International Journal of Grid and High Performance Computing
Volume 10 • Issue 1 • January-March 2018

Embarrassingly Parallel GPU Based Matrix Inversion Algorithm for Big Climate Data Assimilation

M. Varalakshmi, VIT University, Vellore, India
Amit Parashuram Kesarkar, National Atmospheric Research Laboratory, Chittoor, India
Daphne Lopez, VIT University, Vellore, India

ABSTRACT

Attempts to harness the big climate data that come from high-resolution model output and advanced sensors to provide more accurate and rapidly-updated weather prediction, call for innovations in the existing data assimilation systems. Matrix inversion is a key operation in a majority of data assimilation techniques. Hence, this article presents out-of-core CUDA implementation of an iterative method of matrix inversion. The results show significant speed up for even square matrices of size 1024 X 1024 and more, without sacrificing the accuracy of the results. In a similar test environment, the comparison of this approach with a direct method such as the Gauss-Jordan approach, modified to process large matrices that cannot be processed directly within a single kernel call shows that the former is twice as efficient as the latter. This acceleration is attributed to the division-free design and the embarrassingly parallel nature of every sub-task of the algorithm. The parallel algorithm has been designed to be highly scalable when implemented with multiple GPUs for handling large matrices.

KEYWORDS

Big Climate Data, Convergence Rate, GPU, Iterative Method, Matrix Type Identification, Numerical Weather Prediction, Parallel Matrix Inverse, Parallel Reduction

1. INTRODUCTION

The advent of Big data technology has brought a great revolution in the science of Numerical Weather Prediction. Big data in NWP actually refers to 'climate big data' that come from rapid and dense observations from advanced sensors and very high-resolution model output. A ten-fold increase in the model resolution would require 10^4 more computations for the four dimensions in space and time. To achieve this massively challenging throughput and to fully utilize this big data so as to provide more accurate and rapidly updated weather prediction, innovations have to be brought to the existing Data Assimilation and NWP systems (Big Data Assimilation) (Miyoshi et al., 2016a; Miyoshi et al., 2016b). This can help strengthen our early warning system against regional, sudden and severe calamities such as hurricanes, heavy rain, flooding, landslides and the alike. Innovative research has already started towards speeding up the various phases of NWP such as observation data processing, model run and data transfer between model and DA. Even in the Data assimilation phase, ways to improve storage and processing of large matrices and vectors can be explored. With the three spatial dimensions and one temporal dimension considered in Variational data assimilation algorithms and Kalman Filter based assimilation algorithms, the atmospheric state variables such as Wind, Pressure,

DOI: 10.4018/IJGHPC.2018010105

Humidity etc at all grid points for various vertical layers and time instants are represented in a vector with around 10^8 entries. Likewise, the measurement vector contains 10^6 observation entries. Due to large size of these vectors, the resulting model error covariance and observation error covariance matrices too will be large, of the order of $O(10^8X10^8)$. Hence the performance of these assimilation methods depends on the design and implementation of better algorithms for processing of large matrices in general and inversion in particular, and this was the impetus behind our proposed work.

The massive number crunching capacity needed to work with large matrices can be made possible by employing Graphics Processing Units (GPUs). CUDA is well suited for data-parallel algorithms (Garland et al., 2008) such as shallow water model (Playne & Hawick, 2015), delivering high computational throughput if few design principles are followed to fully utilize the GPU's processor cores and their shared memory that is critical to the performance of many efficient algorithms. Various improvements made to the storage format for efficient execution of SpMV operations on GPUs (Gao, Qi & He, 2016; Koza, Matyka, Szkoda & Mirosław, 2014; Dziekonski, Lamecki & Mrozowski, 2011) have shown this. Wu, Ke, Lin and Jhan (2014) claim that adjusting the number of threads dynamically helps to completely utilize the compute power of GPUs. Modeling tools (Zouaneb, Belarbi & Chouarfia, 2016) also lend a helping hand in validating task scheduling on GPUs and analyzing the performance. Earlier studies show that GPU implementations are several times faster than its CPU counterpart (Helfenstein & Koko, 2012) and can be efficient if the matrix is represented and processed using the two-dimensional textures that GPUs are optimized for (Galoppo, Govindaraju, Henson & Manocha, 2005). Further studies have revealed that parallel implementation of algorithms on hybrid platform consisting of CPU and GPUs (Ezzatti, Quintana & Remón Gómez, 2011a; Benner, Ezzatti, Quintana-Ortí & Remón, 2009; Ezzatti, Quintana-Orti, & Remon, 2011b) has proved to be more efficient for both small and large size matrices than the pure GPU implementation.

To support the efficient execution of linear algebra applications, there are several linear Algebra libraries optimized for GPU architecture such as CUBLAS and MAGMA for finding matrix inverse. MAGMA linear algebra C/C++ library (A. Chrzeszczyk & J. Chrzeszczyk, 2013) provides code for calculating matrix inverse for a regular matrix and positive definite matrix both in single precision and double precision. However, these libraries are not efficient for certain applications and there are other findings that show that further enhancements can be made to these implementations.

According to Haidar, Abdelfatah, Tomov and Dongarra (2017), high performance GPU-only algorithm developed for dense Cholesky factorization to run on latest GPUs and the hybrid panel-based LU decomposition algorithm outperform the existing libraries. The tile data layout followed in Cholesky-based matrix inversion (Ibeid, Kaushik, Keyes & Ltaief, 2011) results in up to 5 and 6-fold improvement compared to the equivalent routines from MAGMA V1.0 by completely removing the synchronization points and unlike Magma it is not memory-limited and can scale beyond the available device memory. Efficient batched solvers have been developed for a set of small dense matrices as the pre-existing solvers were either just memory-bound, or even if highly optimized, did not exceed in performance the corresponding CPU versions (Haidar, Dong, Luszczek, Tomov & Dongarra, 2015). If matrix inversion algorithms are tailored to handle specific application requirements, they outperform the methods employed in the standard libraries to calculate direct inverse (Prabhu, Rodrigues, Edfors & Rusek, 2013; Ylinen, Burian & Takala, 2003; Xingbo, 2011).

Moreover, these libraries employ direct methods that either use LU-decomposition with partial pivoting or Cholesky decomposition, for factorization of matrix. While the former suffers from lack of optimal stability, high convergence time for sparse matrices as compared to dense matrices and inability to find approximate solution (Agarwal & Mehr, 2014), the latter is not very robust and works only for symmetric positive definite matrices. On the other hand iterative methods are more stable, simpler, less prone to numerical errors, best suited for large matrices due to smaller storage requirements and more specifically, efficient for sparse matrices. They compensate for individual and accumulation of round-off errors as they are a process of successive refinement (Jamil, 2012).

The existing iterative methods are not without their limitations. Either their convergence rate is too slow or convergence is not guaranteed for all types of matrices. Amidst the iterative methods, those that involve only the simple elementary arithmetical operations (Chang, 2015) and use approximations to avoid division and square root operations (Zhou et al., 2012) have been shown to execute faster. Hence in the recent years, the quest for inversion algorithms with high order of convergence has intensified; more specifically division-free algorithms are explored, the reason being two-fold: It is never divide overflow and also division operation is the slowest among all the arithmetic operations. These are the motivating factors for choosing a matrix multiplication based iterative method of inversion that has seventh order of convergence as a topic of study in this paper. Such high order iterative methods are so efficient for very ill-conditioned linear systems or to find robust approximate inverse preconditioners (Soleymani, 2012). Also, this method will be of great accuracy when implemented on parallel machines. The embarrassingly parallel nature of the various subtasks of this method such as matrix type identification and initial inverse construction makes it favorable for implementing in a GPU hardware environment that has a massively parallel architecture. Thus, this paper focuses on implementing a rapid numerical algorithm to compute out-of-core matrix inverse in a GPU-accelerated parallel computing platform, so as to obtain high throughput. To facilitate the comparison of our parallel iterative approach with a direct method of matrix inversion, it also parallelizes the Gauss-Jordan algorithm. Although Sharma, Agarwala and Bhattacharya (2013) have made attempts earlier to parallelize Gauss-Jordan algorithm, it might not be useful for applications that involve double-precision floating point arithmetic and matrices too large to be processed within a single kernel call. On similar grounds of comparison, our parallel iterative method has been proved to be more efficient than the direct method.

The rest of the paper is organized as follows. Section 2 describes the matrix multiplication based iterative algorithm for matrix inversion. Section 3 expounds the implementation of proposed parallel algorithm for the iterative method of matrix inversion. Section 4 presents the experimental results. At last, Section 5 draws the conclusions.

