Indian Institute of Information Technology, Allahabad

Parallel LU Decomposition

Submitted to:
Dr. Anshu Anand

GROUP MEMBERS-
Sarvesh **IIT2019031**
Rajat Mehra **IIT2019034**
Abhishek Bhaware **IIT2019037**
Ritik Mehra **IIT2019038**

Contents:

1. PROBLEM STATEMENT

Find the Lower Upper Decomposition of a matrix.

For the course project we finalized and worked on the LU Decomposition of a matrix. Factorization of a matrix in two matrices i.e. Lower and Upper is known as LU Decomposition. A matrix "mat" is decomposed into two matrices 'L' and 'U' whose product can retrieve us back to the original matrix "mat".

2. INTRODUCTION

As per the problem here, we are required to decompose a matrix into two other matrices, say L and U. The matrix 'L' has non-zero values in the lower diagonal region of the matrix while the matrix 'U' has non-zero values in the upper diagonal region of the matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

The image above clearly shows how a matrix gets decomposed into L and U matrices. For a matrix having determinant 0, then pure LU Decomposition for that matrix doesn't exist.

Here, in the project we've implemented the LU Decomposition using three methods. First one is the sequential method and the other two are parallel. For parallel we've implemented the same using OpenMP and CUDA (GPU based).

The LU Decomposition can be used in finding the determinant of a matrix. Once we are done finding the 'L' and 'U' matrices we can easily find the determinant of the matrix by doing the product of their diagonal elements.

3. ALGORITHM DESCRIPTION

For all the three implementations we first fill the matrix with random values. Then, as mentioned above we first checked if the matrix is invertible or not. To do so, we used the concept of diagonally dominant matrix. All the diagonally dominant matrices are always invertible. So, we updated the randomly filled matrix and made sure that it is invertible.

i) Sequential Algorithm:

We first implemented the sequential program. In the serial implementation we first run a for loop N times, where N is the dimension of the matrix. Inside the for loop, there exists two other for loops, one for filling the L matrix and the other for the U matrix. We know that the elements in the lower diagonal region of the L matrix are non-zero therefore we set the values at all other places as 0 inside the first inner for loop. Similarly, in the U matrix, the elements are non-zero only if they exist in the upper diagonal region hence we set the elements at other places as 0 inside the second inner loop. Now, in matrix L, to fill the matrix at remaining places, we first set the element equal to that of the original matrix and then run a for loop and set matrix element using mathematical calculations.

Finally, after iterating the loop we're left with matrices 'L' and 'U'.

ii) OpenMP:

Now in the OpenMP we just have to see what parts we can parallelize there are 2 parts :

First we can use to initialize the Matrix using the pragma omp for where we are initializing the matrices A, L and U

Second: We are using the shared variables that are A, L and U that are your Matrices.

This is the main one for the LU decomposition function where your code is going to parallelize. Over here we have parallelized the second loop which is doing the calculation rowwise and colwise and due to size issues we have taken this as pragma schedule.

The last thing is setting the number of threads and for that we have tried with several thread numbers and we are getting better results in the number range 40-45 and by using too many threads we are losing the efficiency but this Openmp works really well and compared to sequential.

iii) Cuda:

We have done cuda implementation a little bit differently than the other previous implementations.Our CUDA implementation also uses the shared memory method and each thread keeps the copy of pivot row because the time to call global memory again and again is high. This implementation contains 2 kernel functions.

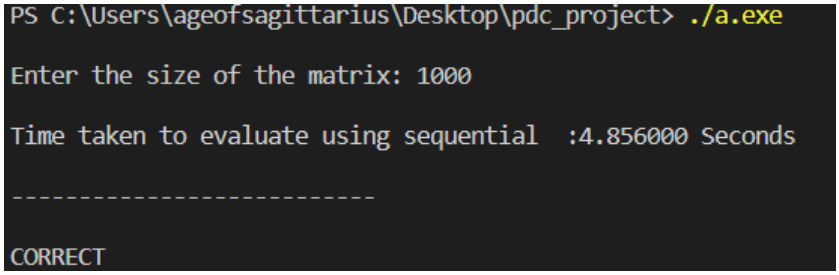
fun1 kernel helps to find the appropriate index value that each thread has to use for its corresponding index value in which they are operating.

The fun2 kernel is used for row reduction so that we can calculate the lower and upper matrix. So Instead of creating separate matrices for lower and upper matrices. Each thread has its own copy of matrices to write value. With the help of this we reduce the overhead and runtime.

