ME 766 Course Project

Matrix Decomposition

Gopalan Iyengar (19D170009)
Om Prabhu (19D170018)
Punit Madia (190100096)
Sangraam Patwardhan (19D100017)

Block algorithm for LU decomposition

$$\begin{array}{|c|c|}\hline A_{00} & A_{01} \\ \hline A_{10} & A_{11} \\ \hline \end{array} = \begin{array}{|c|c|}\hline L_{00} & 0 \\ \hline L_{10} & L_{11} \\ \hline \end{array} * \begin{array}{|c|c|}\hline U_{00} & U_{01} \\ \hline 0 & U_{11} \\ \hline \end{array}$$

LU Decomposition of a Matrix

A nonsingular matrix $\boldsymbol{A} \in \mathbb{R}^{nxn}$ is factored into a product of lower and upper triangular matrices $\boldsymbol{L} \in \mathbb{R}^{nxn}$ and $\boldsymbol{U} \in \mathbb{R}^{nxn}$ respectively such that $\boldsymbol{A} = \boldsymbol{LU}$.

A sequential "right-looking" algorithm loops over columns and updates columns to its right based on the current column. The pseudo-code shown alongside describes the algorithm.

This helps solve a linear system Ax=b, by reducing the complexity of the calculations y=Ux, and b=Ly. Since, U and L are triangular, simple substitution solving is applicable.

The general procedure:

```
for i=0 to N-1  /*loop over columns*/
      for j = i to N-1
            /*update L, nested loop to dynamically
            compute 'sum' parameter*/
      end
      for k = i+1 to N-1
            /*update U*/
      end
end
```

LU Decomposition of a Matrix

We looked at 3 different methods of LU decomposition. All the methods have a theoretical complexity of $O(n^3)$

- Gaussian elimination algorithm: Elementary row operations on matrices to reduce it to an upper triangular "row echelon form"
- Doolittle algorithm: Factorization into unit diagonal lower triangular matrix and upper triangular matrix
- Crout's method: Factorization into lower triangular matrix and unit diagonal upper triangular matrix