2. MATRIX MULTIPLICATION BASED ITERATIVE ALGORITHM FOR MATRIX INVERSION

Soleymani (2012) and Soleymani (2013) have proved that the following matrix multiplication-based iterative method for finding matrix inverse has seventh order of convergence, with an appropriate initial guess for the inverse.

$$V_{n+1} = \frac{1}{16} V_n \begin{pmatrix} 120I + AV_n(-393I + AV_n(735I + AV_n(-861I + \\ AV_n(651I + AV_n(-315I + AV_n(93I + \\ AV_n(-15I + AV_n))))))) \end{pmatrix}$$

n= 0, 1, 2,…

With a higher rate of convergence, matrix inverse can be evaluated with fewer iterations but still with high accuracy.

To achieve seventh order convergence for this method, the initial inverse, V_0 should be constructed using any of the following three ways.

A. For a strictly diagonally dominant matrix, $V_0 = \text{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \frac{1}{a_{33}}, \ldots\ldots, \frac{1}{a_{nn}}\right)$ where a_{ii} are the diagonal elements of A.

B. For a matrix A, $V_0 = \frac{A^T}{\left(\|A\|_1 \|A\|_\infty\right)}$., where A^T is the transpose of the original matrix A and $\|A\|_1$ and $\|A\|_\infty$ are 1-Norm and Infinity Norm of the matrix A.

C. In case of failure of the above two approaches, $V_0 = \alpha I$ where I is the identity matrix, and $\alpha \in R$ should adaptively be determined such that $\|I - \alpha A\| < 1$.

In the context of big climate data, as huge matrices of size million by million have to be worked upon, the aforementioned method requires $O(n^3)$ operations to be executed for the long series of matrix multiplications that are to be performed. Even with a machine offering Teraflops performance, overall computation time will be approximately $(10^{18}) / (8.6X104^X1012^)) = 11.5$ days. To cater to this huge computation demand of the application, Graphics Processing Unit (GPUs) featuring 1000s of general purpose compute processors has been employed.

2.1. Parallelization of Matrix Inverse

The extremely high processing power of GPU-accelerated high-end system has been used to expedite both inversion and initial inverse matrix construction as well, without sacrificing accuracy. Figure 1 is a schematic representing the subtasks of the proposed parallel algorithm. Salim, Akkirman, Hidayetoglu and Gurel (2015) have examined the size limit of the matrices that can be solved by a GPU and Intel Xeon Phi. Their investigation shows that GPU cannot support matrices larger than a specific size (20000) owing to the lesser amount of memory directly available to the device. Our proposed approach overcomes this limitation by parallelizing every subtask in a way that works by splitting up huge matrices into several smaller blocks/tiles that can be supported by a specific GPU and iterating the kernel for all these blocks. Literature shows that this approach is not very new. To overcome the limitation of device memory on GPU, only a chunk of the distance matrix is computed instead of the entire distance matrix for each kernel call (Arefin, Riveros, Berretta & Moscato, 2012) and only partial input tiles are loaded into the shared memory (Kijsipongse, Suriya, Ngamphiw & Tongsima, 2011) and the kernel is repeated several times.

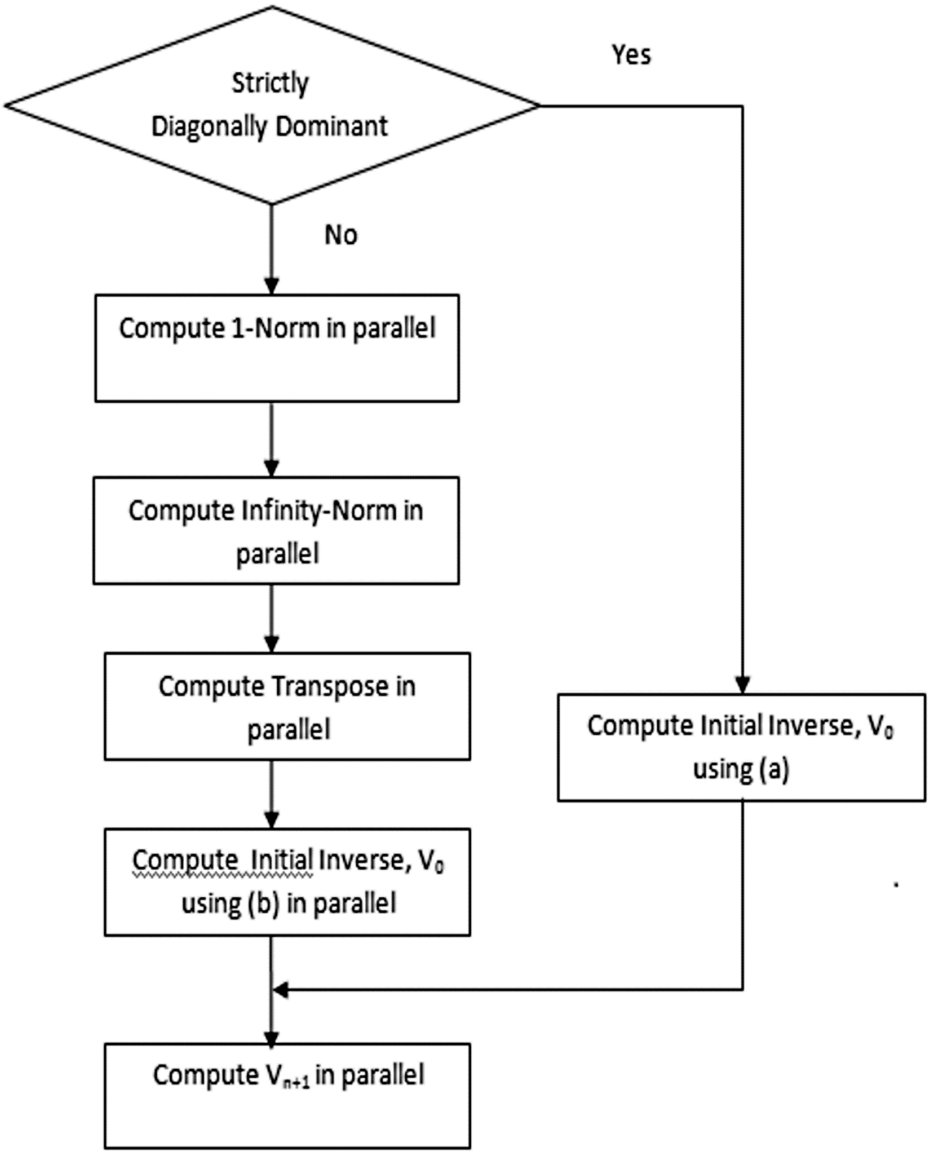
2.1.1. Matrix Type Identification

Speed of convergence of iterative methods depends on the initial inverse approximation. This conveys the fact that construction of initial guess for the inverse is of prime importance to this approach. Multiple ways have been suggested by which initial inverse can be constructed depending on the type of the original input matrix (Soleymani, 2012).

A. For a strictly diagonally dominant matrix, $V_0 = \text{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \frac{1}{a_{33}}, \ldots\ldots, \frac{1}{a_{nn}}\right)$ where a_{ii} are the diagonal elements of A.

This makes it necessary to check for the type of the input matrix, a. A strictly diagonally dominant matrix is one in which for every row the absolute value of the diagonal element is greater than the sum of absolute values of all other elements along that row. An efficient method to perform this test in a parallel environment is to sum up all the elements along a particular row by performing parallel reduction of that row and then subtract the diagonal element. The difference should be smaller than the diagonal element. This condition must be satisfied for all the rows to declare it to be a strictly diagonally dominant matrix.

Figure 1. Implementation flowchart of the proposed parallel matrix inversion algorithm



$$\left(\sum_{j=0}^{n} a_{ij} - a_{ii}\right) < a_{ii} \; for \; all \; i = 0,1,\ldots n$$

The above procedure is tantamount to the summation of non-diagonal elements of every row.

Very large matrices cannot be transferred to the GPU in a single step for its limited memory. This requires partitioning of matrices and reducing the partitions individually. There are several approaches to partitioning the large size matrices and transferring them to the device for reduction. Column-wise and row-wise partitioning of matrices albeit simple to implement will not scale well should the matrix size increase further. Individual columns and rows should in turn be split up to fit into the

device memory and the algorithm has to be restructured to handle segments of rows and columns. To alleviate these problems, block approach is followed in which matrices are partitioned into blocks and parallel reduction is done for individual blocks. The original matrix of order *fullsizeXfullsize* is partitioned into several small sub-matrices of order s*izeXsize.* This will yield (f*ullsize/size) X* (fu*llsize/size)* unique sub-matrices (see Figure 2).

