PES UNIVERSITY, BENGALURU

COURSE PROJECT – B.TECH. (CSE) – VI SEM

SESSION: JANUARY– MAY, 2020

UE17CS335 – PARALLEL COMPUTING

Project Report

on

"SOLVING MASSIVE LINEAR EQUATIONS USING CUDA"

SUBMITTED BY

HARISH P B	**PES1201701435**
UMASHANKAR	**PES1201701296**
SHREYAS S M	**PES1201700837**

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

PES UNIVERSITY

(ESTABLISHED UNDER KARNATAKA ACT NO. 16 OF 2013)

100 FEET RING ROAD, BENGALURU – 560085, KARNATAKA, INDIA

TABLE OF CONTENTS

INTRODUCTION

The main objective is to efficiently solve massive linear equations in parallel. A system of n homogeneous or non-homogeneous linear system of equations in n variables x_1, x_2,..xn or simply a linear system is a set of n linear equations, each in n variables. The linear system could be represented in matrix form as

$Ax = B$

where A is called the coefficient matrix of order $n \times n$, x is any solution vector of order $n \times 1$ and b is any vector of order $n \times 1$. The most common method to solve simultaneous linear equations is the Gaussian elimination method, the main disadvantage of this is that it is often impractical to solve larger problems on serial systems since it takes a lot of time and money. Hence, parallel methods implement a lot of resources working together to reduce the time and cut down the potential costs.

The two methods implemented as part of this project include
First, Row partitioning solution using openmp and second, implementation of the same on CUDA. Further, the runtimes are analysed from the plot and the speedup is calculated.

LITERATURE SURVEY

[PAPER1: M. Garland et al., "Parallel Computing Experiences with CUDA," in IEEE Micro, vol. 28, no. 4, pp. 13-27, July-Aug. 2008, doi: 10.1109/MM.2008.57.]

With the rise of parallelism and mainstream parallelism practically becoming available at an affordable price, the CPU's today are all parallel processors. The trend drastically shifted from pushing the boundaries of the speed of the processor to how many physical processors could be added onto a chip. It has also been foreshadowed that this trend would also likely continue into the future. As scaling becomes the common issue among many systems today, parallel processors have come to the forefront of efficient processing and enterprise scaling.

For some time now, modern advances in chip technology have allowed Graphical Processing Unit's (commonly known as GPU's) to become the standard of today's definition of chip-level parallelism. The range of cores are from 8 to 240 in the current offerings of NVIDIA's latest architecture which allows for the direct programming of the GPU. GPU's today has evolved from what they were a few years ago to fill the gap needed by the increasing human demand of being able to solve real world applications which has inherent parallelism. This trumps the old method of using Graphics API accelerators for improving the performance of programs.

The CUDA programming model was first created by the company Nvidia and this particular software space was used to let programmers develop efficient and scalable parallel programs worldwide. This was enabled by the various C language extensions that were integrated into the software. Programmers were exposed to the fine-grained parallelism with extensive usage of the multi-threaded programs that were capable of being rendered by the GPU's. This also enhances the scalability in the broader sense that physical parallelism was available within a range of GPU devices. This made it simple for the programmer to define an abstract level of parallelism with the already well-known semantics of the C-programming language. This enabled them to develop massive parallel programming models with relative ease.

Many developers feel that CUDA has achieved the main objective of simplifying the process of making parallel programming mainstream and with the simple practices that come along with it.

PAPER2: [Using gauss - Jordan elimination method with CUDA for linear circuit equation systems Nesrin Aydin Atasoya *, Baha Senb , Burhan Selcukc]

The system of linear equations are the main engineering problems that challenge the engineering community. The n x n matrices that are described in the above paper are carefully studied and using a LCES or a Linear Circuit Equation System and CUDA (Compute Unified Device Architecture in order to achieve the goal. The LCES model uses the GPU architecture without relying heavily on the system's internal CPU architecture in order to come up with a suitable solution. The CUDA architecture was developed by a company called NVIDIA. Some of the factors which the Linear Circuits depend upon are impedance, resistance, dependent-independent sources of current, capacitance, DC ad AC voltage sources. The authors have conducted this study to find the solution of circuits that include several factors such as independent current sources, resistance and DC voltage.