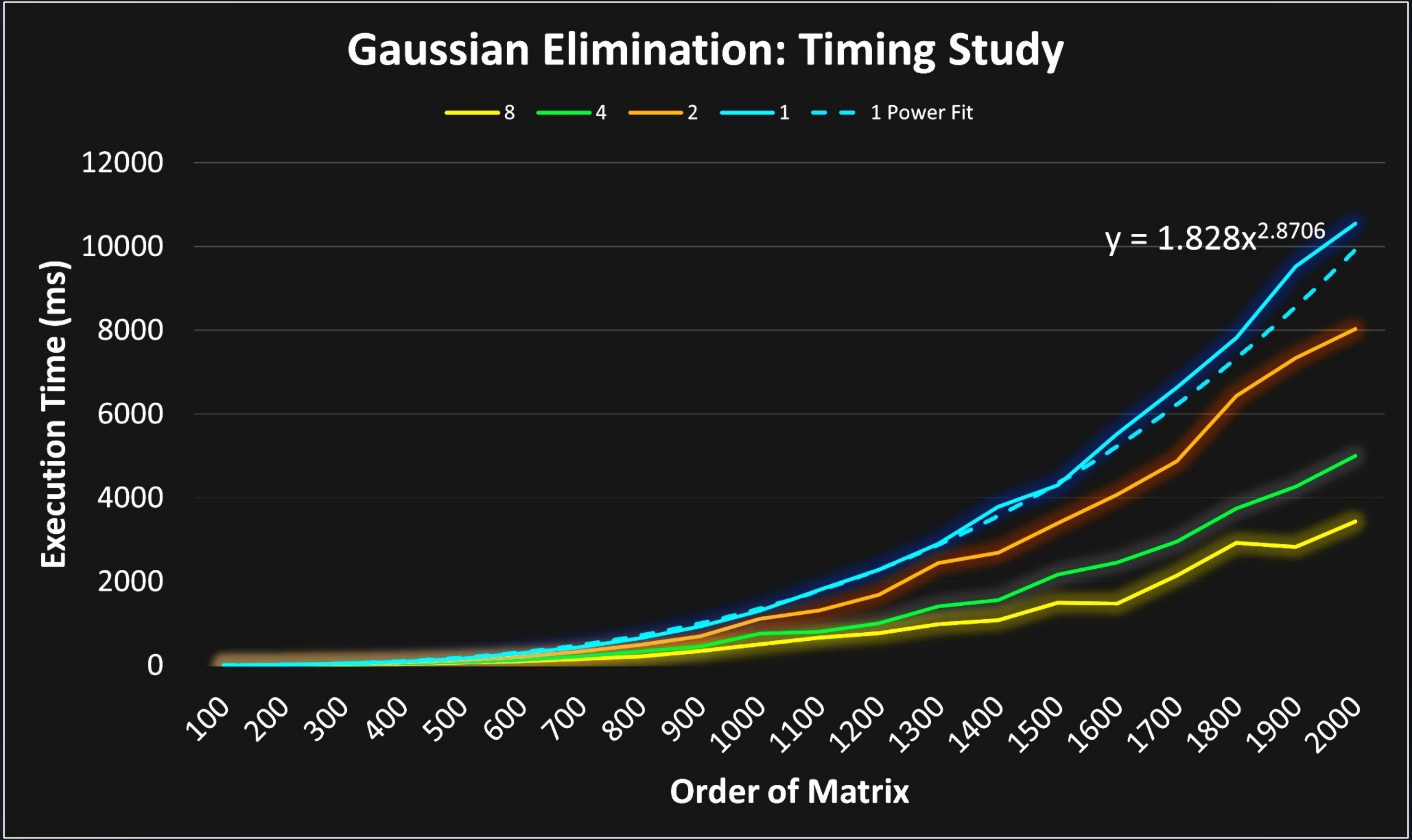
Timing Study using OpenMP

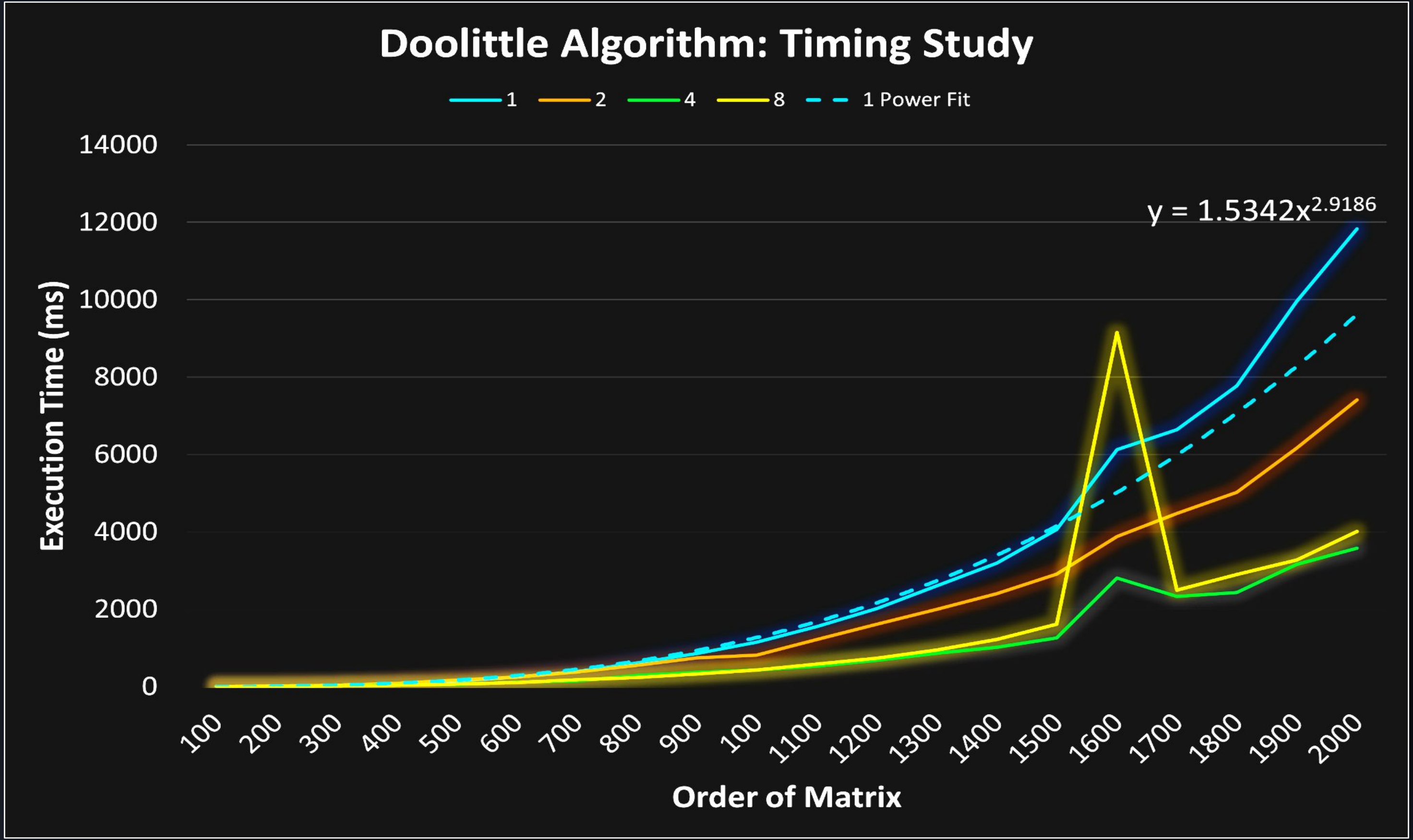
We parallelized the 3 methods of LU decomposition using OpenMP and carried out a timing study on each of the codes. The specifics are listed below:

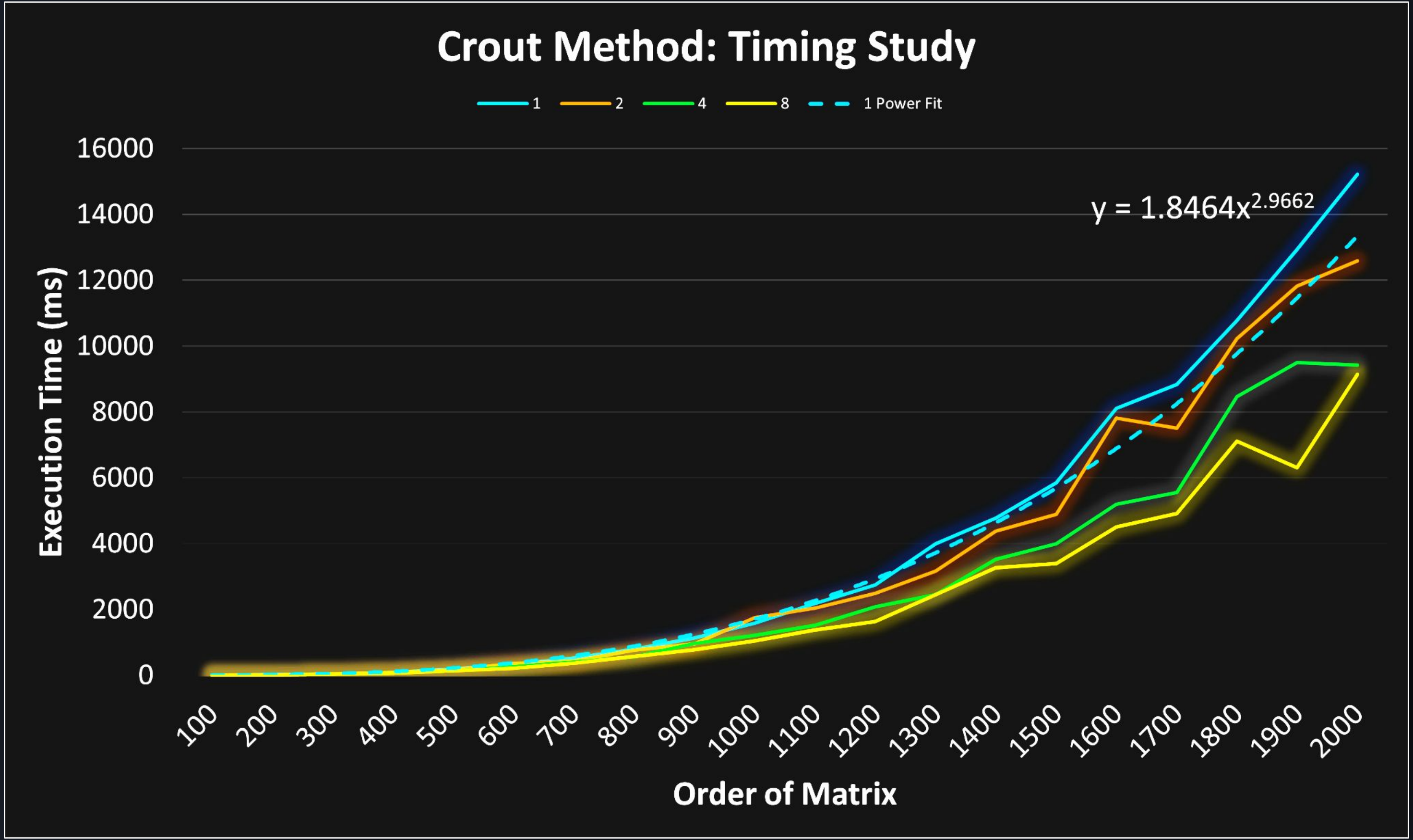
- Order of the matrix *N* was varied from 100 to 2000 in steps of 100 (possible to go beyond 2000, but time explodes)
- Number of threads was varied between 1, 2, 4 and 8 for each value of *N*

Initially, the code caused a segmentation error at about *N*=800, which was corrected by dynamic declaration of variables.

The entire timing study was performed on a system running Ubuntu 20.04 (alongside a native Windows 10 operating system) on an Nvidia GTX 1650M GPU and a quad-core Intel i5-9300H CPU.