To befit the two-level hierarchy of threads prevailing in GPU architecture, each *sizeXsize* sub-matrix is further sub-divided into several blocks of b*locksizeXblocksize e*lements each. Thus a total of (si*ze/blocksize)X*(*size/blocksize)* blocks have to be processed to perform row-wise summation of a single sub-matrix. After splitting up the matrix into blocks, parallel reduction technique is applied to sum up the bl*ocksize e*lements of each row. Eventually, si*zeX(size/blocksize) w*ould be the order of the matrix to be taken for the next level processing. During the next phase each row of (s*ize/blocksize) e*lements is reduced individually to obtain a single column of si*ze e*lements. The same procedure is repeated for every si*zeXsize s*ub-matrix until all the (fu*llsize/size) X* (ful*lsize/size) s*ub-matrices are processed. At the end, a single ful*lsizeX 1 arr*ay is obtained from which the appropriate diagonal elements are subtracted and verified if the results are still lesser than the diagonal elements in each row or not. If true then the initial inverse matrix is computed as follows.

$$V0 = \text{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \frac{1}{a_{33}}, \ldots\ldots, \frac{1}{a_{nn}}\right)$$ where aii $_a$re the diagonal elements of A.

This can be best illustrated by considering a matrix with f*ullsize* = 16, s*ize* = 4 and b*locksize* = 2. Figures 3-7 show the first level partitioning of the original matrix into 4 sub-matrices of order 4*X4* and the second level partitioning of a single sub-matrix into 4 smaller blocks of order 2*X2* and their parallel reduction.

2.1.2. Initial Inverse Construction

For a strictly diagonally dominant matrix, initial inverse V_0 can be easily constructed as the non-diagonal elements are zeroes. On the other hand, for a matrix A that is not strictly diagonally dominant initial inverse is calculated as $V_0 = \frac{A^T}{\left(\|A\|_1 \|A\|_\infty\right)}$, where A^T is the transpose of the original matrix A

Figure 2. Reading the original matrix as sub-matrices

```
quot=fullsize/size;
repeat=quot*quot;
for(r=0;r<repeat;r++)
 {
   quot1=r/quot;
   rem=r%quot;
       fseek(fp,(quot1*size*fullsize+rem*size)*sizeof(double),0);
       for(i=0;i<size;i++)
       {
              for(j=0;j<size;j++)
              {
                     fread(&a[i*size+j],sizeof(double),1,fp);
              }
              fseek(fp,sizeof(double)*size*(quot-1),SEEK_CUR);
       }
}
```

Figure 3. Partitioning of original matrix into sub-matrices

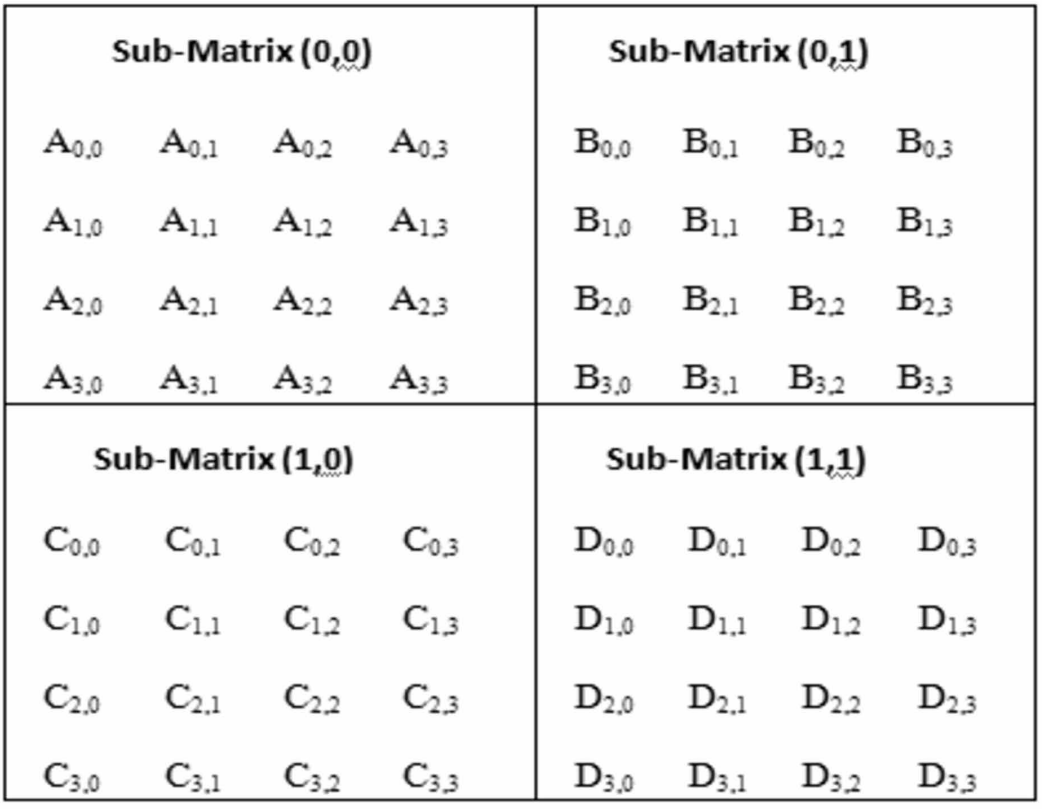


Figure 4. Partitioning of sub-matrix into blocks and their reduction

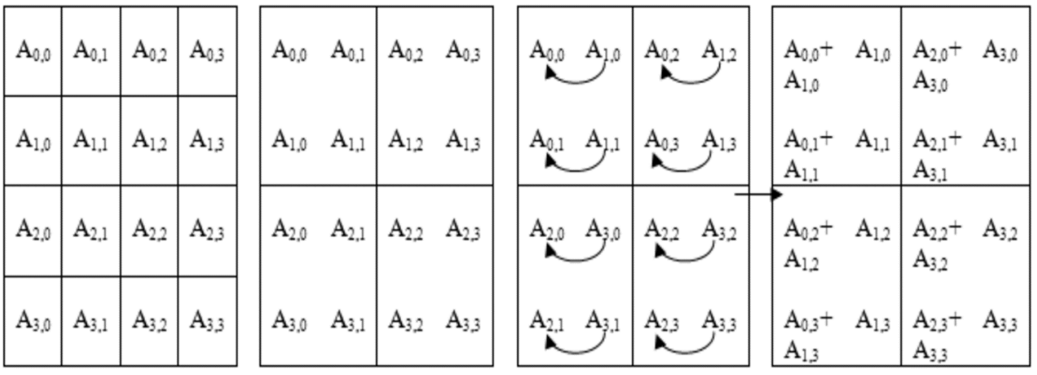


Figure 5. Reduction of individual sub-matrices

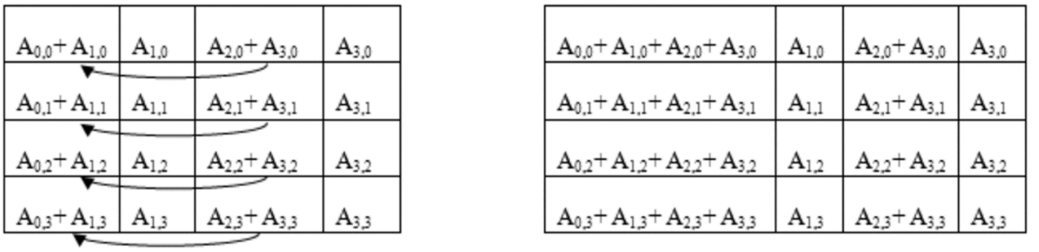
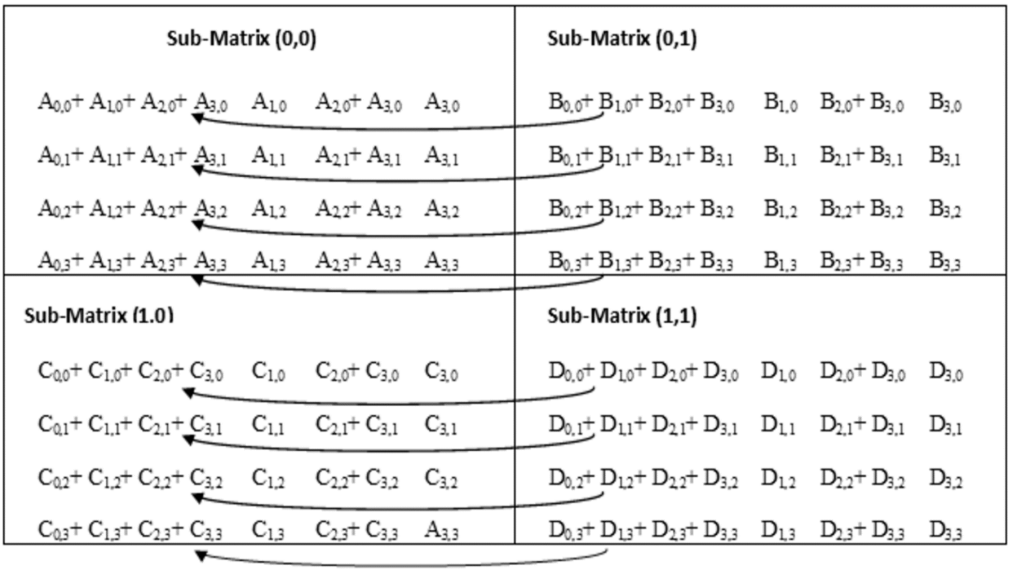


Figure 6. Reduction of sub-matrices to form reduced matrix



and $\|A\|_1$ and $\|A\|_\infty$ being 1-Norm and Infinity Norms of the matrix A. As a result, initial inverse construction necessitates the computation of 1-Norm and Infinity-Norm and transpose of the matrix.

