

Y is the input received signal to the equalizer, with size *RxAnts* by 1.

Here both matrices that need to be inverted are Hermitian matrices. *B* is a 2×2 matrix for 2-stream MIMO and a 4×4 matrix for 4-stream MIMO.

In this paper, we will focus on 4×4 matrix inversion and evaluate three matrix inversion algorithms: Cholesky decomposition based matrix inversion, a blockwise method, and the cofactor method. Using a fixed-point implementation of the Cholesky decomposition-based matrix inversion as the reference, we will look at the cycle count cost of the blockwise and cofactor methods in floating-point using TI's new multicore SOC, as well as the error performances of each algorithm.

II. NEW MULTICORE SYSTEM-ON-CHIP ARCHITECTURE FOR WIRELESS APPLICATIONS FROM TI

In February 2010, TI announced a new SoC architecture based on its multicore DSPs that integrates fixed and floating point capabilities in the industry's highest performing CPU at up to 1.2GHz clock speed.

A. Floating-point computational capability of the new TI SOC architecture

The new TI SOC architecture offers the industry highest floating-point computational capability. In summary:

1. The floating-point engine runs at the same clock speed as the fixed-point engine.
2. The floating-point engine has the same single precision floating-point operations efficiency as the fixed-point 32-bit operation efficiency, which is:

- 8 real multiplications per clock cycle;
- 2 complex multiplications per clock cycle;
- 8 real additions/subtractions per clock cycle;
- 2 real inverse or 2 square root inverse operations per clock cycle (8-bit mantissa precision, Newton-Raphson interpolation is needed for higher precision). These are fully pipelineable operations, while fixed-point division often involves in iterative subtraction, or table lookups;
- 8 conversions per clock cycle to convert between 16-bit/32-bit integer and single-precision floating-point numbers. This makes mixed floating/fixed-point coding more efficient.

3. In addition to the single-precision operations, the floating-point engine also has offers fast pipelinable double precision floating-point operations

B. *Floating-point advantages in wireless application*

For wireless applications, especially in the area of receiver algorithms, floating-point brings some key advantages over fixed-point implementations:

- Most of the time, simulation of algorithms starts on floating-point (MATLAB, C, or C++) platforms. While there is rarely a need to re-tune/test precision of the algorithm after implementation on single-precision floating-point for wireless application, fixed-point implementation can run into many rounds of adjustment and retest within the link/system level simulator to ensure the performance under different scaling, rounding, and data bit-width precisions combinations. In addition, these fixed-point scaling, rounding, and data bit-width precisions may not be run efficiently on a particular fixed-point DSP architecture and could cost more cycles to execute.
- Very often for an advanced algorithm, fixed-point implementation can run into the conflict of precision vs dynamic range. The developer has to either increase the bit-width of the value which leads to much more cycles used, or scale dynamic range along the calculation which also cost more MIPS, or sacrifice the performance. Single-precision floating point can easily solve this problem with much wider dynamic range than 32-bit integer operations.

All these advantages can result in fast prototyping and productization, and possibly lower cost of wireless algorithm implementations. In addition, combining with an advanced compiler, floating-point coding on TI's new SOC is mostly native C with a few Single Instruction Multiple Data (SIMD) instructions needed for manual optimization with C function-like intrinsic calls because of no need for the variations of instructions for rounding, saturation, bit-width, and extra shifts as fixed-point arithmetic does. This also reduces the time for prototyping and productization.

III. MATRIX INVERSION ALGORITHMS AND IMPLEMENTATION

To invert a matrix of size bigger than 2×2 in fixed-point implementation, it is often suggested to avoid simple analytic types of approach and use more stable decomposition-based algorithms such as Cholesky, QRD, LUD or LDL decomposition ([1], [2]). They easily require 1000 cycles or more to invert a 4×4 matrix. We will use the Cholesky decomposition-based fixed-point inverse in this paper as the reference for fixed-point implementation.

Having floating-point precision and dynamic range in mind, we will compare the Cholesky decomposition method to two algorithms that are often avoided in fixed point because of the poor stability, but work well in floating point and require fewer cycles: a cofactor method and a blockwise method.

