Computer Architecture Lab Project

Parallelization of Gauss-Jordan Algorithm for Matrix Inversion using CUDA

Mohit Bhasi
16CO126
16co126.mohit@nitk.edu.in

November 4, 2018

1. Introduction

With the onset of Graphics Processing Units in the recent past, computation capabilities have sky rocketed to an all time high. Many trivial algorithms are being made parallel to achieve considerable speed up. A slight limitation to a GPU is the amount of shared memory it contains. This acts a bottle neck to the degree of parallelization. Parallel computations can be performed on both CPUs and GPUs. The number of cores present in a GPU is far more than the number of cores present in a CPU. Therefore, more number of threads can be deployed on a GPU. The use of GPGPU for parallelizing tasks was common until the release of CUDA by Nvidia. CUDA includes faster memory sharing and minimal thread creation overhead.

2. Problem Statement

Matrix inversion serves as a major step in countless number of tasks. Many image processing, image filtering, signal processing and 3D rendering applications are heavily dependent on matrix inversion.

Many methods including, but not only, Strassen, Gaussian elimination, Gauss-Jordan, LU Decomposition, Cholesky Decomposition etc. are available for finding an inverse of a matrix. The Gauss Jordan method is the oldest algorithm devised for the inversion task. The Gauss-Jordan algorithm has a running time of $O(n^3)$. The algorithm also has good scope for parallelization unlike many other advanced techniques.

Strassen'a algorithm was the first to demonstrate a sub cubic run time. The operations needed to find an inverse of a n x n matrix include 7 arithmetic operations and 2 matrix inversions, where the size of the matrix is $n/2$. Since the algorithm itself requires calculation of inversions, the scope of parallelization reduces. Hence even though the run time of the algorithm is less than a cubic polynomial, the speed up achieved by parallelizing the algorithm is going to be minimal.

Years of research lead the running time to reduce to $O(n^{2.37})$. Further reduction seemed cumbersome and very value dependent. For example the Monte Carlo method could invert a Hermitian matrix at a $O(n^{2.125})$ run time but was no where near this time complexity for a general matrix. The scope of parallelizing the whole task seems to be extremely beneficial and the need of the hour. Recent developments in architecture of GPUs has brought scope of reducing run time of the inversion algorithms to less than $O(n^2)$

3. Implementation

3.1 Algorithm

The computation of the inverse matrix begins by augmenting it with an identity matrix of the same size.

$[C] = [A|I]$

Elementary row transformation is performed on matrix C and it is transformed column by column into the unit matrix. This step involves two process, the first being making A_{11} into one by :

$R_i = \frac{R_i}{A_{ii}}$

If A_{ii} is zero, then any non zero row is picked and added to the i^{th} row before applying the above step. The second step is needed to reduce the elements of j^{th} column to zero, this can be done by :

$R_i = R_i - R_j \text{ x } A_{ij}$

After one round of transformation, the first column is augmented and the matrix C now looks like :

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

The above two steps need to be applied to each column to obtain the inverse of matrix A

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

The run time without any parallelization of the above algorithm is $O(n^3)$ as each of the two steps run at $O(n)$ and these two steps needs to be repeated n times.

In the parallel approach, the Steps 1 and 2 are performed together. For step 1, n threads are created to process the whole row and for step 2, n x $(n-1)$ threads are needed to convert the rest of the column to zeros. The parallel approach hence runs at $O(n^2)$. Further speed up is obtained when shared memory is used.