2.1.2.1. Norm Computation

1-Norm ($\|A\|_1$) $= \max\limits_{1\le j\le n}\left(\sum\limits_{i=1}^{n} |\, a_{ij}\,|\right)$ gives the maximum of the sum of absolute values taken along each column. Infinity-Norm ($\|A\|_\infty$) $= \max\limits_{1\le i\le n}\left(\sum\limits_{j=1}^{n} |\, a_{ij}\,|\right)$ gives the maximum of the sum of absolute values taken along each row. Additional speed up can be achieved by parallelizing the norm computation. Both the norms involve summing up all values along a particular column or row. The idea is to adopt the same parallel reduction technique but this time with max operator. For 1-Norm and Infinity-Norm, *1Xfullsize and fullsizeX1 a*rrays are obtained respectively which are in turn reduced to single scalar maximum each.

2.1.2.2. Optimization Techniques

Figure 7. Matrix after reduction

Sub-Matrix (0,0)				Sub-Matrix (0,1)			
$A_{0,0}+A_{1,0}+A_{2,0}+A_{3,0}+B_{0,0}+B_{1,0}+B_{2,0}+B_{3,0}$	$A_{1,0}$	$A_{2,0}+A_{3,0}$	$A_{3,0}$	$B_{0,0}+B_{1,0}+B_{2,0}+B_{3,0}$	$B_{1,0}$	$B_{2,0}+B_{3,0}$	$B_{3,0}$
$A_{0,1}+A_{1,1}+A_{2,1}+A_{3,1}+B_{0,1}+B_{1,1}+B_{2,1}+B_{3,1}$	$A_{1,1}$	$A_{2,1}+A_{3,1}$	$A_{3,1}$	$B_{0,1}+B_{1,1}+B_{2,1}+B_{3,1}$	$B_{1,1}$	$B_{2,1}+B_{3,1}$	$B_{3,1}$
$A_{0,2}+A_{1,2}+A_{2,2}+A_{3,2}+B_{0,2}+B_{1,2}+B_{2,2}+B_{3,2}$	$A_{1,2}$	$A_{2,2}+A_{3,2}$	$A_{3,2}$	$B_{0,2}+B_{1,2}+B_{2,2}+B_{3,2}$	$B_{1,2}$	$B_{2,2}+B_{3,2}$	$B_{3,2}$
$A_{0,3}+A_{1,3}+A_{2,3}+A_{3,3}+B_{0,3}+B_{1,3}+B_{2,3}+B_{3,3}$	$A_{1,3}$	$A_{2,3}+A_{3,3}$	$A_{3,3}$	$B_{0,3}+B_{1,3}+B_{2,3}+B_{3,3}$	$B_{1,3}$	$B_{2,3}+B_{3,3}$	$B_{3,3}$
Sub-Matrix (1,0)				**Sub-Matrix (1,1)**			
$C_{0,0}+C_{1,0}+C_{2,0}+C_{3,0}+D_{0,0}+D_{1,0}+D_{2,0}+D_{3,0}$	$C_{1,0}$	$C_{2,0}+C_{3,0}$	$C_{3,0}$	$D_{0,0}+D_{1,0}+D_{2,0}+D_{3,0}$	$D_{1,0}$	$D_{2,0}+D_{3,0}$	$D_{3,0}$
$C_{0,1}+C_{1,1}+C_{2,1}+C_{3,1}+D_{0,1}+D_{1,1}+D_{2,1}+D_{3,1}$	$C_{1,1}$	$C_{2,1}+C_{3,1}$	$C_{3,1}$	$D_{0,1}+D_{1,1}+D_{2,1}+D_{3,1}$	$D_{1,1}$	$D_{2,1}+D_{3,1}$	$D_{3,1}$
$C_{0,2}+C_{1,2}+C_{2,2}+C_{3,2}+D_{0,2}+D_{1,2}+D_{2,2}+D_{3,2}$	$C_{1,2}$	$C_{2,2}+C_{3,2}$	$C_{3,2}$	$D_{0,2}+D_{1,2}+D_{2,2}+D_{3,2}$	$D_{1,2}$	$D_{2,2}+D_{3,2}$	$D_{3,2}$
$C_{0,3}+C_{1,3}+C_{2,3}+C_{3,3}+D_{0,3}+D_{1,3}+D_{2,3}+D_{3,3}$	$C_{1,3}$	$C_{2,3}+C_{3,3}$	$A_{3,3}$	$D_{0,3}+D_{1,3}+D_{2,3}+D_{3,3}$	$D_{1,3}$	$D_{2,3}+D_{3,3}$	$D_{3,3}$

In addition to the optimization strategies recommended for parallel reduction in previous works (Lungu, Petrosanu & Pirjan, 2012; Martín, Ayuso, Torres & Gavilanes, 2012), two other optimization techniques have been implemented in our parallelization that greatly improves the device performance. The first of them being the allocation of host arrays directly in page-locked memory of the host using cudaMallocHost() to help for higher bandwidth between device and host. The advantage of using pinned memory is twofold. It also helps to overlap data transfer with computation on host which is the second major optimization technique followed in our approach (Harris, 2012). Data transfer and computation operations should occur in different, non-default streams in order for them to be overlapped. Hence it becomes mandatory to work with multiple streams instead of a single default stream. Creation of multiple streams is made possible with cudaStreamCreate() function and cudaStream_t structure variable. After the creation of multiple streams, the blocking or synchronous data transfer function cudaMemcpy() has been replaced with its asynchronous counterpart cudaMemcpyAsync(). While the second stream of data is being transferred, the kernel call being asynchronous can proceed with computation of data in the first stream. Also cudaStreamSynchronize(), a less severe method of synchronizing the host with a stream is used to synchronize the host code with operations in a stream.

Our implementation employs four streams to read a *sizeXsize* sub-matrix and copy it to GPU memory. Each stream comprises of *(size/nstreams)size* elements. Once the transfer of first stream of *(size/nstreams)Xsize* data is complete, transfer of second stream of *(size/nstreams)Xsize* data and kernel execution with first stream of data are performed in tandem (see Figure 8).

Initially columns belonging to various blocks are reduced and their results stored in a *(size/blocksize)Xsize* matrix that is further reduced to *1Xsize* 1-D array in the next level (Figure 9).

Figure 8. Overlapping data transfer and computation

```
const int nstreams=4, streamsize=size/nstreams;
cudaStream_t stream[nstreams];
cudaMallocHost((void**)&a,size*size*sizeof(double));
//read 'a' matrix
for(int i=0;i<nstreams;i++)
  cudaStreamCreate(&stream[i]);
for(int i=0;i<nstreams;i++)
{
 int offset=i*streamsize*size;
 cudaStatus = cudaMemcpyAsync(&dev_a[offset], &a[offset], size*streamsize *
 sizeof(double), cudaMemcpyHostToDevice,stream[i]);
 if (cudaStatus != cudaSuccess)
 {
   fprintf(stderr, "cudaMemcpy failed!");
   goto Error;
 }
}
for(int i=0;i<nstreams;i++)
 cudaStreamSynchronize(stream[i]);
for(int i=0;i<nstreams;i++)
{
  parallelreduction<<<dimgrid,dimblock,0,stream[i]>>>(dev_a,i);
}
for(int i=0;i<nstreams;i++)
 cudaStreamSynchronize(stream[i]);
```

2.1.2.3. Transpose Computation

For transpose generation of a given matrix the task to be performed is to simply interchange the elements along rows and columns. However, performing this task for a reasonably large matrix requires considerable amount of time and hence transpose computation has also been parallelized. As done for matrix type identification and norm computation, the original matrix of order *fullsizeXfullsize* is partitioned into several small sub-matrices of order s*izeXsize.* This will yield (f*ullsize/size) X* (fu*llsize/size)* unique sub-matrices. Every sub-matrix is in turn partitioned into (s*ize/blocksize) X (size/blocksize) b*locks with bl*ocksizeXblocksize e*lements in each block. Transpose of a single sub-matrix is formed by interchanging the elements within the individual blocks followed by interchanging the blocks themselves along rows and columns. In a similar way, transpose is found for all the sub-matrices. While writing these transposed sub-matrices back to the file, sub-matrices along the rows and columns should be interchanged and written.