The program calls the kernels N times which is equivalent to the size of the matrix N and in each time 1 column is eliminated for each N-1 remaining rows. Since there are N pivots for triangular matrices so it has to be N times

4. Results

The output of the sequential program is as follows: The first screenshot shows the execution time for a large matrix and the second one shows the original matrix along with the resultant matrices with the execution time and correctness.



```
Enter the size of the matrix: 8

The Original Matrix is:
2.000   8.000   5.000   1.000   10.000  5.000   9.000   9.000
3.000   5.000   6.000   6.000   2.000   8.000   2.000   2.000
6.000   3.000   8.000   7.000   2.000   5.000   3.000   4.000
3.000   3.000   2.000   7.000   9.000   6.000   8.000   7.000
2.000   9.000   10.000  3.000   8.000   10.000  6.000   5.000
4.000   2.000   3.000   4.000   4.000   5.000   2.000   2.000
4.000   9.000   8.000   5.000   3.000   8.000   8.000   10.000
4.000   2.000   10.000  9.000   7.000   6.000   1.000   3.000

The resultant matrices are:

L Matrix is:
2.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
3.000   -7.000  0.000   0.000   0.000   0.000   0.000   0.000
6.000   -21.000 -2.500  0.000   0.000   0.000   0.000   0.000
3.000   -9.000  -3.571  13.286  0.000   0.000   0.000   0.000
2.000   1.000   4.786   -15.543 11.351  0.000   0.000   0.000
4.000   -14.000 -4.000  8.200   -4.514  3.491   0.000   0.000
4.000   -7.000  -0.500  0.400   -6.049  -0.753  3.647   0.000
4.000   -14.000 3.000   -13.400 21.157  -3.962  -6.675  4.564

U Matrix is:
1.000   4.000   2.500   0.500   5.000   2.500   4.500   4.500
0.000   1.000   0.214   -0.643  1.857   -0.071  1.643   1.643
0.000   0.000   1.000   3.800   -4.400  4.600   -4.200  -4.600
0.000   0.000   0.000   1.000   -0.376  1.075   -0.430  -0.613
0.000   0.000   0.000   0.000   1.000   -0.020  0.773   0.603
0.000   0.000   0.000   0.000   0.000   1.000   -0.798  -1.046
0.000   0.000   0.000   0.000   0.000   0.000   1.000   1.181
0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

Time taken to evaluate using sequential  :1.926000 Seconds

--------------------------

CORRECT
```

The output of the OpenMP is: The first screenshot shows, the original matrix, the resultant L and U matrices along with the time and correctness while the second screenshot shows the time for a large matrix.

```
Enter the size of the matrix: 8

The Original Matrix is:
2.000   8.000   5.000   1.000   10.000  5.000   9.000   9.000
3.000   5.000   6.000   6.000   2.000   8.000   2.000   2.000
6.000   3.000   8.000   7.000   2.000   5.000   3.000   4.000
3.000   3.000   2.000   7.000   9.000   6.000   8.000   7.000
2.000   9.000   10.000  3.000   8.000   10.000  6.000   5.000
4.000   2.000   3.000   4.000   4.000   5.000   2.000   2.000
4.000   9.000   8.000   5.000   3.000   8.000   8.000   10.000
4.000   2.000   10.000  9.000   7.000   6.000   1.000   3.000
Enter the number of threads: 8

The resultant matrices are:

L Matrix is:
2.000   0.000   0.000   0.000   0.000   0.000   0.000
3.000   -7.000  0.000   0.000   0.000   0.000   0.000   0.000
6.000   -21.000 -2.500  0.000   0.000   0.000   0.000   0.000
3.000   -9.000  -3.571  13.286  0.000   0.000   0.000   0.000
2.000   1.000   4.786   -15.543 11.351  0.000   0.000   0.000
4.000   -14.000 -4.000  8.200   -4.514  3.491   0.000   0.000
4.000   -7.000  -0.500  0.400   -6.049  -0.753  3.647   0.000
4.000   -14.000 3.000   -13.400 21.157  -3.962  -6.675  4.564

U Matrix is:
1.000   4.000   2.500   0.500   5.000   2.500   4.500   4.500
0.000   1.000   0.214   -0.643  1.857   -0.071  1.643   1.643
0.000   0.000   1.000   3.800   -4.400  4.600   -4.200  -4.600
0.000   0.000   0.000   1.000   -0.376  1.075   -0.430  -0.613
0.000   0.000   0.000   0.000   1.000   -0.020  0.773   0.603
0.000   0.000   0.000   0.000   0.000   1.000   -0.798  -1.046
0.000   0.000   0.000   0.000   0.000   0.000   1.000   1.181
0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

Time taken to evaluate using OpenMP  :1.339000 Seconds

CORRECT
```

```
PS C:\Users\ageofsagittarius\Desktop\pdc_project> g++ omp.cpp -o omp.exe -fopenmp
PS C:\Users\ageofsagittarius\Desktop\pdc_project> ./omp.exe

Enter the size of the matrix: 1000
Enter the number of threads: 8

Time taken to evaluate using OpenMP  :3.425000 Seconds

CORRECT
PS C:\Users\ageofsagittarius\Desktop\pdc_project> g++ omp.cpp -o omp.exe -fopenmp
PS C:\Users\ageofsagittarius\Desktop\pdc_project> ./omp.exe

Enter the size of the matrix: 1000
Enter the number of threads: 16

Time taken to evaluate using OpenMP  :3.218000 Seconds

CORRECT
```