All three methods are implemented and optimized in C with intrinsics on TI new SOC architecture. For the fixed-point implementation, 32-bit intermediate results are used. For the floating-point methods, all intermediate values are kept in single-precision floating-point. More implementation details can be found in the following subsections.

A. *Cofactor method for matrix inversion*

Use 4×4 matrix as an example, the inversion of matrix A can be written as:

$$A^{-1} = \frac{1}{|A|}\left(C^H\right)_{ij} = \frac{1}{|A|}\left(C\right)_{ji} = \frac{1}{|A|}\begin{bmatrix} C_{00} & C_{10} & C_{20} & C_{30} \\ C_{01} & C_{11} & C_{21} & C_{31} \\ C_{02} & C_{12} & C_{22} & C_{32} \\ C_{03} & C_{13} & C_{23} & C_{33} \end{bmatrix} \quad (3)$$

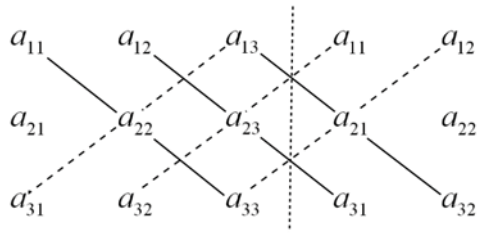
Where $|A|$ is the determinant of A, C_{ij} is the matrix of cofactors, and C^H represents the matrix conjugate transpose.

$$C_{ij} = (-1)^{i+j} M_{ij} \quad (4)$$

Where M_{ij} is the $(i,j)^{th}$ minor of A, and is defined to be the determinant of the submatrix obtained by removing from A its *i-th* row and *j-th* column. For example, M_{11} of 4x4 matrix is generated as below.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \xrightarrow{M_{11}} \begin{vmatrix} a_{00} & a_{02} & a_{03} \\ a_{20} & a_{22} & a_{23} \\ a_{30} & a_{32} & a_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} a_{00} & a_{02} & a_{03} \\ a_{20} & a_{22} & a_{23} \\ a_{30} & a_{32} & a_{33} \end{vmatrix} \quad (5)$$

The determinant of a 3x3 matrix can be calculated by its diagonals like below. The sum of the products of three diagonal north-west to south-east lines of matrix elements, minus the sum of the products of three diagonal south-west to north-east lines of elements when the copies of the first two columns of the matrix are written beside it as below:



In our floating-point implementation, one division is used for calculation of $1/|A|$. There is no requirement on pre-scaling the input matrix, or internal scaling to preserve the dynamic range. We also fully unrolled the calculation and only the lower half of the matrix is calculated to take advantage of the Hermitian property of the input matrix. All calculations are done using single-precision floating-point.

B. *Blockwise method for matrix inversion*

Matrices can also be inverted blockwise by using the following analytic inversion formula:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D-CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D-CA^{-1}B)^{-1} \\ -(D-CA^{-1}B)^{-1}CA^{-1} & (D-CA^{-1}B)^{-1} \end{bmatrix} \quad (6)$$

where *A, B, C* and *D* are matrix sub-blocks of arbitrary size. (*A* and *D* must be square, so that they can be inverted). In this example, the 4×4 matrix is equally split into four, 2×2 matrices.

In the floating-point implementation, two divisions are used to calculate the inverse of two, 2×2 matrices. All calculations are done using single-precision floating-point.

C. *Cholesky decomposition based matrix inversion*

If $\underline{A}$ is Hermitian and positive definite, then *A* can be decomposed as

$$A = LL^H = \begin{bmatrix} l_{00} & & & \\ l_{10} & l_{11} & & \\ \cdots & \cdots & \cdots & \\ l_{(n-1)0} & l_{(n-1)1} & \cdots & l_{(n-1)(n-1)} \end{bmatrix} \times \begin{bmatrix} \overline{l_{00}} & \overline{l_{10}} & \cdots & \overline{l_{(n-1)0}} \\ & \overline{L_{11}} & \cdots & \overline{l_{(n-1)1}} \\ & & \cdots & \cdots \\ & & & \overline{l_{(n-1)(n-1)}} \end{bmatrix} \quad (7)$$

Where *L* is a lower triangular matrix with strictly positive diagonal entries, and L^H denotes the conjugate transpose of L. This is the Cholesky decomposition.