This is done because e circuit analysis frequently involves the solution of a system of linear equations. One of the methods used to approach this problem is the Gauss-Jordan Elimination method (which is a derivative of the Gaussian Elimination method) of calculating the upper triangular matrix and the subsequent back substitution of the rows after performing the relevant row operations in order to find the value of the variables and to find the solution to the augmented matrix. This is generally used to solve the system of linear equations of the form Ax = B. The algorithm focusses on the arrival of the echelon form to reduce the rows by performing suitable elementary row operations. The Gauss-Jordan Elimination method involves two main parts – Forward Elimination to reduce a given system to its triangular form and the Back substitution to find the solution of said system. The second step is usually not seen as an effective step for a parallel programming model because of the fact that the number of elements of column matrix which are unknown are dependent on each other and it is in this inter-dependency that parallel programming models fail to take into account.

This is different from the Normal Gaussian Elimination and is rightfully the preferred option. The CPU simply fails to match up to the speed at which the system of linear equations can be solved on a modern-day GPU.

PAPER3: [Z. Zhang, Q. Miao and Y. Wang, "CUDA-Based Jacobi's Iterative Method," 2009 International Forum on Computer Science-Technology and Applications, Chongqing, 2009, pp. 259-262, doi: 10.1109/IFCSTA.2009.68.]

The solution to a system of linear equations is one of the most seasoned problems in the field of computer science engineering. Solving a system of linear equations and the process related to it is of great significance in today's world where modern day style of GPU's dominates the market and are very capable of performing high intensity processes such that they involve many cores and are fit for large scale parallel computing. This provides us with a new way for accelerating the process of solving the system of linear equations using a graphical processing unit based on Jacobi's iterative method is presented in the above paper. We can then introduce the backgrounds for accelerating the solving of linear equations together with the graphical processing units the corresponding parallel platform CUDA as a software architecture on top of that. We can then implement the Jacobi's iterative method based on this architecture developed by the company NVIDIA. We can then compare the results of the experiment based on the coder programs with the GPU when compared with traditional programs on an internal CPU.

The experiment results show that the new method obtains a speedup of approximately 59 times with a low precision of a single floating point Additionally it does have 19 times with a double at a high precision.

A single CPU is normally limited because of many factors such as the area on the chip as well as the consumption of power. Nowadays growth in these areas we can physically encounter any bottlenecks and thus hinders the improvement for performance of the CPU. This benefits from the multicore architecture of set CPU. Mainstream multicore CPUs usually have around two to four cores and these are used commonly and will continue to be used in future generations.

METHODOLOGY AND IMPLEMENTATION

Gaussian Elimination -

First let us see how Gaussian elimination is performed, the Gaussian elimination method is used to solve system of linear equations serially in which we have to create an augmented matrix of given n linear equations in n variables. The linear system could be represented as

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \cdots\cdots\cdots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \cdots\cdots\cdots + a_{2n}x_n = b_2 \\
&a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \bullet\bullet\bullet\bullet\bullet\bullet\bullet\bullet + a_{3n}x_n = b_3 \\
&\bullet \quad \bullet \quad \bullet \qquad\qquad \bullet \quad \bullet \\
&a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 \bullet\bullet\bullet\bullet\bullet\bullet\bullet\bullet + a_{in}x_n = b_i \\
&\bullet \quad \bullet \quad \bullet \qquad\qquad \bullet \quad \bullet \\
&a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 \bullet\bullet\bullet\bullet\bullet\bullet\bullet\bullet + a_{nn}x_n = b_n
\end{aligned}
$$