Following a similar assumption made earlier with f*ullsize* = 16, s*ize* = 4 and b*locksize* = 2, Figures 10 and 11 show the first level partitioning of the original matrix of f*ullsizeXfullsize* into 4 sub-matrices of order s*izeXsize* and the second level partitioning of a single sub-matrix into 4 smaller blocks of order b*locksizeXblocksize* respectively. Figure 10 also portrays the 4 individual transposed blocks and the interchange of the blocks along rows and columns. Figure 11 depicts the 4 transposed sub-matrices and the final matrix obtained by interchanging the sub-matrices along rows and columns. Figure 12 shows transposition of individual blocks and their interchange.

Figure 9. Parallel reduction in multiple levels

```
__device__ double da[size][size],global_colsum[gridsize][size]={0};
__global__ void columnreduction(int off)
{ __shared__ double temp[blocksize][blocksize];
 int i,j;
 i=(blockIdx.y)*blockDim.y+threadIdx.y;
 j=(blockIdx.x+off*(gridsize/4))*blockDim.x+threadIdx.x;
 int tx=threadIdx.y,f
 int ty=threadIdx.x;
 if(tx<blockDim.x&&ty<blockDim.y)
   temp[tx][ty]=da[i][j];
 __syncthreads();
 if(tx<blockDim.x&&ty<blockDim.y)
 {
  for(int offset=blockDim.x/2;offset>0;offset>>=1)
  {
        if(tx<offset)
        {
          temp[tx][ty]+=temp[tx+offset][ty];
        }
        __syncthreads();
  }
 }
 __syncthreads();
 if(tx==0&&ty<blockDim.y)
   global_colsum[blockIdx.y][j]=temp[tx][ty];
 __syncthreads();
}
```

Now the initial inverse is computed as $V_0 = \frac{A^T}{\left(\|A\|_1 \|A\|_\infty\right)}$ for which division of all the elements in the transpose by the product of 1-Norm and Infinity-Norm, is done in parallel.

2.1.3. Matrix Inversion

Out-of-core matrix inversion using matrix multiplication-based iterative method with seventh order of convergence forms the crux of our approach. With a higher rate of converge, matrix inverse can be evaluated with fewer iterations but still with high accuracy.

$$V_{n+1} = \frac{1}{16} V_n \begin{pmatrix} 120I + AV_n(-393I + AV_n(735I + AV_n(-861I + \\ AV_n(651I + AV_n(-315I + AV_n(93I + \\ AV_n(-15I + AV_n))))))) \end{pmatrix}$$

n = 0,1,2,…

From the equation it is very obvious that the matrix inverse computation has been reduced to a series of matrix multiplication operations. In the very beginning, product of the original matrix and its initial inverse (AV_n) should be computed. At every step this product should be multiplied with the intermediate matrix computed at that step. A total of 9 such multiplications are to be performed in the series.

Figure 10. Partitioning of a sub-matrix into four blocks and its transpose computation

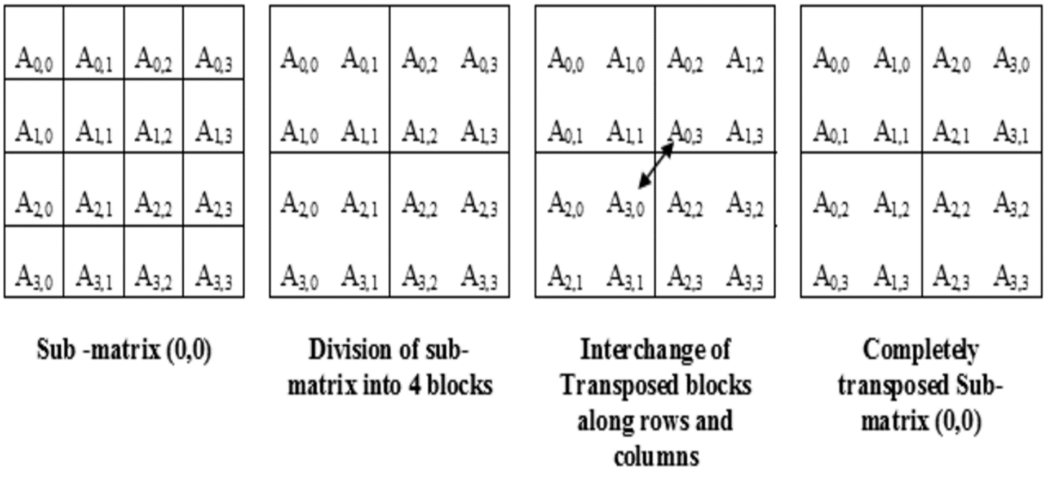
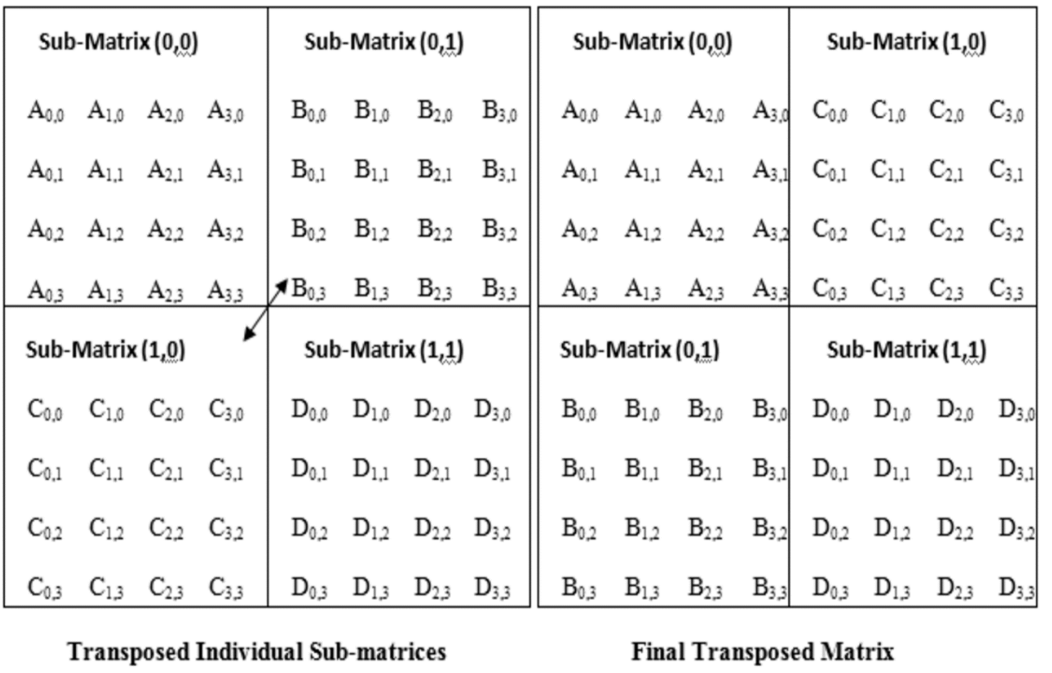


Figure 11. Interchange of transposed sub-matrices to form a complete transposed matrix



2.1.3.1. Block Matrix Multiplication Approach

Block partitioning technique followed in the previous stages is used here too and hence a block matrix multiplication approach is seemingly the right choice. Consider the size of the original matrix and the initial inverse matrix I_initial to be n*X*n where n=2k. For large sizes, both the matrices are partitioned into several small blocks each of size k*X*k. By this way of partitioning, matrices A and I_initial can be represented as a combination of four smaller matrices as given below.

