Computers and Structures 128 (2013) 31–37

Contents lists available at SciVerse ScienceDirect

Computers and Structures

journal homepage: www.elsevier.com/locate/compstruc

Technical Note

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

ARTICLE INFO

Article history:
Received 5 April 2012
Accepted 24 June 2013
Available online 13 September 2013

Keywords:
Graphics processing unit
Compute unified development architecture
Matrix inversion
Gauss Jordan
Parallelization

ABSTRACT

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.

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

* Corresponding author. Tel./fax: +91 3222 283422.
E-mail addresses: baidurya@civil.iitkgp.ernet.in, baidurya.bhattacharya@jhu.edu, baidurya@udel.edu (B. Bhattacharya).

0045-7949/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compstruc.2013.06.015

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:

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

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

$$\begin{array}{lll} 1.\ P_1 = A_{11}^{-1} & 2.\ P_2 = A_{21} \times P_1 & 3.\ P_3 = P_1 \times A_{12} \\ 4.\ P_4 = A_{21} \times P_3 & 5.\ P_5 = P_4 - A_{22} & 6.\ P_6 = P_5^{-1} \\ 7.\ C_{12} = P_3 \times P_6 & 8.\ C_{21} = P_6 \times P_2 & 9.\ C_{11} = P_1 - P_3 \times C_{21} \\ 10.\ C_{22} = -P_6 & & \end{array} \tag{2}$$

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 $\mathbf{A_{11}}$. If the quarter matrix chosen in step 1 is singular, then pivoting is required and the matrix with the larger determinant between $\mathbf{A_{11}}$ and $\mathbf{A_{22}}$ is chosen for step 1. This check is performed in each level of recursion in steps 1 as well as 6, making this algorithm cumbersome. While certain parallel implementations for specific types of matrices already have been studied before [34,35], developing an algorithm to comply with all types of matrices is a difficult task. A much easier and robust parallel implementation of an algorithm can be developed using the straightforward method, i.e. Gauss Jordan method, as we have attempted in this paper.

Parallel computations can be performed both on Central Processing Unit (CPU) and graphics processing unit (GPU). While the term CPU has been used at least since 1960s [36], GPU which is a single chip processor with multiple capabilities and a highly parallel structure, is of more recent vintage since 1999 [32]. Among the many differences between CPU and GPU in terms of architecture and usage, the overriding one with respect to parallelization is the number of cores present on the respective chip. The number of parallel threads that can run on a GPU is orders of magnitude larger than in a CPU [37].

Within a GPU, there are a couple of techniques for performing parallel computation. In the early years, General-purpose computing on graphics processing units (GPGPU) [38] was the technique used to harness the computational power of a GPU by programmers, until CUDA (Compute Unified Device Architecture) was made publically available in 2007 as the platform to replace GPGPU. CUDA has several advantages [39] over GPGPU and CPU which includes faster memory sharing and read backs, minimal threads creation overhead, and flexibility in the choice of programming language, and has been adopted for this work. More extensive information about Nvidia and CUDA model is explained in the Programming Guide published by Nvidia [32].

3. Parallel Gauss–Jordan algorithm

Gauss Jordan method for computation of inverse of a matrix is one of the oldest methods. It is robust, accurate over range of matrices and does not require multiple checks depending upon the type of 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 $\boldsymbol{A}$ of size n starts by augmenting the matrix with the identity matrix of size n:

$$[C] = [A|I] \tag{3}$$

Then, performing elementary row transformations on matrix $\boldsymbol{C}$, the left half of $\boldsymbol{C}'$ is transformed column 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_{ii} into 1 by the transformation:

$$R_i \leftarrow R_i / a_{ii} \tag{4}$$

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

$$R_i \leftarrow R_i - R_j \times a_{ij} \tag{5}$$

After transforming the first column, the matrix reduces to:

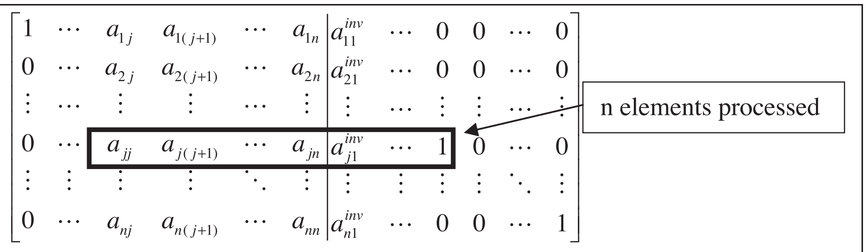
$$[C'] = \left[\begin{array}{ccccc|ccccc} 1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & \cdots & 1/a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} - a_{21} \times a_{12}/a_{11} & a_{23} - a_{21} \times a_{13}/a_{11} & \cdots & \cdots & -a_{21}/a_{11} & 1 & 0 & \cdots & 0 \\ 0 & a_{32} - a_{31} \times a_{12}/a_{11} & a_{33} - a_{31} \times a_{13}/a_{11} & \cdots & \cdots & -a_{31}/a_{11} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} - a_{n1} \times a_{12}/a_{11} & a_{n3} - a_{n1} \times a_{13}/a_{11} & \cdots & \ddots & -a_{n1}/a_{11} & 0 & 0 & \cdots & 1 \end{array}\right] \tag{6}$$

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

$$[C'] = [I|A^{-1}] \tag{7}$$

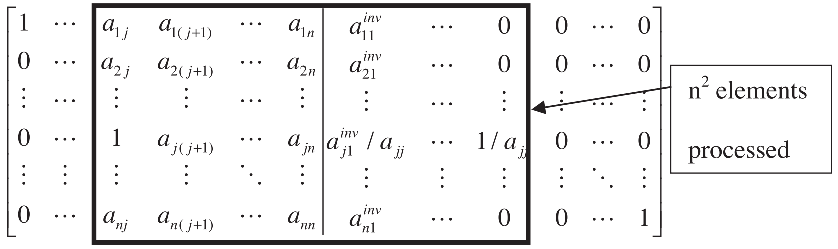
Thus, *Step 1* of the original Gauss Jordan algorithm (which converts jth element of jth column to 1, Eq. (4)) involves processing $2n$ elements, and *Step 2* (which converts the remaining elements of jth column to 0, Eq. (5)) involves processing $(n-1)$ rows of $2n$ elements each (of which n are identically 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. $O(n^3)$.

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.



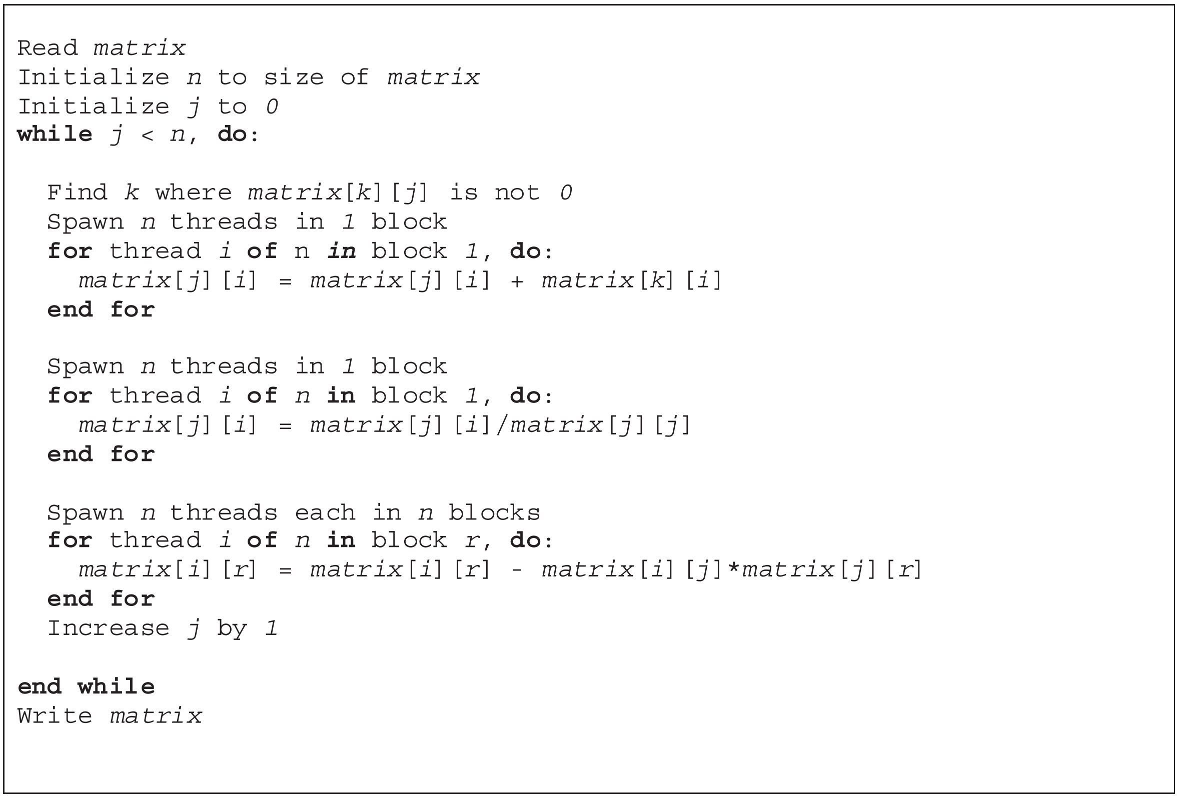
```
__global__ void fixRow(float *matrix, int size, int rowId) {
  // The ith row of the matrix
  __shared__ float Ri[512];
  // The diagonal element for ith row
  __shared__ float Aii;
  int colId = threadIdx.x;
  Ri[colId] = matrix[size * rowId + colId];
  Aii = matrix[size * rowId + sharedRowId];
  __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_{jj} in the left side matrix to a_{jj} 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_{1j} on the left side matrix to a_{nj} on the right side matrix (top), and corresponding pseudo code (bottom).