3.2 Metrics to be measured

The algorithm will be tested with varying the block size in increments of 2. Another thing that will be varied is the type input matrix. The type of input matrix used will be :

1. Dense Matrix
2. Spare Matrix
3. Identity Matrix
4. Band Matrix
5. Hollow Matrix

Algorithm 1 Gauss Jordan Matrix Inversion

```
procedure INVERSION
    $matrix \leftarrow$ Read matrix from file
    $n \leftarrow$ Size of $matrix$
    $j \leftarrow 0$
    while $j <= n$ do
        $k \leftarrow 0$
        for $matrix(k)(j)$ != $0$ do
            $k \leftarrow k + 1$

        Span n threads in 1 block
        for thread $i$ of $n$ in $block(1)$ do
            $matrix(j)(i) \leftarrow matrix(j)(i) + matrix(k)(i)$

        Span n threads in 1 block
        for thread $i$ of $n$ in $block(1)$ do
            $matrix(j)(i) \leftarrow matrix(j)(i)/matrix(j)(j)$

        Span n threads in n blocks
        for thread $i$ of $n$ in $block(r)$ do
            $matrix(i)(r) \leftarrow matrix(i)(r) - matrix(i)(j) * matrix(j)(r)$

        $j \leftarrow j + 1$
```

3.3 Inputs and Outputs

All the input matrices will be randomized depending on the matrix type. The matrix will be stored in a text file and all the data will be generated using Python. Timing calculation will be done using built in CUDA/C functions.

The output, the inverse matrix, will be stored in a text file and all the running time values obtained for different set of inputs will be plotted for analysis using the matplotlib library for Python.

4. Project timeline

October 23rd, 2018

Project topic finalization

October 25th, 2018

Detailed reading of existing research papers in the same topic. References:

1. https://www.sciencedirect.com/science/article/pii/S0045794913002095
2. https://www.thinkmind.org/download.phpusg=AOvVaw2fAKol8KLVcTOxP7AxA3hE

October 28th, 2018

Complete Python script used to generate inputs and a script that plots the results after results are obtained. The output script will be designed with dummy values initially and will not take any real data as input.

November 5th, 2018

Complete C implementation of Gauss-Jordan matrix inversion and obtain resulting graphs for run time. A graph will be drawn for each corresponding set of inputs.

November 12th, 2018

Complete CUDA implementation of Gauss-Jordan matrix inversion and obtain resulting graphs for run time. A graph will be drawn for each corresponding set of inputs.

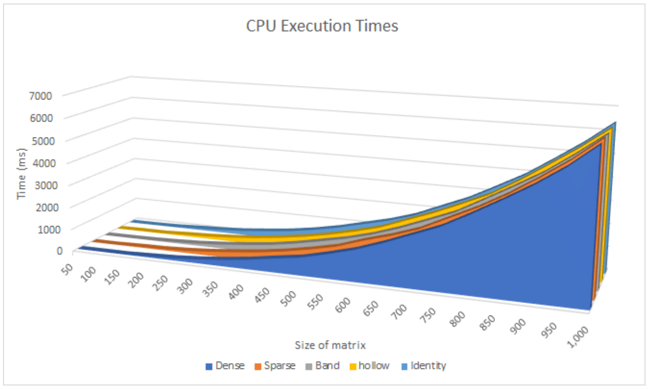
November 13th, 2018

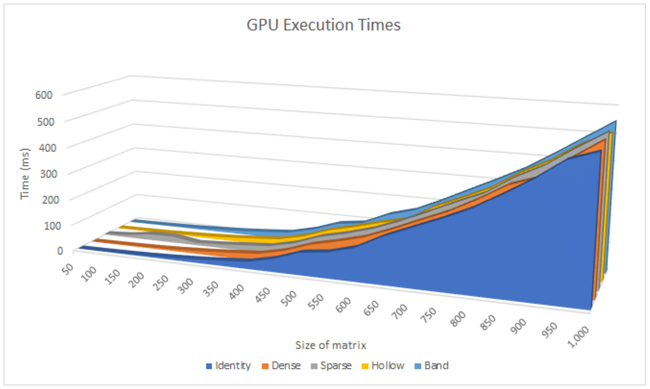
Comparison plots will be drawn between results from CPU implementation and CUDA implementation and speedup conclusion will be drawn

November 14th, 2018

Compilation of report with retrieved data. The day will be mostly spent with documentation. The final report will be drafted and ready for submission.