Figure 12. Transpose of individual blocks and their interchange

```
__global__ void transpose(double *trans, int off)
{
  __shared__ double temp_in[blocksize][blocksize], temp_output[blocksize][blocksize];
  int i,j;
  i=(blockIdx.y)*blockDim.y+threadIdx.y;
  j=(blockIdx.x+off*(gridsize/4))*blockDim.x+threadIdx.x;
  int tx=threadIdx.y;
  int ty=threadIdx.x;
  if(ty<blockDim.x&&tx<blockDim.y)
    temp_in[tx][ty]=da[i][j];
    syncthreads();
  if(ty<blockDim.x&&tx<blockDim.y)
    temp_output[ty][tx]=temp_in[tx][ty];
  __syncthreads();
  if(ty<blockDim.x&&tx<blockDim.y)
   trans[j*size+i]=temp_output[ty][tx];  // for interchanging the blocks
  __syncthreads();

}
```

$$\text{A}=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ I\_initial}=\begin{pmatrix} I\_ini_{11} & I\_ini_{12} \\ I\_ini_{21} & I\_ini_{22} \end{pmatrix}$$

Block-matrix multiplication approach is followed to evaluate the product of these two matrices as

$$\text{Product C}=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}=\begin{pmatrix} A_{11}I\_ini_{11}+A_{12}I\_ini_{21} & A_{11}I\_ini_{12}+A_{12}I\_ini_{22} \\ A_{21}I\_ini_{11}+A_{22}I\_ini_{21} & A_{21}I\_ini_{12}+A_{22}I\_ini_{22} \end{pmatrix}$$

where every element A_{ij} or B_{ij} is itself a submatrix of size k*X*k. Partitioning is continued till k equals s*ize*. Multiplication of two matrices of order s*izeXsize* and multiplication of identity matrix by a constant are done in parallel. Same multiplication strategy is applied for the entire multiplication chain proposed in Soleymani's method (see Figure 13).

3. RESULTS AND DISCUSSION

3.1. Performance Evaluation of the Proposed Parallel Iterative Algorithm

Every subtask of the algorithm has been parallelized and designed to work for double-precision floating-point arithmetic. Performance of our approach has been tested for various matrix sizes in a GPU with 96 cores, having a global memory of 1024 MB and shared memory size of 49152 Bytes per block. Time taken for computing matrix norm and transpose have been recorded and shown in Table 1. Also for the same operation, time taken by CPU without offloading the workload to GPU has been measured and given in Table 1 that aids in evaluating the speed up achieved.

Figures 14 and 15 show the individual plot of tabulated values for matrix norm and matrix transpose, using logarithmic scale. In case of matrix norm computation, 5 to 6 times speedup has been achieved for a *100,000X100,000* size matrix using a single device. Likewise matrix transpose computation has exhibited 4 to 5 times increase in the speed up.

3.2. Comparison with Existing CUDA Algorithm for Gauss Jordan Method

Girish Sharma et al. (2013) have developed a fast parallel Gauss Jordan algorithm for Matrix Inversion using CUDA. In their work, although they have proved that GPU based parallelization for matrix inversion is orders of magnitude faster than CPU based parallelization, they have failed to orchestrate a parallel algorithm that would process a matrix too large to be handled directly by the available device memory under consideration. Also the results shown pertain to single precision floating-point

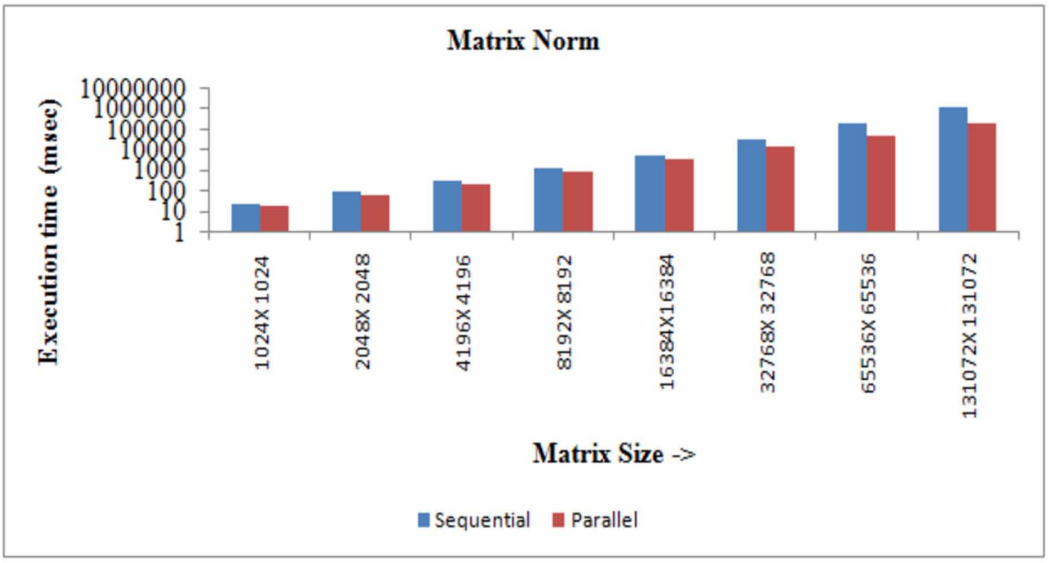
Figure 13. Block matrix multiplication

```
multiply(float *a, float *b, float *c)
{  int i,j,k;
   i=blockIdx.y*blockDim.y+threadIdx.y;
   j=blockIdx.x*blockDim.x+threadIdx.x;
   if(i<size && j<size) {
    c[i][j]=0;
    for(k=0;k<size;k++) {
     c[i][j]+=a[i][k]*b[k][j];
    }
   }
   __syncthreads();
}
__device__ void temp_multiply(const int con)
{  int i,j;
   i=blockIdx.y*blockDim.y+threadIdx.y;
   j=blockIdx.x*blockDim.x+threadIdx.x;
   if(i==j && i<size && j<size)
        temp[i][i]=con*iden[i][i];
   __syncthreads();
}
```

Table 1. Sequential and Parallel Execution time for Matrix Norm and Matrix Transpose

Matrix Norm	Matrix Size	*512X 512*	*1024X 1024*	*2048X 2048*	*4096X 4096*	*8192X 8192*	*16384X16384*	*32768X 32768*
	Sequential (msec)	5	20	80	296	1237	4885	28959
	Parallel (msec)	4.12	15.23	51.20	197.77	782.51	3134.96	12762
Matrix Transpose	Matrix Size	*512X 512*	*1024X 1024*	*2048X 2048*	*4096X 4096*	*8192X 8192*	*16384X16384*	*32768X 32768*
	Sequential (msec)	5	23	92	373	1542	5907	25304
	Parallel (msec)	0.98	4.86	19.22	77.26	308.23	1231.51	5229.78

Figure 14. Execution Time - Matrix Norm



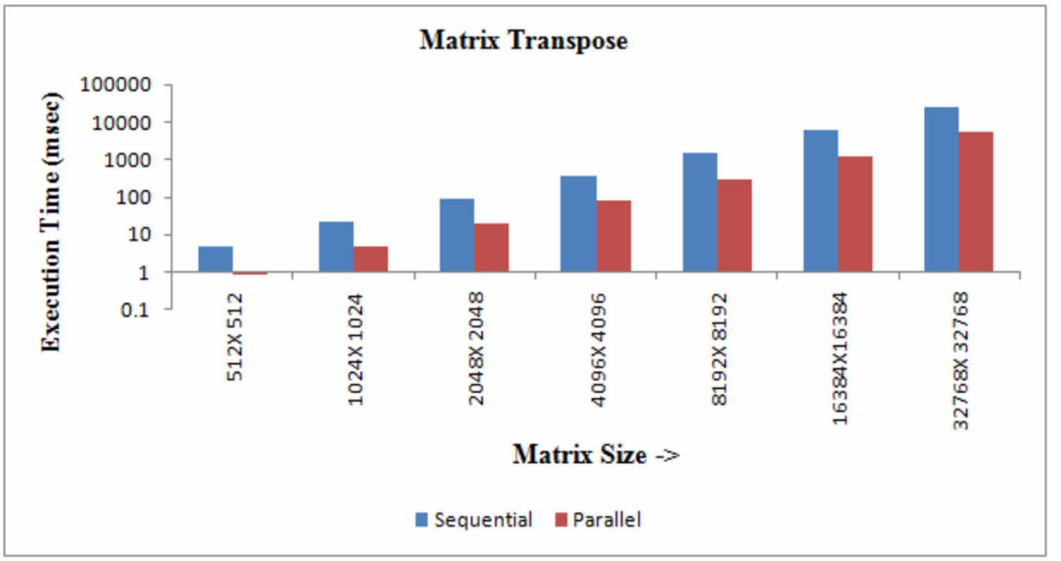
arithmetic but in reality most of the scientific and engineering applications necessitate computations with much higher precision. Hence to establish a similar test environment for comparison with the proposed parallel algorithm, in our work CUDA implementation of Gauss Jordan method also has been done to work for double precision floating-point arithmetic and to support out-of-core matrix inversion.

3.2.1. CUDA Implementation of Gauss-Jordan Method

Generally, Gauss-Jordan method proceeds by first augmenting the original matrix with the identity matrix and then performing the following two transformations repeatedly for all rows.