```
Read matrix
Initialize n to size of matrix
Initialize j to 0
while j < n, do:

  Find k where matrix[k][j] is not 0
  Spawn n threads in 1 block
  for thread i of n in block 1, do:
    matrix[j][i] = matrix[j][i] + matrix[k][i]
  end for

  Spawn n threads in 1 block
  for thread i of n in block 1, do:
    matrix[j][i] = matrix[j][i]/matrix[j][j]
  end for

  Spawn n threads each in n blocks
  for thread i of n in block r, do:
    matrix[i][r] = matrix[i][r] - matrix[i][j]*matrix[j][r]
  end for
  Increase j by 1

end while
Write matrix
```

Fig. 3. Pseudo code explaining the Gauss Jordan algorithm for matrix inversion adapted to GPU computing.

We also minimize computations by processing only the first n columns (starting from the jth 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 jth 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 $n \times (1+1)$, 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 × 48 Resident Warps per SM × 32 Threads per warp (Table 9 of [32]) active threads but only 32 SM × 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 $n = 1024$. 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 bock). 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 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 × 32 Resident Warps/SM × 32 Threads/Warp and 9 SM × 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

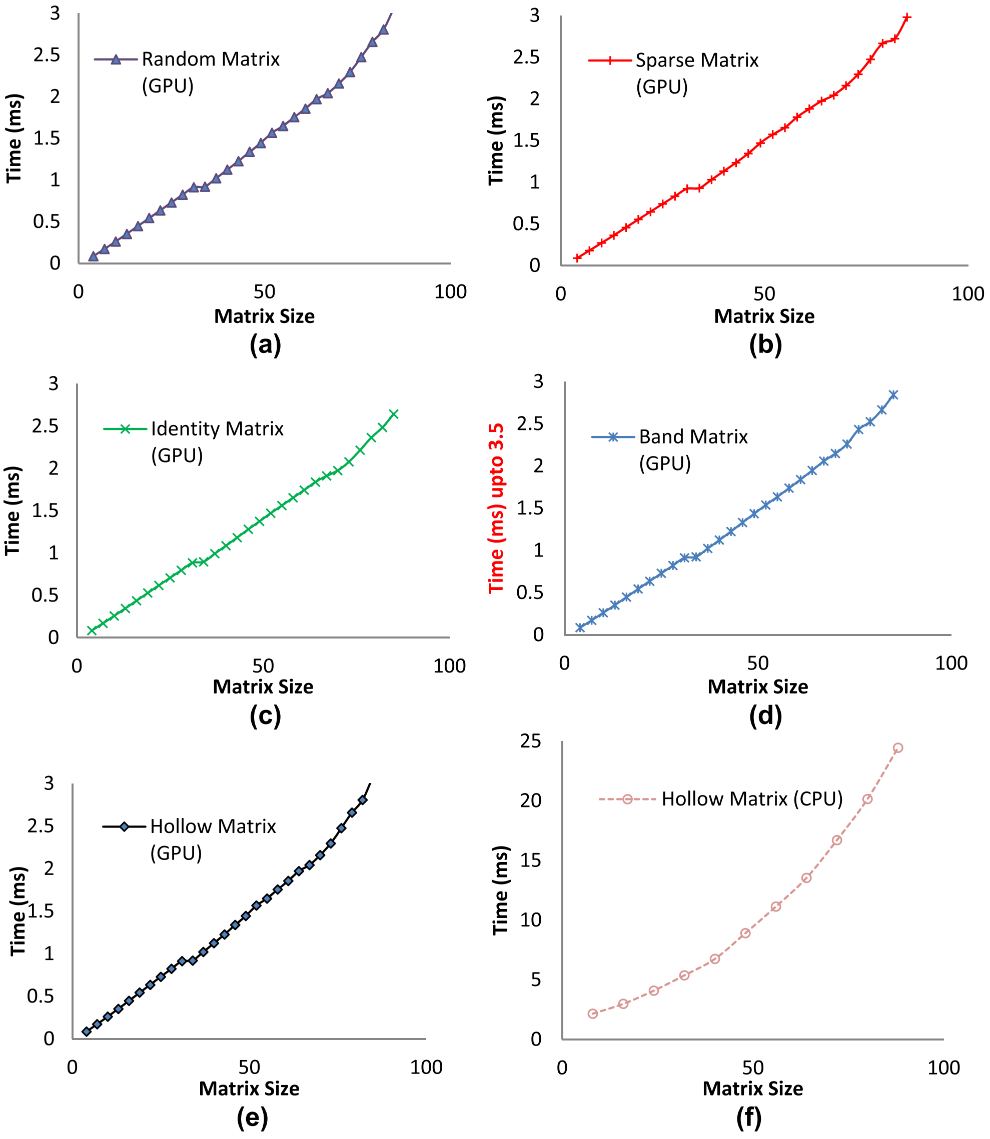


Fig. 4. (a–e): Linear computation time for matrix inversion is observed up to $n \approx 64$ using GPU, (f) computation time for inverting hollow matrix using CPU.

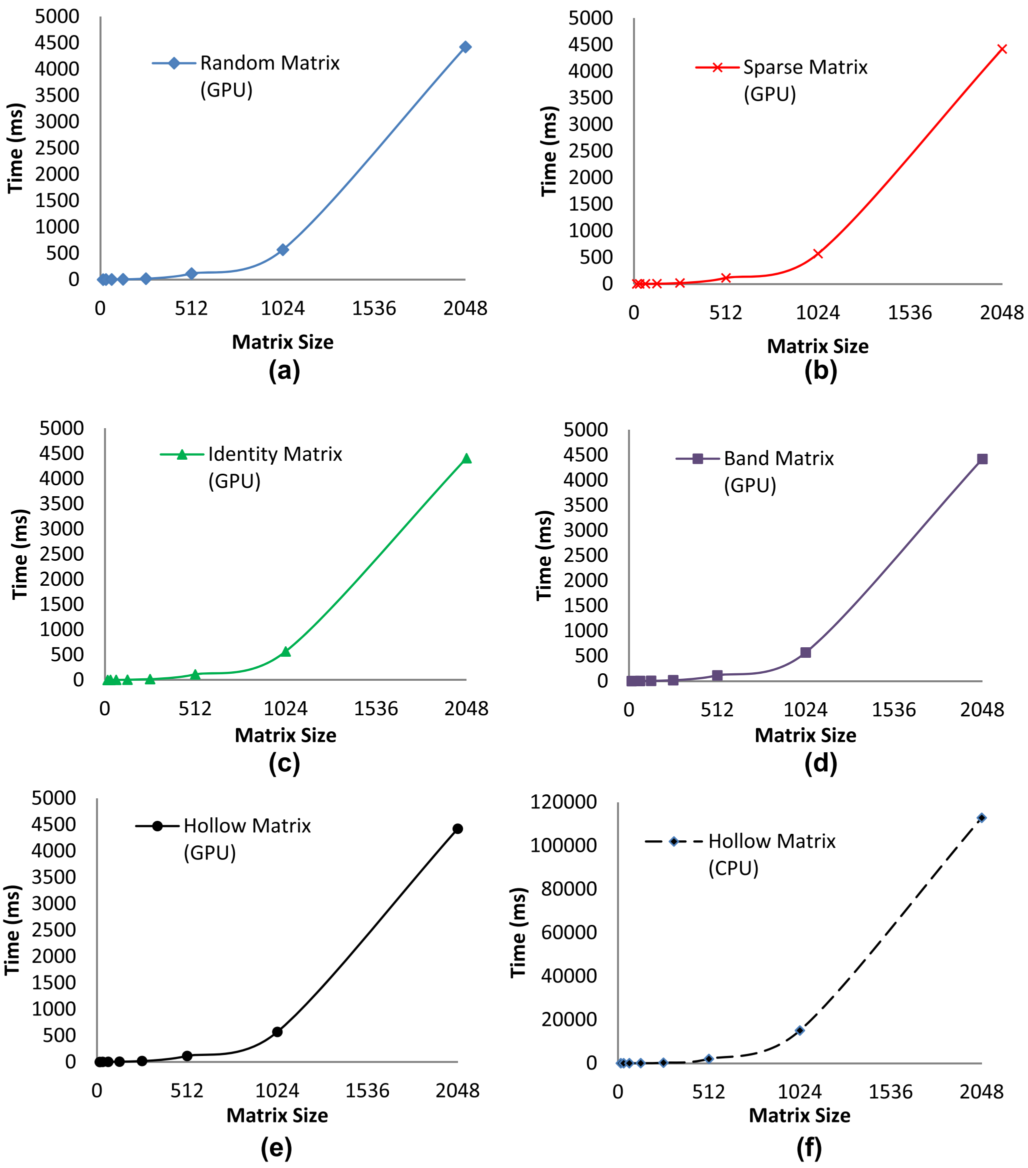


Fig. 5. Computation time for inverting different types of matrices, (a–e): using GPU, (f) using CPU.

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 × 10 did better using the parallel CUDA Gauss Jordan algorithm.

References

[1] Demmel JW. Applied numerical linear algebra. 1st ed. Society for Industrial Mathematics; 1997.
[2] Gantmacher FR. Applications of the theory of matrices. 1st ed. Dover Publications; 2005.