5. Results





Most general science questions involve using sparse matrices. In a hollow matrix, all the diagonal elements are zero and hence an extra row transformation is needed to fix the diagonal element for each column making it the matrix that needs the most number of computation. For performance and testing, all the matrices were stored in a dense manner.

For confirmation of inversion, the identity matrix generated after inversion was checked.

Hardware used

All execution was done on a system with a Intel i7-7700HQ quad core processor with a clock rate of 2.80GHz. The system was also equipped with a Nvidia GTX1060 GPU. The GPU has a compute compatibility of 6.1 and can perform 120GFLOPS double precision operations.

Observations

The code was run for matrices of size 50-1000 in increments of 50. All the matrices were pre-generated and hence do not play a role in execution time calculation. After plotting the results, it was concluded that the type of matrix used had no significant impact on the execution times. This could be mainly because no optimization was performed in storing non-dense matrices. For example, sparse matrices could be stored in a compressed row form. Doing so will reduce memory utilization in the GPU and could speed up execution time.

That being said, the speedup achieved is phenomenal and the GPU implementation is almost ten times faster. Another thing observed is the growth of the execution times. The CPU algorithm has an exponential growth whereas the GPU implementation has a very small growth rate.

Another experimentation done was changing the block size of the GPU thread-block but this did not give a huge performance boost as hoped. But the said experiment helped conclude that the block size should be set to 8 for the fastest running time. Therefore the block size was fixed to 8 for every other tests conducted

At matrix size n=250 it is observed that the GPU implementation begins displaying quadratic nature due to hardware limitations. The number of threads that can be deployed by the GPU is 64,635. Since a matrix of size n needs n^2 threads, it is obvious that at n=250, about 62,500 threads are needed and all can be deployed at once by the GPU, but after this the thread deployment latency is observed due to warp stacking. This explains the slight quadratic nature of the graph after n=250.

6. Conclusion

The effectiveness of a large number of computational problems are based on the ability to invert large matrices accurately and quickly. GPUs have helped enable the computation of such problems in a massively parallel architecture. The main advantage of GPUs is that the overhead in thread creation is negligible. This project successfully redesigned the Gauss Jordan algorithm for matrix inversion on GPU using Nvidia's CUDA platform. Different types and sizes of matrices were used and results were obtained. For small matrix sizes, the running time of the parallel algorithm is $O(n)$ and the total threads deployed in the GPU is n^2