- $R_i \leftarrow R_i / a_{ii}$
- $\forall R_j$ where $j \neq i$, $R_j \leftarrow R_j - R_i \; X \; a_{ji}$

Figure 15. Execution Time - Matrix Transpose



The existing work has parallelized both the transformations individually but only for the matrix size supported by the GPU architecture considered for the study. But our implementation handles out-of-core computation of matrix inverse (Figure 16).

Our proposed approach first identifies the row with a row-number equal to iteration count, *i;* divides the active row of the actual sub-matrix and the augmented matrix by the diagonal element of the active row; processes the other rows in such a way that elements along the same column as that of the diagonal element reduce to zero; retains the augmented matrix and sends the modified sub-matrix back to the CPU for all iterations except the last; sends the augmented sub-matrix in the last iteration. The code is executed for all the sub-matrices of the original matrix (Figure 17).

The proposed parallel iterative algorithm and parallel Gauss-Jordan algorithm have been implemented to perform double-precision floating-point arithmetic and tested using the same GPU platform. Execution times of the two methods for matrices of sizes *512X512* and *1024X1024* have been tabulated in Table 2. Execution time of the proposed parallel iterative algorithm includes the time needed for computation of norm, transpose and then the inverse. It is observed from Figure 18 that the proposed parallel algorithm requires approximately only half of the time needed for executing parallel Gauss-Jordan algorithm.

CONCLUSION

The embarrassingly parallel nature of the algorithm naturally fits the Throughput-oriented architecture of GPUs. Experimental results have shown 5X speedup of the sub-tasks for matrices of order 10^5X10^5 by employing GPUs. Thus, the entire process of matrix inversion is accelerated manifold without any setback in the accuracy of the results. To facilitate the comparison of our parallel iterative approach with a direct method of matrix inversion in a similar test environment, Gauss-Jordan algorithm has been parallelized too. Comparative study has shown that the proposed parallel iterative algorithm is twice as fast as the parallel Gauss-Jordan algorithm for out-of-core matrix inversion and this acceleration is attributed to the division-free design of the algorithm.

Figure 16. Reading of input and storage of intermediate results being altered between two files

```
Assuming the matrix to be inverted to be stored initially in File 1, read the original matrix of
order fullsizeXfullsize from File1
Partition it into multiple sub-matrices of order nr_rowsXfullsize where nr_rows depends on the
blocksize

Repeat fullsize times
{
  Assign the current iteration count i to row-number
  If i is even
    Readfile=File 1
    Writefile=File 2
  Else
    Readfile=File 2
    Writefile=File 1
  End If
  Identify the sub-matrix in which the row Ri resides
  Read the sub-matrix from Readfile into the device
  Divide Ri by Rii
  LOOP:
  ? Rj in the sub-matrix where j ? i, compute Rj as Rj - Ri X aji
  Write this sub-matrix into Writefile
  Read the next sub-matrix from Readfile
  Repeat LOOP till all the sub-matrices in Readfile are processed
}
```

Every subtask of the algorithm has been parallelized in a way that works by splitting up huge matrices into several smaller blocks/tiles that can be supported by a specific GPU and iterating the kernel for all these blocks. This serves to alleviate to a great extent, the restriction posed by the limited device memory size, over the size of the matrices that can be handled. Multiple devices can be employed to enhance the performance further. As the algorithm has been designed to be highly scalable, with minimal efforts it can be ported to multiple GPU architecture in which case the algorithm with its embarrassingly parallel nature will be a boon for the big data climate research community.

Figure 17. Processing of actual sub-matrix and augmented matrix

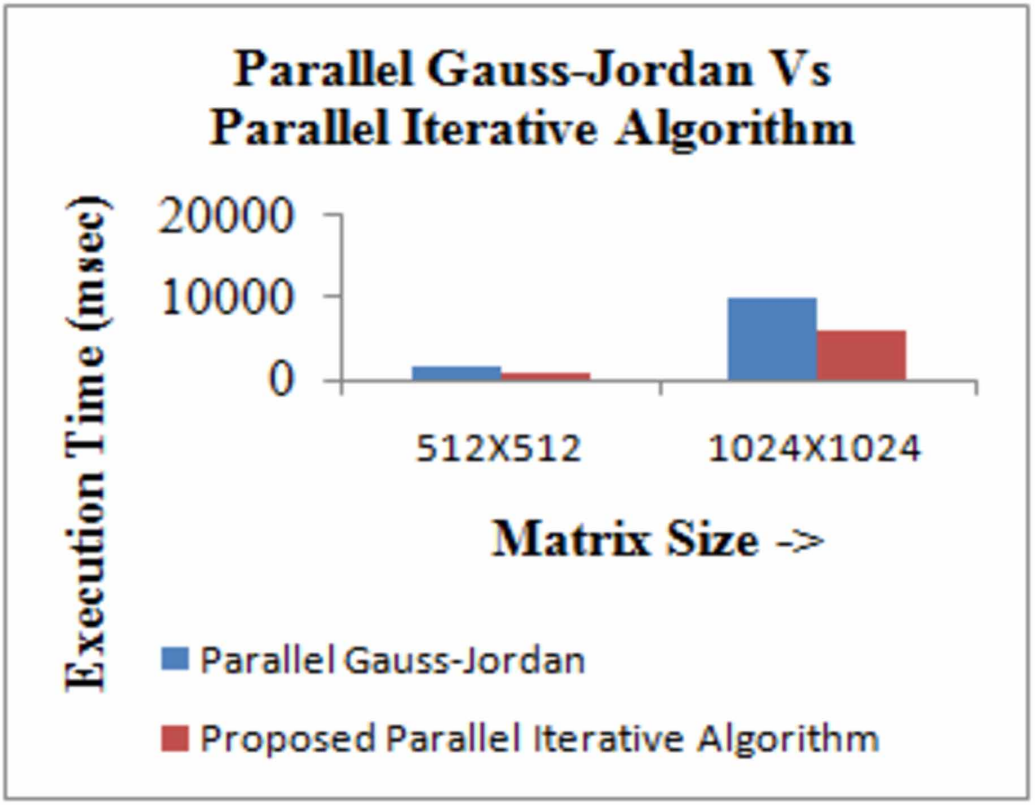
```
__device__ double da[nr_rows][fullsize], augment[nr_rows][fullsize];
__device__ double active_row[fullsize],active_augment_row[fullsize];
__global__ void gauss_jordan(double *augment_output, int off, int *r,int row_nrr)
{ int i,j;

 i=(off * gridDim.x+blockIdx.x)/n_partitions;
j=((off * gridDim.x+blockIdx.x)%n_partitions)*blockDim.x+threadIdx.x;
  int actual_row_number=*r*nr_rows+(off*gridDim.x+blockIdx.x)/n_partitions;
  if(actual_row_number==row_nrr&&j<fullsize)
  {
          active_row[j]=da[i][j];
          active_augment_row[j]=augment[i][j];
  }
  __syncthreads();
  if(actual_row_number==row_nrr&&j<fullsize)
  {
          da[i][j]/=active_row[row_nrr];
          augment[i][j]/=active_row[row_nrr];
  }
  __syncthreads();
   if(actual_row_number!=row_nrr&&j<fullsize)
   {
         da[i][j]=da[i][j]-da[i][row_nrr]*active_row[j];
         augment[i][j]=augment[i][j]-da[i][row_nrr]*active_augment_row[j];
   }
  __syncthreads();
  if(row_nrr==fullsize-1)  //while working for the last row alone, send the
                           //augmented matrix to the CPU
  {
          if(i<nr_rows&&j<fullsize)
          augment_output[i*fullsize+j]=augment[i][j];
          __syncthreads();
  }
  else                  // for the remaining rows, send only the actual matrix
                        //which is getting updated
  {
          if(i<nr_rows&&j<fullsize)
          augment_output[i*fullsize+j]=da[i][j];
          __syncthreads();
  }
  __syncthreads();
}
```

Table 2. Execution Time of Parallel Gauss-Jordan Vs Parallel Iterative Algorithm

Matrix Size	*512X512*	*1024X1024*
Parallel Gauss-Jordan Algorithm (In msec)	1432.14	10101.23
Proposed Parallel Iterative Algorithm [norm + transpose + inverse] (In msec)	719.99	5849.57

Figure 18. Parallel Gauss-Jordan Vs Parallel Iterative algorithm



REFERENCES

Agarwal, M., & Mehr, R. (2014). Review of matrix decomposition techniques for signal processing applications. *Int. J. Eng. Res. Appl*, *4*(1), 90–93.