The Cholesky decomposition is implemented step by step as shown below

$$A = \begin{bmatrix} a & V^h \\ V & B \end{bmatrix} = \begin{bmatrix} \sqrt{a} & 0 \\ \frac{V}{\sqrt{a}} & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & B - \frac{VV^H}{a} \end{bmatrix} \times \begin{bmatrix} \sqrt{a} & \frac{V^H}{\sqrt{a}} \\ 0 & 1 \end{bmatrix} = L_0 A_0 L_0^H \quad (8)$$

Where *a* is the matrix element in the first row and first column, *V* is an (*n*-1)×1 matrix, *B* is (*n*-1) × (*n*-1) matrix. L_0 is the first column of the solution *L*. A_0 is still a Hermitian matrix, and can be further decomposed in the same way and get the next column of *L*. This continues until the size of the last A_i is one, then the last element of *L* is simply $\sqrt{a}$, and the Cholesky decomposition has been accomplished.

Once the *L* is calculated, instead of doing matrix inversion of *L* to get an MMSE solution in Equation (2), we can use a lower and upper equation solver to calculate $\hat{D}_{MMSE}$:

$$\begin{aligned} &\hat{D}_{MMSE} = B^{-1}H^H R_{ZZ}^{-1}Y \\ &B\hat{D}_{MMSE} = \hat{Y}, \quad where \quad \hat{Y} = H^H R_{ZZ}^{-1}Y \\ &LL^H\hat{D}_{MMSE} = \hat{Y} \end{aligned} \quad (9)$$

In our fixed-point implementation, the $1/\sqrt{a}$ (for Cholesky decomposition) and $1/a$ (for equation solvers) are calculated using table look up and Newton Raphson interpolation to achieve 32-bit output precision. All intermediate values are 32-bit real and 32-bit imaginary precision.

IV. PERFORMANCE TEST SETUP

This section describes the setups for performance testing of the matrix inversion algorithms we have selected to evaluate.

For our 4×4 matrix inversion performance test study we use a simplified MMSE receiver as the target. Here we assume that the noise covariance matrix R_{ZZ}^{-1} in equation (1) is a diagonal matrix with equal diagonal elements, meaning that we assume white noise and equal noise variance for all receive antennas. The equation can be further simplified to

$$B = \left(H^H H + \frac{\sigma_n^2}{\sigma_S^2} I_{N_T} \right) \quad (10)$$

$$\hat{D}_{MMSE} = B^{-1}H^H Y \quad (11)$$

Figure 2 shows the block diagram of the test bench for our 4×4 matrix performance study.

In this test bench, we inject a very small input noise to limit the matrix condition to under 2×10^3.

The matrix condition is defined as the maximum eigenvalue divided by the minimum eigenvalue for the matrix. It indicates the "invertibility" of a matrix. The bigger the condition number, the more singular the matrix is. In this test the matrix condition is calculated using the MATLAB function *cond()*.

The error statistics are defined as the output signal to error ratio (SER):

$$SER_x = -20 * \log_{10}(mean(\frac{|\hat{D}Mat - \hat{D}_x|}{|\hat{D}Mat|})) \quad (12)$$

This calculates the log ratio of the MATLAB equalizer output of $\hat{D}_{MMSE}$ as the signal, over the error between MATLAB and corresponding inversion method for $\hat{D}_{MMSE}$.

Statistics are collected from 10,000 runs of the test.

