A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA

Girish Sharma a, Abhishek Agarwala b, Baidurya Bhattacharya a,⇑

a Department of Civil Engineering, Indian Institute of Technology, Kharagpur 721302, India
b Archayne Labs, Gurgaon 122001, India

The ability to invert large matrices quickly and accurately determines the effectiveness of a computational tool. Current literature suggests that time complexity of matrix inversion is 2 or higher. This paper redesigns the Gauss Jordan algorithm for matrix inversion on a CUDA platform to exploit the large scale parallelization feature of a massively multithreaded GPU. The algorithm is tested for various types of matrices and the performance metrics are studied and compared with CPU based parallel methods. We show that the time complexity of matrix inversion scales as $n$ as long as $n^2$ threads can be supported by the GPU.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Matrix inversion is an essential step in a wide range of numerical problems – starting with solving linear equations [1–4], structural analyses using finite element method [5–7], 3D rendering [8], digital filtering [9], image filtering [10,11] and image processing [12] – and constitutes an indispensable component in almost all mathematical/statistical software suites. Some of the common available algorithms for computing the inverse of a matrix are Strassen [13], Strassen-Newton [14], Gaussian elimination [15], Gauss–Jordan [15], Coppersmith and Winograd [16], LUP Decomposition [17], Cholesky decomposition [18], QR decomposition [19], RRQR factorization [20], Monte Carlo Methods for inverse [21,22], etc.

Until the late 1960s matrix inversion was believed to require a cubic number of operations, as the fastest algorithm known was Gaussian elimination method, or rather Gauss Jordan [15] method which runs in $O(n^3)$ time where $n$ is the size of the matrix. In 1969, Strassen [13] excited the research community by giving the first sub cubic time algorithm for matrix multiplication, running in $O(n^{2.808})$ time. This also reduced the time complexity, $w$, of Matrix Inversion using Strassen Multiplication to $O(n^{2.808})$ time. This discovery triggered a long line of research that gradually reduced the time complexity $w$ over time. In 1978, Pan [23] presented a method that proved $w < 2.796$ and the next year, Bini et al. [24] introduced the notion of border rank and obtained $w < 2.78$. Schönhage [25] generalized this notion in 1981, proving $w < 2.548$. In the same paper, combining his work with ideas by Pan [23], he also showed $w < 2.522$. The following year, Romani [26] found that $w < 2.517$. The first result to break 2.5 was by Coppersmith and Winograd [16] who obtained $w < 2.496$. In 1988, Strassen introduced his laser method [27] which was a novel approach for matrix multiplication, and thus decreased the bound to $w < 2.479$. Two years later, Coppersmith and Winograd [28] combined Strassen’s technique with a novel form of analysis based on large sets avoiding arithmetic progressions and obtained the famous bound of $w < 2.376$ which has remained unchanged for more than 22 years (a very recent unpublished work [29] claims to have brought down the limit to $w < 2.3727$). While most activity focuses on trying to reduce the exponent $w$, both Coppersmith and Winograd [28] and Cohn et al. [30] presented conjectures which if true would imply $w = 2$, but never less than that.

Inversion methods for specific types of matrices sometimes with no set time complexity, like Monte Carlo Methods [21,22] for inverting Hermitian matrix and positive definite matrix, a fast algorithm [31] for the inversion of general Toeplitz matrices and various matrix decomposition methods [16–19] also exist.

Recent developments in parallel architecture and its use in computation have brought about the prospect of massively parallel systems capable of reducing running times of codes below the limit of $w = 2$. The amount of performance boost of course depends largely upon the scope of parallelization that the algorithm provides. Owing to its unique architecture, the graphics processing unit (GPU) enables massive parallelization unlike anything possible on a CPU based network, as described later. The Gauss Jordan method is...
one of the oldest methods for matrix inversion. It is straightforward and robust and is particularly suitable for massive parallelization unlike many of the more advanced methods. Nevertheless, the available literature either on the scope of parallelization of the Gauss Jordan algorithm or its optimization appear rather insufficient. This paper tailors and implements the Gauss–Jordan algorithm for matrix inversion on a CUDA (Compute Unified Device Architecture [32]) based GPU platform and studies the performance metrics of the algorithm.