Arefin, A. S., Riveros, C., Berretta, R., & Moscato, P. (2012, July). Computing large-scale distance matrices on GPU. In *Proceedings of the 2012 7th International Conference on Computer Science & Education (ICCSE)* (pp. 576-580). IEEE. doi:10.1109/ICCSE.2012.6295141

Benner, P., Ezzatti, P., Quintana-Ortí, E. S., & Remón, A. (2009, August). Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function. In *Proceedings of the European Conference on Parallel Processing* (pp. 132-139). Springer Berlin Heidelberg.

Chang, F. C. (2015). Inverse and Determinant of a Square Matrix by Order Expansion and Condensation. *IEEE Antennas & Propagation Magazine*, *57*(1), 28–32. doi:10.1109/MAP.2015.2401792

Chrzeszczyk, A., & Chrzeszczyk, J. (2013). Matrix computations on the GPU CUBLAS and MAGMA by example. Retrieved 16.09. 2015 from https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf

Dziekonski, A., Lamecki, A., & Mrozowski, M. (2011). A memory efficient and fast sparse matrix vector product on a GPU. *Progress in Electromagnetics Research*, *116*, 49–63. doi:10.2528/PIER11031607

Ezzatti, P., Quintana-Orti, E. S., & Remon, A. (2011b, February). High performance matrix inversion on a multi-core platform with several GPUs. In *Proceedings of the 2011 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (pp. 87-93). IEEE. doi:10.1109/PDP.2011.66

Ezzatti, P., Quintana Ortí, E. S., & Remón Gómez, A. (2011a). Using graphics processors to accelerate the computation of the matrix inverse.

Galoppo, N., Govindaraju, N. K., Henson, M., & Manocha, D. (2005, November). LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (p. 3). IEEE Computer Society. doi:10.1109/SC.2005.42

Gao, J., Qi, P., & He, G. (2016). Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU. *Mathematical Problems in Engineering*.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., & Volkov, V. et al. (2008). Parallel computing experiences with CUDA. *IEEE Micro*, *28*(4), 13–27. doi:10.1109/MM.2008.57

Haidar, A., Abdelfatah, A., Tomov, S., & Dongarra, J. (2017, February). High-performance Cholesky factorization for GPU-only execution. In *Proceedings of the General Purpose GPUs* (pp. 42-52). ACM. doi:10.1145/3038228.3038237

Haidar, A., Dong, T., Luszczek, P., Tomov, S., & Dongarra, J. (2015). Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, *29*(2), 193–208. doi:10.1177/1094342014567546

Harris, M. (2012). How to overlap data transfers in cuda c/c++. Retrieved from http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc

Helfenstein, R., & Koko, J. (2012). Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics*, *236*(15), 3584–3590. doi:10.1016/j.cam.2011.04.025

Ibeid, H., Kaushik, D., Keyes, D., & Ltaief, H. (2011). Toward accelerating the matrix inversion computation of symmetric positive-definite matrices on heterogeneous GPU-based systems. In *Proceedings of the Student Research Symposium, International High Performance Computing Conference (HiPC)*.

Jamil, N. (2012). A comparison of direct and indirect solvers for linear systems of equations. *International Journal of Emerging Sciences*, *2*(2), 310–321.

Kijsipongse, E., Suriya, U., Ngamphiw, C., & Tongsima, S. (2011, May). Efficient large Pearson correlation matrix computing using hybrid MPI/CUDA. In *Proceedings of the 2011 Eighth International Joint Conference on Computer Science and Software Engineering* (pp. 237-241). IEEE.

Koza, Z., Matyka, M., Szkoda, S., & Mirosław, Ł. (2014). Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, *36*(2), C219–C239. doi:10.1137/120900216

Lungu, I., Petrosanu, D. M., & Pirjan, A. (2012). Optimization Solutions for Improving the Performance of the Parallel Reduction Algorithm Using Graphics Processing Units. *Informações Econômicas*, *16*(3), 72.

Martin, P. J., Ayuso, L. F., Torres, R., & Gavilanes, A. (2012, July). Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In *Proceedings of the 2012 International Conference on High Performance Computing and Simulation* (pp. 511-519). IEEE. doi:10.1109/HPCSim.2012.6266966

Miyoshi, T., Kunii, M., Ruiz, J., Lien, G. Y., Satoh, S., Ushio, T., & Ishikawa, Y. et al. (2016b). "Big Data Assimilation" revolutionizing severe weather prediction. *Bulletin of the American Meteorological Society*, *97*(8), 1347–1354. doi:10.1175/BAMS-D-15-00144.1

Miyoshi, T., Lien, G. Y., Satoh, S., Ushio, T., Bessho, K., Tomita, H., & Gerofi, B. (2016a). "Big Data Assimilation" Toward Post-Petascale Severe Weather Prediction: An Overview and Progress. *Proceedings of the IEEE*, *104*(11), 2155–2179. doi:10.1109/JPROC.2016.2602560

Playne, D. P., & Hawick, K. A. (2015). Benchmarking multi-GPU communication using the shallow water equations. *International Journal of Big Data Intelligence*, *2*(3), 157–167. doi:10.1504/IJBDI.2015.070596

Prabhu, H., Rodrigues, J., Edfors, O., & Rusek, F. (2013, April). Approximative matrix inverse computations for very-large MIMO and applications to linear pre-coding systems. In *Proceedings of the Wireless Communications and Networking Conference (WCNC)* (pp. 2710-2715). IEEE. doi:10.1109/WCNC.2013.6554990

Salim, M., Akkirman, A. O., Hidayetoglu, M., & Gurel, L. (2015, July). Comparative benchmarking: Matrix multiplication on a multicore coprocessor and a GPU. In *Proceedings of the Computational Electromagnetics International Workshop (CEM)*. IEEE. doi:10.1109/CEM.2015.7237429

Sharma, G., Agarwala, A., & Bhattacharya, B. (2013). A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. *Computers & Structures*, *128*, 31–37. doi:10.1016/j.compstruc.2013.06.015

Soleymani, F. (2012). A rapid numerical algorithm to compute matrix inversion. *International Journal of Mathematics and Mathematical Sciences*.

Soleymani, F. (2013). On a fast iterative method for approximate inverse of matrices. *Communications of the Korean Mathematical Society*, *28*(2), 407–418. doi:10.4134/CKMS.2013.28.2.407

Topa, T. (2015). Efficient out-of-GPU memory strategies for solving matrix equation generated by method of moments. *Electronics Letters*, *51*(19), 1542–1544. doi:10.1049/el.2015.2175

Wu, C. C., Ke, J. Y., Lin, H., & Jhan, S. S. (2014). Adjusting thread parallelism dynamically to accelerate dynamic programming with irregular workload distribution on GPGPUs. *International Journal of Grid and High Performance Computing*, *6*(1), 1–20. doi:10.4018/ijghpc.2014010101

Xingbo, W. (2011, October). A rank-reducing and division-free algorithm for inverse of square matrices. In *Proceedings of the 2011 International Workshop on Open-Source Software for Scientific Computation (OSSC)* (pp. 17-21). IEEE. doi:10.1109/OSSC.2011.6184687

Ylinen, M., Burian, A., & Takala, J. (2003, November). Updating matrix inverse in fixed-point representation: Direct versus iterative methods. In *Proceedings of the International Symposium on System-on-Chip* (pp. 45-48). IEEE.

Zhou, T., Fang, S., Yang, X., Li, Z., Guo, Q., & Jiang, B. (2012, October). A Jacobi-based parallel algorithm for Matrix inverse computations. In *Proceedings of the 2012 International Conference on Wireless Communications & Signal Processing (WCSP)*. IEEE. doi:10.1109/WCSP.2012.6542793

Zouaneb, I., Belarbi, M., & Chouarfia, A. (2016). Multi approach for real-time systems specification: Case study of GPU parallel systems. *International Journal of Big Data Intelligence*, *3*(2), 122–141. doi:10.1504/IJBDI.2016.077385

M. Varalakshmi is an Assistant Professor and Research Scholar in the School of Information Technology and Engineering, VIT University. She received her B.E.(CSE) degree from Madras University and M.Tech. (IT) degree (Gold medalist) from VIT University. Her research focus is primarily in High Performance Computing. She is particularly interested in parallelization of climatic models.

Amit Kesarkar is Scientist SE and Head Weather and Climate Research Group in National Atmospheric Research Laboratory. He obtained his Ph.D. degree from University of Pune, India in the year 2001. His research interest includes weather and climate modelling and high-performance computing on different platforms.

Daphne Lopez is a Professor in the School of Information Technology and Engineering, Vellore Institute of Technology University. Her research spans the fields of grid and cloud computing, spatial and temporal data mining and big data. She has a vast experience in teaching and industry. She is the author/co–author of papers in conferences, book chapters and journals. She serves as a reviewer in journals and conference proceedings. Prior to this, she has worked in the software industry as a consultant in data warehouse and business intelligence. She is a member of International Society for Infectious Diseases.