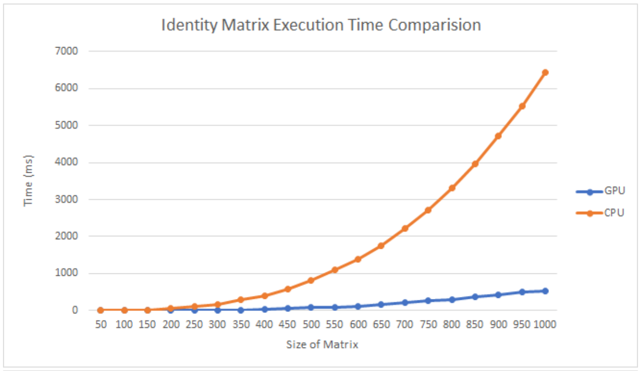
Objectives achieved

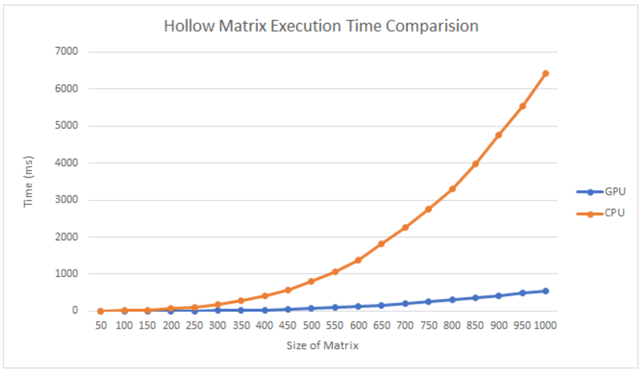
1. Successfully generated random matrix inputs for each type of matrix mentioned
2. Successfully implemented CPU version of Gauss Jordan matrix inversion algorithm using c++
3. Successfully implemented GPU version of Gauss Jordan matrix inversion algorithm using CUDA
4. Successfully achieved execution time graphs for both the implementations

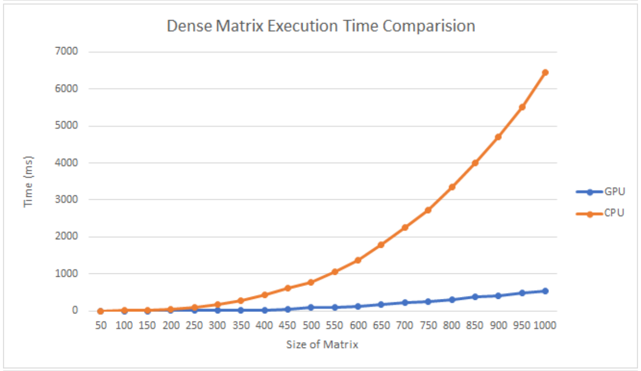
Objectives not achieved

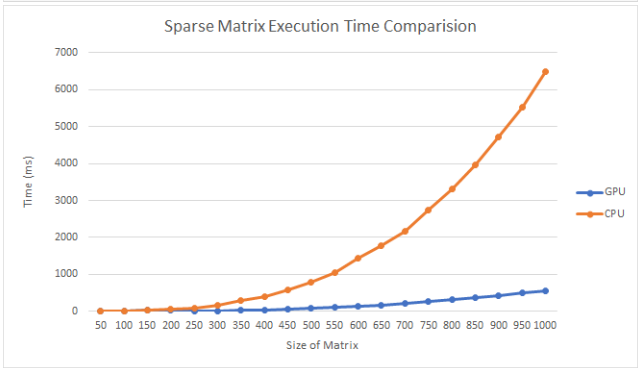
The plan initially was to change the block size in multiples of 2 and see how the execution times varied. On doing this, differences observed were minimal hence the metric was dropped and the block size was set to 8. Additionally, excel was used to perform all the graphing instead of using matplotlib (Python library)

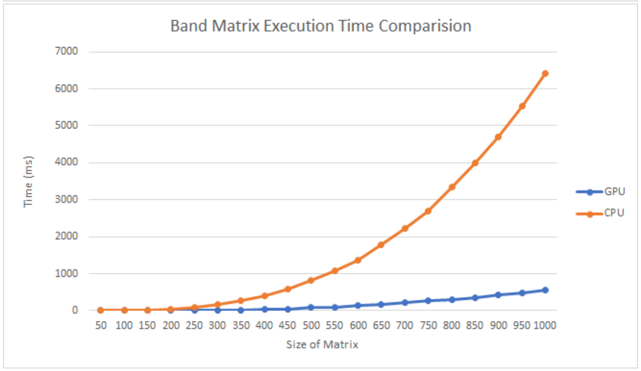
7. GPU vs CPU Comparision Graphs


Identity Matrix Execution Time Comparision
Time (ms)
Size of Matrix
GPU
CPU


Hollow Matrix Execution Time Comparision
Time (ms)
Size of Matrix
GPU
CPU


Dense Matrix Execution Time Comparision
Time (ms)
Size of Matrix
GPU
CPU


Sparse Matrix Execution Time Comparision
Time (ms)
Size of Matrix
GPU
CPU


Band Matrix Execution Time Comparision
Time (ms)
Size of Matrix
GPU
CPU