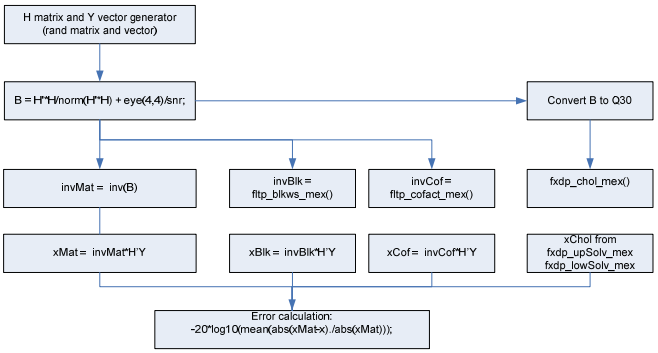


Figure 2. Block diagram of 4×4 matrix inversion performance study

V. BENCHMARK SUMMARIES AND PERFORMANCE RESULTS

This section summarizes the benchmark from TI's SOC and performance results from the 4×4 matrix inversion algorithms we tested.

TABLE I. TI SOC 4×4 MATRIX INVERSION BENCHMARK SUMMARY

Algorithms	4×4 inversion benchmarks (cycles)
Floating-point C blockwise + $B^{-1}Y$	188 + 8 = 196
Floating-point C co-factor + $B^{-1}Y$	265 + 8 = 273
Fixed-point C Cholesky + 2 EQ solver	668 + 80 = 748

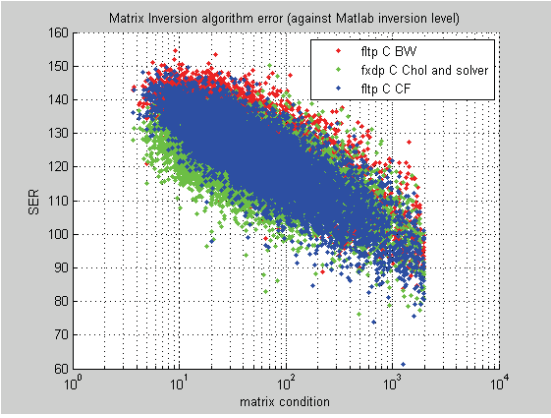


Figure 3. Performance scatter plot for 4×4 matrix inversion algorithms

As we can see from Table 1 and Figure 3, while all three method we evaluated have nearly the same performance in terms of output SER, floating-point blockwise has the lowest cycle counts, which is about a factor of 4 improvement over the fixed-point Cholesky decomposition costs. The floating-point cofactor method has about a 2.5 times improvement factor over the cycle counts of fixed-point Cholesky decomposition. In addition, all floating-point implementations are mostly serialized code with limited parallelization. In the equalization application, we can calculate the inversion of matrices for multiple frequency subcarriers. Putting these floating-point matrix inversions into a loop and using the compiler to better pipeline the loop can result in higher parallelization and further reduce the cost of the inversions.

This shows that with floating-point precision and dynamic range, we can achieve the same performance using much efficient algorithms such as blockwise and cofactor methods, while fixed-point implementation has to use more complex but more computational stable algorithm such as Cholesky decomposition to avoid false results.

VI. CONCLUSION

In this paper, we evaluated three methods for 4×4 matrix inversion for LTE MIMO: Cholesky decomposition in fixed-point, and blockwise and cofactor methods in floating-point. We implemented all three methods using TI's new baseband SOC architecture and measured cycle counts. We also developed performance test benches to evaluate the numerical performance of these methods. As shown in Section V, floating-point blockwise and cofactor method save about 2.7 to 3.8 times the fixed-point Cholesky decomposition based inversion in terms of cycle counts, while maintaining very close output signal to error ratio performance. Floating point DSP saves MIPS (cost) and saves time (is easier to develop) for the same or better performance than older fixed-point as shown in the study of 4x4 matrix inversion.

REFERENCES

[1] N.J. Higham, Accuracy and Stability of Numerical Algorithms 2nd Edition, 2002

[2] Di Wu, Johan Eilert and Dake Liu, Dandan Wang, Naofal Al-Dhahir and Hlaing Minn, "Fast Complex Valued Matrix Inversion for Multi-User STBC-MIMO Decoding" VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium, pp. 325 - 330, March 2007