[3] Arfken GB, Weber HJ, Harris FE. Mathematical methods for physicists: a comprehensive guide. 7th ed. Academic Press Inc; 2012.

[4] Marcus M, Minc H. Introduction to linear algebra. New ed. New York: Dover Publications; 1988.

[5] Felippa CA, Park KC. The construction of free–free flexibility matrices for multilevel structural analysis. Comput Methods Appl Mech Eng 2002;191:2139–68.

[6] Liang P, Chen SH, Huang C. Moore–Penrose inverse method of topological variation of finite element systems. Comput Struct 1997;62.

[7] Fung TC. Derivatives of dynamic stiffness matrices. In: Proceedings of the Asian Pacific conference on computational mechanics. Hong Kong; 1991. p. 607–613.

[8] Shi HF, Payandeh S., GPU in haptic rendering of deformable objects. In: Proceedings of haptics: perception, devices and scenarios: 6th international conference. Madrid; 2008. p. 163–168

[9] Farden DC, Scharf LL. Statistical design of non recursive digital filter. IEEE Trans Acoust Speech Signal Process 1974;22:188–96.

[10] Jain AK. An operator factorization method for restoration of blurred images. IEEE Trans Comput 1977;C-26:1061–71.

[11] Jain AK, Padgug RA. Fast restoration of finite objects degraded by finite PSF. J Comput Phys 1978;28:167–80.

[12] Leroux JD, Selivanov V, Fontaine R, Lecomte R. Fast 3D image reconstruction method based on svd decomposition of a block-circulant system matrix. In: IEEE nuclear science symposium and medical imaging conference, NSS–MIC. Honolulu; 2007. p. 3038–3045.

[13] Strassen V. Gaussian elimination is not optimal. Numer Math 1969;13:354–6.

[14] Bailey DH, Ferguson HRP. A Strassen–Newton algorithm for high-speed parallelizable matrix inversion. In: Supercomputing'88, proceedings of the 1988 ACM/IEEE conference on supercomputing. Orlando; 1988. p. 419–424.

[15] Althoen SC, McLaughlin R. Gauss–Jordan reduction: a brief history. Am Math Mon 1987;94:130–42.

[16] Coppersmith D, Winograd S. On the asymptotic complexity of matrix multiplication. SIAM J Comput 1981;11:472–92.