The output of the CUDA program is: The first screenshot shows, the original matrix, the resultant L and U matrices along with the time and correctness while the second screenshot shows the time for a large matrix.

```
The original matrix is:
42.000  7.000   8.000   6.000   4.000   6.000   7.000   3.000
10.000  45.000  3.000   8.000   1.000   10.000  4.000   7.000
1.000   7.000   49.000  7.000   2.000   9.000   8.000   10.000
3.000   1.000   3.000   38.000  8.000   6.000   10.000  3.000
3.000   9.000   10.000  8.000   45.000  7.000   2.000   3.000
10.000  4.000   2.000   10.000  5.000   50.000  9.000   5.000
6.000   1.000   4.000   7.000   2.000   1.000   26.000  4.000
3.000   1.000   7.000   2.000   6.000   6.000   5.000   33.000

The resultant matrices are:

L Matrix is:
42.000  0.000   0.000   0.000   0.000   0.000   0.000   0.000
10.000  43.333  0.000   0.000   0.000   0.000   0.000   0.000
1.000   6.833   48.637  0.000   0.000   0.000   0.000   0.000
3.000   0.500   2.416   37.206  0.000   0.000   0.000   0.000
3.000   8.500   9.214   5.180   43.285  0.000   0.000   0.000
10.000  2.333   0.036   8.213   2.362   46.828  0.000   0.000
6.000   0.000   2.857   5.801   0.129   -1.101  23.271  0.000
3.000   0.500   6.416   0.728   5.314   4.044   3.055   30.882

U Matrix is:
1.000   0.167   0.190   0.143   0.095   0.143   0.167   0.071
0.000   1.000   0.025   0.152   0.001   0.198   0.054   0.145
0.000   0.000   1.000   0.120   0.039   0.154   0.153   0.184
0.000   0.000   0.000   1.000   0.205   0.137   0.245   0.061
0.000   0.000   0.000   0.000   1.000   0.064   -0.038  -0.011
0.000   0.000   0.000   0.000   0.000   1.000   0.113   0.074
0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.119
0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

Time taken to evaluate using sequential  :0.117809 Seconds

---------------------------

CORRECT



The dimension of the matrix is: 1000

Time taken to evaluate using sequential  :0.622871 Seconds

---------------------------

CORRECT
```

5. CONCLUSION

In this project, we did implement LU Decomposition using three different methods and also checked for their correctness. All the three methods passed the correctness test and gave the desired outputs. We also compared their execution times and found out that the parallel algorithms performed much better than the sequential one. It can be clearly seen that the sequential program is very expensive in terms of time whereas the parallel ones are not. In the OpenMP program the execution time decreases with increase in the number of threads. Also, the CUDA program is the most efficient one and takes very less time as compared to both sequential and openMP. The problem of LU Decomposition is also helpful in various linear algebra problems as well like finding the determinant of a matrix. Therefore, to design an algorithm taking less time in execution than the usual sequential one does help in solving a number of problems.

6. Contribution

Sarvesh: Implementation of sequential and openMP program

Rajat: Implementation of CUDA program and testing for the correctness

Abhishek: Implementation of CUDA program and making report

Ritik: Implementation of openMP program and testing for the correctness

7. Reference

[1] Matrix decomposition Wikimedia Foundation Inc., [online] Available:

[2] Parallelizing doolittle algorithm using TBB 11-13 December 2014 Sushil Kumar Sah,Dinesh Naik

[3] GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis December 201X Kai He Student Member, IEEE, Sheldon X.-D. Tan Senior Member, IEEE Hai Wang Member, IEEE and Guoyong Shi Senior Member, IEEE

[4] Investigation of the performance of LU decomposition method using CUDA © 2011 Published by Elsevier Ltd. Caner Ozcan Baha Sen