Writing this equation in matrix form,

Ax = b

where A is the coefficient matrix

x is the matrix of unknowns and

b is the result matrix of equations

The Gaussian elimination method is used to solve the linear system of equations serially. This includes forward elimination and backward substitution. The goal of forward elimination is to transform the coefficient matrix into an upper triangular matrix. At the end of n-1 Forward elimination steps, the system of equations will looks like

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1$$
$$a'_{22}x_2 + a'_{23}x_3 + \ldots + a'_{2n}x_n = b'_2$$
$$a''_{33}x_3 + \ldots + a''_{3n}x_n = b''_3$$
$$a_{nn}^{(n-1)}x_n = b_n^{(n-1)}$$

The backward substitution then solves each equation from last equation to compute the solution vector x

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

$$x_i = \frac{b_i^{(i-1)} - a_{i,i+1}^{(i-1)}x_{i+1} - a_{i,i+2}^{(i-1)}x_{i+2} - \ldots - a_{i,n}^{(i-1)}x_n}{a_{ii}^{(i-1)}} \text{ for } i = n-1,\ldots,1$$

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)}x_j}{a_{ii}^{(i-1)}} \text{ for } i = n-1,\ldots,1$$

The main drawback of this method is that it is very slow procedure and because of this it takes time, so it is not feasible to solve massive linear equations on serial machine.

The first method implemented in parallel implementation is CUDA implementation on GPU. In GPU implementation, two kernels are written, one for scaling the row which contains the diagonal elements and on for performing row operations on rows which contain non diagonal elements. These two kernels are run as many times as the number of rows. In each kernel, indexing through the matrix is done using x and y coordinates of the thread. Each block in the GPU is defines as two dimensional with 16 as the tile width. Tile width is chosen as 16 for effective utilization of stream multiprocessor threads. Matrix dimensions define the dimensions of the grid i.e. total number of blocks in the 2D grid.
Other various helper functions were implemented to support the above-mentioned kernel

```
__global__ void findOptimalRowIndexKernel(int row_index, matrix  dev_AB, int row_size, double  epsilon, int * new_row_index) {

    int i = row_index + threadIdx.x + 1;

    if ((dev_AB[i * row_size + row_index] < (-MIN_DIAG_VALUE + epsilon) || dev_AB[i * row_size + row_index] > (MIN_DIAG_VALUE - epsilon))) {
        *new_row_index = i;
    }
}

__global__ void getColumn(int column_index, matrix  dev_AB, int row_size, m_elem * array_for_storing) {
    int i = threadIdx.x;
    array_for_storing[i] = dev_AB[i * row_size + column_index];
}

__global__ void swapRows(int source_row_index, int dest_row_index, matrix dev_AB, int row_size) {
    int j = threadIdx.x;
    m_elem  tmp;
    tmp = dev_AB[source_row_index * row_size + j];
    dev_AB[source_row_index * row_size + j] = dev_AB[dest_row_index * row_size + j];
    dev_AB[dest_row_index * row_size + j] = tmp;

}
```

The kernels defined above launch with a single block with thread count as number of rows in the matrix

```
__global__ void gaussEliminationKernel(int row_index, double * zeroing_factors, matrix dev_AB, int row_size)
{
    int i = blockIdx.x;
    int j = threadIdx.x;

    if (i == row_index) {
        return;
    }
    dev_AB[i * row_size + j] -= zeroing_factors[i] * dev_AB[row_index * row_size + j];
}

__global__ void getZeroingFactorsKernel(int row_index, double * zeroing_factors, matrix dev_AB, int row_size)
{
    int i = threadIdx.x;

    if (i == row_index) {
        return;
    }
    zeroing_factors[i] = dev_AB[i *row_size + row_index] / dev_AB[row_index * row_size + row_index];
}
```