[17] Press WH, Flannery BP, Teukolsky SA, Vetterling WT. Section 2.3: LU decomposition and its applications, numerical recipes in FORTRAN: the art of scientific computing. New York: Cambridge University Press; 2007. p. 34–42.

[18] Burian A, Takala J, Ylinen M. A fixed-point implementation of matrix inversion using Cholesky decomposition. In: Proceedings of the 46th international Midwest symposium on circuits and systems. Cairo; 2003. p. 1431–1433.

[19] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Section 2.10: QR decomposition, numerical recipes: the art of scientific computing. New York: Cambridge University Press; 2007. p. 1256.

[20] Ming G, Eisenstat SC. Efficient algorithms for computing a strong rank-revealing QR factorization. SIAM J Sci Comput 1996;17:848–69.

[21] Fathi Vajargah B. A way to obtain Monte Carlo matrix inversion with minimal error. Appl Math Comput 2007;191:225–33.

[22] Fathi Vajargah B. Different stochastic algorithms to obtain matrix inversion. Appl Math Comput 2007;189:1841–6.

[23] Pan VY. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In: Proceedings of the 19th annual symposium on foundations of computer science. Ann Arbo; 1978. p. 166–176.

[24] Bini D, Capovani M, Romani F, Lotti G. O($n^{2.7799}$) complexity for nxn approximate matrix multiplication. Inf Process Lett 1979;8:234–5.

[25] Schönhage A. Partial and total matrix multiplication. SIAM J Comput 1981;10:434–55.

[26] Romani F. Some properties of disjoint sums of tensors related to matrix multiplication. SIAM J Comput 1982;11:263–7.

[27] Strassen V. The asymptotic spectrum of tensors. J Für Die Reine und Angewandte Mathematik 1988;384:102–54.

[28] Coppersmith D, Winograd S. Matrix multiplication via arithmetic progressions. J. Symbolic Comput 1990;9:251–80.

[29] Williams VY. Breaking the Coppersmith–Winograd barrier. Retrieved from: Accessed: http://unicyb.kiev.ua/~vingar/progr/201112/1semestr/matrixmult.pdf.

