Computer Architecture Lab Project

Parallelization of Gauss-Jordan Algorithm for Matrix Inversion using CUDA

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

October 25, 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 :

$R_i = R_i - R_j$ 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. Random Matrix
2. Spare Matrix
3. Identity Matrix
4. Band Matrix
5. Hollow Matrix

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 matplot library for Python.

4. Project timeline

October 23rd, 2018

Project topic finalization

October 25th, 2018

Detailed reading of exisiting 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.