2. Parallelization and challenges

The Strassen approach [14,33] for matrix inversion reduces the problem of inverting an \( n \times n \) matrix into seven \( n/2 \times n/2 \) multiplications and inversion of two \( n/2 \times n/2 \) matrices, each of which then can be solved recursively. The inverse is given as:

\[
A^{-1} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{-1} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}
\]

(1)

The four partitions of \( C \) may be computed in the following steps:

1. \( P_1 = A_{11} \)
2. \( P_2 = A_{21} \times P_1 \)
3. \( P_3 = P_1 \times A_{12} \)
4. \( P_4 = A_{12} \times P_2 \)
5. \( P_5 = P_2 - A_{22} \)
6. \( P_6 = P_5^{-1} \)
7. \( C_{12} = P_3 + P_6 \)
8. \( C_{21} = P_5 \times P_2 \)
9. \( C_{11} = P_1 - P_3 \times C_{21} \)
10. \( C_{22} = -P_0 \)

For large \( n \), the individual matrices will be large enough to provide scope for parallelization of the matrix multiplication steps (steps 2, 3, 4, 7, 8 and 9) while the inverse calculation steps (steps 1 and 6) can be performed recursively. It is this inherent recursive nature of the algorithm that reduces the scope of large scale parallelization.

An additional issue faced by the Strassen approach is that the accuracy of the resultant inverse depends highly on the quarter matrix chosen in step 1, and in particular, the determinant between \( A_{11} \) and \( A_{22} \) is chosen. This is because in each level of recursion in steps 1 and 6, the inverse of the matrix involved. Existing work on the parallelization of the Gauss Jordan algorithm (e.g. [33]) has been limited in the scope for parallelization mainly due to the hardware limitations of the time. This paper redesigns the Gauss Jordan method so as to make full use of the massive parallelization feature of a modern GPU.

The standard Gauss Jordan method for computation of inverse of a matrix \( A \) of size \( n \) starts by augmenting the matrix with the identity matrix of size \( n \):

\[ [C] = [A|I] \]

(3)

Then, performing elementary row transformations on matrix \( C \), the left half of \( C \) is transformed by column into the unit matrix. This step is broken down into 2 steps per column of the left half matrix, the first of which is to convert element \( a_{ij} \) into 1 by the transformation:

\[ R_i \leftarrow R_i/a_{ii} \]

(4)

If \( a_{ii} \) is zero, then any non-zero row is added to the \( i \)th row before applying Eq. (4). The second step is to reduce all the other elements of the \( j \)th column to zero by the following transformation of every row except for the \( j \)th one:

\[ R_i \leftarrow R_i - R_j \times a_{ij} \]

(5)

After transforming the first column, the matrix reduces to:

\[
[C'] = \begin{bmatrix}
1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & \cdots & 1/a_{11} \\
0 & a_{22} - a_{21} \times a_{12}/a_{11} & a_{23} - a_{21} \times a_{13}/a_{11} & \cdots & \cdots & -a_{21}/a_{11} \\
0 & a_{32} - a_{31} \times a_{12}/a_{11} & a_{33} - a_{31} \times a_{13}/a_{11} & \cdots & \cdots & -a_{31}/a_{11} \\
\vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\
0 & a_{n2} - a_{n1} \times a_{12}/a_{11} & a_{n3} - a_{n1} \times a_{13}/a_{11} & \cdots & \cdots & -a_{n1}/a_{11}
\end{bmatrix}
\]

(6)

And following these two steps for each column sequentially, the left half of \( C' \) becomes the unit matrix while the right half becomes the desired inverse of \( A \):

\[ [C] = I|A^{-1} \]

(7)

Thus, Step 1 of the original Gauss Jordan algorithm (which converts \( j \)th element of \( j \)th column to 1, Eq. (4)) involves processing \( 2n \) elements, and Step 2 (which converts the remaining elements of \( j \)th column to 0, Eq. (5)) involves processing \( (n-1) \) rows of \( 2n \) elements each (of which \( n \) are identically zero). If these two steps are performed sequentially, without any parallelization, the time complexity of the program becomes proportional to \( n \times (n-1) \times n \), i.e. \( \mathcal{O}(n^3) \).
We now redesign the algorithm to exploit the capabilities of GPU parallel processing. We perform Steps 1 and 2 in parallel, i.e. for Step 1, spawn \(n\) threads and process the whole row at once, which can be implemented in CUDA C as shown in Fig. 1. For Step 2 we spawn \(n \times (n - 1)\) threads to convert the rest of the column to 0 which can be implemented in CUDA C as shown in Fig. 2.

```
_global_ void fixRow(float *matrix, int size, int rowId) {
    // The ith row of the matrix
    _shared_ float Ri[512];
    // The diagonal element for ith row
    _shared_ float Aii;
    int colId = threadIdx.x;
    Ri[colId] = matrix[size * rowId + colId];
    Aii = matrix[size * rowId + sharedRowId];
    _syncthreads();
    // Divide the whole row by the diagonal element making sure it is not 0
    Ri[colId] = Ri[colId] / Aii;
    matrix[size * rowId + colId] = Ri[colId];
}
```