Gaussian Elimination: Timing Study
8
4
2
1
1 Power Fit
12000
10000
8000
6000
4000
2000
0
Execution Time (ms)
$y = 1.828x^{2.8706}$
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
Order of Matrix


Doolittle Algorithm: Timing Study
1
2
4
8
1 Power Fit
y = 1.5342x^2.9186
Execution Time (ms)
0
2000
4000
6000
8000
10000
12000
14000
100
200
300
400
500
600
700
800
900
100
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
Order of Matrix


Crout Method: Timing Study
1
2
4
8
1 Power Fit
y = 1.8464x^2.9662
Execution Time (ms)
0
2000
4000
6000
8000
10000
12000
14000
16000
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
Order of Matrix

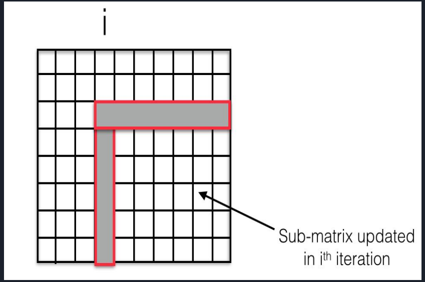
Results

From the graphs, we can infer that:

1. The execution time for a given order of matrix *N*, decreases with increase in number of threads, up till a certain number of threads. eg. 4 threads perform better for Doolittle's algorithm than 8 threads. Generally, for an c-core CPU, performance increases uptil c threads, then decreases.
2. The execution time for a given number of threads, increases proportionally with the x^th power of the order of matrix n, i.e. $t \propto n^x$, such that $3>x>1$. This is backed by the fact that the theoretical complexity of the algorithms is $O(n^3)$.
3. It is seen that the Gaussian elimination implementation takes the least time for 1 thread(serial implementation), at a glance from the power approximations of the curves.
4. These power fit approximations also provide an upper bound to the time curves for higher number of threads.
5. Also, it is observed that the Doolittle algorithm and Gaussian elimination have higher parallelization possibility than the Crout method, since the exponent of the Crout method is higher.

CuDA Implementation

We implemented a parallel version of the LU factorization algorithm in CuDA, to account for higher order matrices. The algorithm was slightly modified to enable better mapping to a GPU - this was done by separating the loops which involve updating the L and the U parts of the matrix.

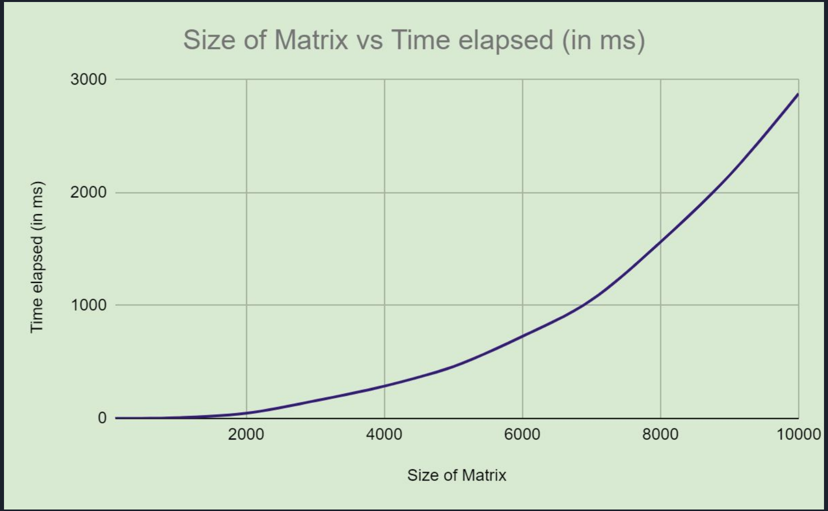


LU - Matrix update at ith iteration

The figure above shows the part of the matrix that is updated at the ith iteration. To begin with, the most simple approach was implemented wherein each element of this sub-matrix is assigned a thread and a kernel with required number of blocks and threads per block is launched. First, the sub-matrix is divided into blocks as before, but within the block only one thread per row of the block is launched (instead of one thread per element). Second, the same thing is done with columns - one thread per column of the block is launched.

LU decomposition in CUDA

On increasing the size of matrix, the time required for LU decomposition also increases. When we plot a graph between size of matrix vs. time elapsed, we get a graph of exponential nature.



References

- http://www.mymathlib.com/matrices/linearsystems/doolittle.html
- https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination
- https://github.com/roguexray007/LU-factorisation
- https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/
- MA214: Introduction to Numerical Analysis course content

Contribution

- **Gopalan:**
  - OpenMP implementation of Gaussian elimination and Crout’s method
  - Graphs based on timing study
  - Presentation
- **Punit:**
  - Report
  - CuDA implementation
  - Presentation
- **Om:**
  - OpenMP implementation of Doolittle algorithm
  - Code execution and data collection for timing study
  - Presentation
- **Sangraam:**
  - Report

Thank You!