These kernels execute the gaussian elimination with n blocks and n threads.

```
    // Copy output matrix from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(A, dev_AB, (size) * sizeof(m_elem), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMempy  failed for %d elements! ", size);
        goto Error;
    }


    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Sth was broken launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }

Error:
    cudaFree(dev_AB);
    cudaFree(device_tmp_column);
    cudaFree(new_row_index);
    cudaFree(zeroing_factors);
    free(host_tmp_column);

    return cudaStatus;
```

```
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed for variable device_tmp_column !");
    goto Error;
}

int * local_new_row_index = (int*)malloc(sizeof(int));

if (local_new_row_index == NULL) {
    fprintf(stderr, "Malloc on CPU failed for new_row_index !");
    goto Error;
}
double diagonal_value = 0;
int * new_row_index;
cudaStatus = cudaMalloc((void**)&new_row_index, sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed for new_row_index!");
    goto Error;
}
```

Error handling code is placed at various points in the kernel to check if the synchronization and memory copy is executed properly. If not, the program throws an error with appropriate message from GPU.

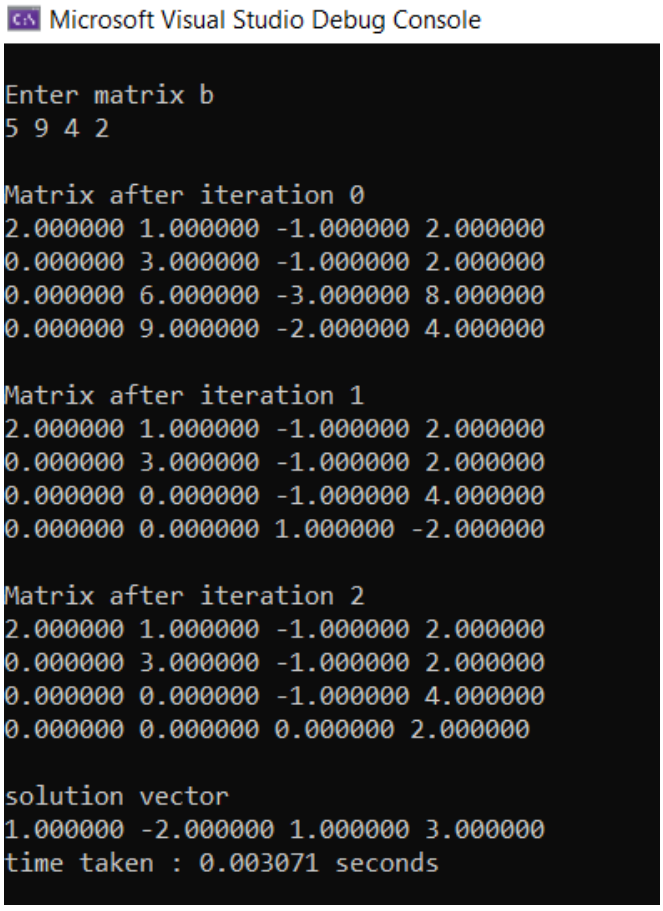
The second method implemented is the parallel implementation using openmp library.

In this parallel implementation, the matrices written in the form Ax = b for solving system of equations are partitioned using column partition strategy.

In this kind of partitioning, each processor gets SIZE/MAXTHREADS of columns from the coefficient matrix A, where SIZE is the order of the matrix and MAXTHREADS is the number of threads or CPUs.