**Fig. 1.** Process \(n\) elements i.e. \(a_{ij}\) in the left side matrix to \(a_{ij}\) on the right side matrix (top), and corresponding pseudo code (bottom).

```
_global_ void fixColumn(float *matrix, int size, int colId) {
    int i = threadIdx.x;
    int j = blockIdx.x;
    // The colId column
    _shared_ float col[512];
    // The jth element of the colId row
    _shared_ float AColIdj;
    // The jth column
    _shared_ float colj[512];
    col[i] = matrix[i * size + colId];
    if (col[i] != 0) {
        colj[i] = matrix[i * size + j];
        AColIdj = matrix[colId * size + j];
        if (i != colId) {
            colj[i] = colj[i] - AColIdj * col[i];
        }
        matrix[i * size + j] = colj[i];
    }
}
```

**Fig. 2.** Process \(n^2\) elements starting from \(a_{ij}\) on the left side matrix to \(a_{ij}\) on the right side matrix (top), and corresponding pseudo code (bottom).
We also minimize computations by processing only the first \( n \) columns (starting from the \( j \)th column) in the Step 2 instead of processing all the \( 2n \) columns, as also indicated in Fig. 2. This will reduce the computations to half without affecting the final result, since for the columns after the \((n+j)\)th column, the elements above the \( j \)th row are still zero, hence they will not contribute to the step as the product will become zero.

A further speed boost is obtained when the program is run using shared memory. Shared memory is a local common memory available to all the threads of the same thread block. Data read-and-write on shared memory is faster than that from GPU global memory [39] making the shared memory act like a cache for the GPU. Thus, subject to availability of computational resources, the time complexity becomes proportional to \( \frac{n}{C^2(1+1)} \), i.e. \( O(n) \).

The final pseudo code is presented in Fig. 3. Pivoting is done to prevent division by zero exception. While pivoting can be done inside the second for thread loop using an if...else condition, it is done prematurely for all elements to avoid extra GPU cycles being used in the if...else comparison.

4. Testing and results

4.1. Hardware advantages and limitations

As shown above, if all the \( n^2 \) computations can be done in parallel, the complexity of the Gauss Jordan algorithm on a massively parallel architecture reduces to \( O(n) \). For all the computation to be in parallel, the thread creation should be very light, else the creation time of \( n^2 \) threads will contribute to time complexity. This is where CUDA has an advantage [39] over other methods for parallel computation. Thread creation in CUDA is very light and it does not contribute to the run-time of the program.

While thread creation is lighter using CUDA, the use of GPU brings in some limitations dependent on the hardware of the GPU. The first limitation has to do with the running of \( n^2 \) threads in parallel. This depends upon the type of GPU used for computation. A modern GPU (Nvidia GTX 590) may have up to 49,152 active threads divided into sets of 32 (called warps) [32] but only 1024 [32] dispatched threads running in parallel, i.e. 32 SM \times 48\) Resident Warps per SM \times 32 Threads per warp (Table 9 of [32]) active threads but only 32 SM \times 32 Threads per warp running in parallel, any number above that will lead to stacking of warps waiting for their turns. Thus, up to \( n = \sqrt{1024} = 32 \), the algorithm will have time complexity of \( O(n) \); the complexity starts increasing for \( n > 221 \) and becomes quadratic at \( n = \sqrt{1024} = 32 \). If cluster of parallel GPUs is used and the program is run using all the GPUs, then this hardware capacity can be further increased.

The second issue is the space available for the shared memory. We are using GPU’s shared memory to store maximum of one row and one column of the matrix in order to prevent reads and writes to the global memory of the GPU. Access to the global memory is very slow as compared to the shared memory [40], but at the same time, the size of the shared memory is very small (up to 48 KB) as compared to the global memory (up to 3 GB).