[30] Cohn H, Kleinberg R, Szegedy B, Umans C. Group-theoretic algorithms for matrix multiplication. In: 46th annual IEEE symposium on foundations of computer science. Pittsburgh; 2005. p. 379–388.

[31] JAIN AK. Fast Inversion of banded toeplitz matrices by circular decompositions. IEEE Trans Acoust Speech Signal Process 1978;26.

[32] CUDA Programming Guide. Retrieved from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 05 December 2012.

[33] Vancea C, Vancea F. Parallel algorithm for computing matrix inverse by Gauss–Jordan method. J Comput Sci Control Syst 2008;1:110–3.

[34] Gravvanis GA, Filelis-Papadopoulos CK, Giannoutakis KM. Solving finite difference linear systems on GPUs: CUDA based parallel explicit preconditioned biconjugate conjugate gradient type methods. J Supercomput 2011;61:590–604.

[35] Filelis-Papadopoulos CK, Gravvanis GA, Matskanidis PI, Giannoutakis KM. On the GPGPU parallelization issues of finite element approximate inverse preconditioning. J Comput Appl Math 2011;236:294–307.

[36] Weik MH. A third survey of domestic electronic digital computing systems, (Report No. 1115); 1961 Retrieved from B.R. Laboratories: http://ed-thelen.org/comp-hist/BRL61.html.

[37] CUDA C Best Practices Guide. Retrieved from: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed: 05 December 2012.

[38] General-purpose computing on graphics processing units. Retrieved from: http://www.gpgpu.org/. Accessed: 01 April 2012.

[39] Mei W, Hwu W, Kirk D. Video lectures for ECE 498AL, University of Illinois [m4v Video]; Retrieved from: http://www.nvidia.com/content/cudazone/cudacasts/CUDA%20Programming%20Model.m4v. Accessed: 01 April 2012.

[40] Farber R. CUDA application design and development. 1st ed. Waltham Massachusetts: Morgan Kaufman; 2011.

[41] Intel-Core-i7-3930K-Processor. Retrieved from: http://ark.intel.com/products/63697/Intel-Core-i7-3930K-Processor-%2812M-Cache-3_20-GHz%29. Accessed: 01 April 2012.

[42] Whitehead N, Fit-Florea A. Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs. Retrieved from: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf. Accessed: 05 December 2012.