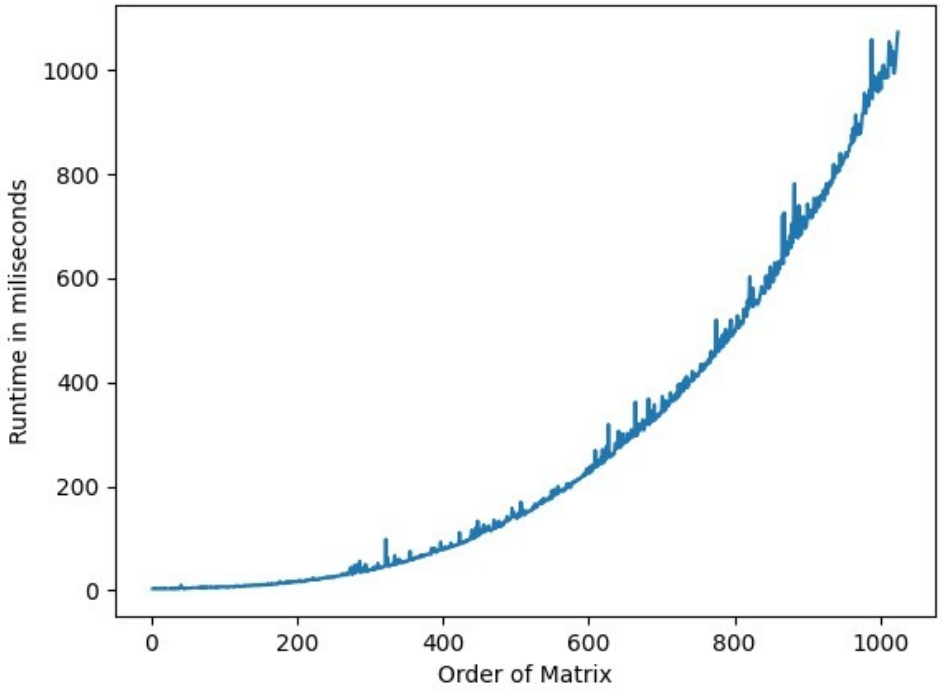
PIPELINIG METHOD

First the processor performs elimination phase on the columns partitioned to with the pivot value of the columns above the current processor's or thread's partition. When the value of the pivotal element is equal to the diagonal element of the current process it first performs the division step and then performs the elimination phase on the columns below the pivotal column of the current processor. Each processor or thread follows the same method. At the end, we get the matrices in the eliminated form Uy = b which is then used for back substitution. The back substitution is performed in the same way with each processor performing operations on the columns allotted, to obtain the solution vector. This is illustrated in the figure below.

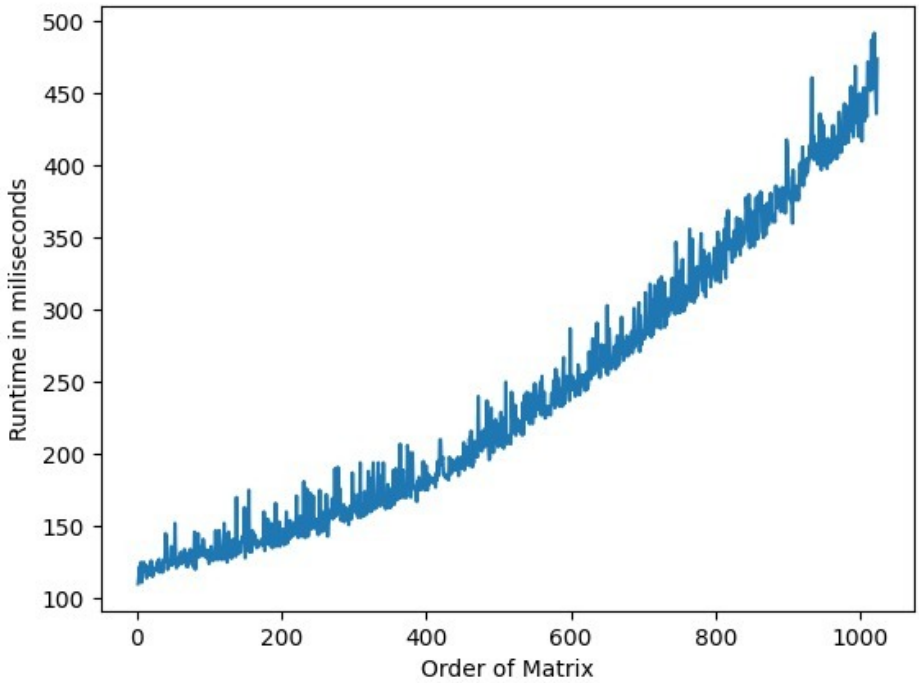


REPORTS AND OBSERVATIONS

Plot of order of matrix vs Runtime in milliseconds for openmp implementation



Plot of order of matrix vs Runtime in milliseconds for CUDA implementation



The figure below displays the sample input and output for the openmp implementation