The third issue is the maximum allowed block size (in terms of maximum number of threads per block). In this algorithm, each block handles the computations involved for one column at a time. If the column size is greater than the maximum allowed block size, then the matrix would be needed to vertically split so that each part is having appropriate column size. Maximum number of threads allowed in a modern GPU is 1024 [32] (#compute-capability-3.0).

A modern CPU, in comparison, can spawn a small number of threads, typically 8 or 12 [41]. Adding to this the fact that thread

---

**Fig. 3.** Pseudo code explaining the Gauss Jordan algorithm for matrix inversion adapted to GPU computing.
creation on a CPU is not as light as that on a GPU, a GPU has a clear advantage over a CPU for parallel computation.

4.2. Results and comparisons

The Algorithm was programmed using CUDA C and was tested against various sizes of the following types of matrices: identity matrix, sparse matrix, band matrix (with \( k = n/2 \)), random matrix and hollow matrix. For this paper, the GPU used was Nvidia GTX 260 (216 Cores), capable of storing 9216 active threads and 288 dispatched concurrent threads (9 SM \( \times \) 32 Resident Warps/SM \( \times \) 32 Threads/Warp and 9 SM \( \times \) 32 Threads/Warp respectively), having a shared memory size of 16 KB (up to 4096 floating point locations) and a maximum thread per block limit of 512. The CPU used for comparison purpose is Intel Quad Core processor Q8400 @ 2.66 GHz each, capable of running 4 threads in parallel.

As the compute capability of GTX 260 is 1.3, we chose to use single-precision floating-point arithmetic operations instead of double-precision floating-point numbers as the wrap cycles involved for double-precision operations is 8 times that of single precision [32] (#compute-capability-1-x). In depth accuracy and performance analyses on single and double precision floating point numbers can be found in [42]. Finally, the optimization flags used while compiling the program are -ftz = true, --use_fast_math and --prec-div = true.

While Identity matrix is used to test the accuracy of the inverse, sparse matrix is the most general matrix in case of problems involving structural analysis. Hollow matrix is a matrix having all the diagonal elements as zero. Thus a hollow matrix would require an extra row transformation to fix the diagonal element for each column making it the type of matrix with the maximum amount of computation involved. For performance testing of the algorithm, all the matrices were stored in dense format.

Fig. 4 displays the computational time taken for matrix size up to 100 and demonstrates the linear nature of time complexity. Fig. 5 displays the time taken by the algorithm on GPU over a larger scale as compared to a CPU.

It is observed from Fig. 4 that the graph is still linear for around \( n = 64 \) even though a matrix of size 64 require 4096 parallel running threads. This is explained by the fact that all the 4096 threads or 128 warps were already loaded in memory and scheduled to be dispatched thus there was no latency observed while dispatching

Fig. 4. (a–e): Linear computation time for matrix inversion is observed up to \( n = 64 \) using GPU, (f) computation time for inverting hollow matrix using CPU.
these warps of threads. Also, due to the elegant design of the algorithm in Fig. 2, running time of each thread is not significant enough as compared to the total number of threads to contribute to complexity. We also observe that the first quadratic curvature in the graph is observed at around \( n = 100 \). For \( n = 100 \), 10,100 threads are required, whereas only 9216 threads can be scheduled at a time. This leads to latencies between dispatching warps when the remaining 884 threads are loaded in the memory. Moreover, 10,100 would require 35 cycles of 288 parallel running threads to complete the algorithm. Thus values around \( n = 100 \) start displaying the quadratic nature of algorithm due to hardware limitations.

5. Conclusions

The ability to invert large matrices accurately and quickly determines the effectiveness of a wide range of computational algorithms and products. GPU computing is ideally suited for massively parallel tasks as the thread creation and memory transfer overheads are negligible. We have redesigned the Gauss Jordan algorithm for matrix inversion on GPU based CUDA platform, tested it on five different types of matrices (identity, sparse, banded, random and hollow) of various sizes, and have shown that the time complexity of matrix inversion scales as \( n \) if enough computational resources are available (we were limited by only one GPU capable of running 288 threads in parallel with which we showed that the time complexity is in order of \( n \) up to \( n \approx 64 \)). Linear scaling will continue by using a network of GPUs. We also show that GPU based parallelization for matrix inversion is orders of magnitude faster than CPU based parallelization. Even a small matrix of size \( 10 \times 10 \) did better using the parallel CUDA Gauss Jordan algorithm.

References