Microsoft Visual Studio Debug Console

```
Enter matrix b
5 9 4 2

Matrix after iteration 0
2.000000 1.000000 -1.000000 2.000000
0.000000 3.000000 -1.000000 2.000000
0.000000 6.000000 -3.000000 8.000000
0.000000 9.000000 -2.000000 4.000000

Matrix after iteration 1
2.000000 1.000000 -1.000000 2.000000
0.000000 3.000000 -1.000000 2.000000
0.000000 0.000000 -1.000000 4.000000
0.000000 0.000000 1.000000 -2.000000

Matrix after iteration 2
2.000000 1.000000 -1.000000 2.000000
0.000000 3.000000 -1.000000 2.000000
0.000000 0.000000 -1.000000 4.000000
0.000000 0.000000 0.000000 2.000000

solution vector
1.000000 -2.000000 1.000000 3.000000
time taken : 0.003071 seconds
```

The figure below displays the sample input and output for the CUDA implementation

```
umashankar  Umashankar-PC  mnt > d > Projects > ParallelSystemLinearEquationsSolver-master > ParallelLinearEquationsSolver  %  cat test/test1.txt
4
15 -2 -6 0
-2 12 -4 -1
-6 -4 19 -9
0 -1 -9 21
300 0 0 0

//expected solution
26.5 9.35 13.3 6.13
umashankar  Umashankar-PC  mnt > d > Projects > ParallelSystemLinearEquationsSolver-master > ParallelLinearEquationsSolver  %  ./a.exe < test/test1.txt
4
1.000000 0.000000 -0.000000 0.000000
0.000000 1.000000 -0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 -0.000000 1.000000
26.549158 9.353702 13.254994 6.126126
umashankar  Umashankar-PC  mnt > d > Projects > ParallelSystemLinearEquationsSolver-master > ParallelLinearEquationsSolver  %
```

RESULTS AND DISCUSSIONS

The methods developed in openmp in C and CUDA GPU using the tiling mechanism has successfully implemented for massive linear equations over sizes of 1000. The task is completed by distributing the work among different threads to achieve the optimal speedup when compared to parallel implementation in openmp.

CONCLUSIONS

This project was done as a part of the Parallel Computing course under Dr. Nagegowda KS and is a method of solving massive linear equations using CUDA.

The results were compared to tasks run on openmp parallel implementation for various sizes of matrices and the speed up has been plotted and shown in the observation section. From the plots, conclude that the for smaller values of n, OMP performs better than CUDA implementation but for massive values of n, CUDA outperforms OMP implementation.

Further work can be done on this to come up with more efficient mechanisms using different CUDA libraries.

BIBLIOGRAPHY

1. M. Garland et al., "Parallel Computing Experiences with CUDA," in IEEE Micro, vol. 28, no. 4, pp. 13-27, July-Aug. 2008, doi: 10.1109/MM.2008.57.]
2. Using gauss - Jordan elimination method with CUDA for linear circuit equation systems Nesrin Aydin Atasoya *, Baha Senb , Burhan Selcukc
3. Z. Zhang, Q. Miao and Y. Wang, "CUDA-Based Jacobi's Iterative Method," 2009 International Forum on Computer Science-Technology and Applications, Chongqing, 2009, pp. 259-262, doi: 10.1109/IFCSTA.2009.68.
4. CUDA crash course https://www.youtube.com/playlist?list=PLxNPSjHT5qvu4Q2UElj3HUCh2lpSooQWo
5. CUDA-c-programming guide - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html